UMA ABORDAGEM OBJETIVA

MODELOS DE LINGUAGEM COM PYTHON

GISELDO NEO

ALANA NEO

Índice

Capítulo 1

Explicando Transformers

Bem Vindos!

Baixar PDF

Este livro estabelece as fundações teóricas e práticas necessárias para trabalhar com Modelos de Linguagem, ou *Large Language Models* (LLMs).

Capítulo 2

Introdução aos LLMs

Exploraremos a transição arquitetural que permitiu o surgimento de modelos como GPT e BERT, seguida de um guia para a configuração de um ambiente de desenvolvimento de alto desempenho com Python e PyTorch.

> **Objetivos de aprendizagem**
>
> Ao final deste capítulo, você deverá ser capaz de explicar o que um modelo de linguagem calcula, distinguir parâmetro de hiperparâmetro, comparar n-gramas, RNNs e Transformers e reconhecer as três famílias mais comuns de arquiteturas Transformer.

2.1 O Que São LLMs?

Neste livro, vamos destrinchar o funcionamento dos Large Language Models (LLMs), mergulhando nos detalhes da arquitetura Transformer e abrindo a 'caixa-preta' por trás dessas tecnologias.

Os Modelos de Linguagem de Grande Escala, conhecidos pela sigla em inglês LLM (Large Language Models), representam uma das conquistas mais impressionantes da inteligência artificial moderna. Trata-se de sistemas computacionais projetados para compreender, gerar e ma-

nipular texto em linguagem natural de maneira que se assemelha — e em muitos casos, ultrapassa — a capacidade humana de comunicação escrita.

Um LLM é, fundamentalmente, um sistema matemático sofisticado que aprende padrões linguísticos a partir de vastas quantidades de texto. Diferentemente dos programas de computador tradicionais, que seguem instruções explícitas escritas por programadores, um LLM "aprende" a linguagem através da exposição massiva a exemplos de texto. Essa abordagem é comumente denominada aprendizado de máquina, ou machine learning, e mais especificamente, aprendizado profundo, ou deep learning, devido à utilização de redes neurais com múltiplas camadas.

A essência de um modelo de linguagem reside na sua capacidade de prever o que vem a seguir em uma sequência de texto. Quando você digita uma frase incompleta, o modelo é capaz de sugerir palavras ou construções que completem o pensamento de forma coerente. Essa habilidade de previsão probabilística é o fundamento sobre o qual toda a arquitetura dos LLMs foi construída.

Essa descrição pode ser tornada mais precisa. Dado um contexto formado pelos tokens $x_1, \ldots, x_t$, o modelo produz um vetor de *logits*, um valor para cada item do vocabulário. A função *softmax* converte esses valores em uma distribuição $P(x_{t+1} \mid x_{1:t})$. O sistema não recupera uma frase pronta: ele recalcula essa distribuição, escolhe um token segundo uma estratégia de decodificação e repete o processo.

Observe que **token** não é necessariamente uma palavra. Ele pode ser uma palavra inteira, parte de uma palavra, pontuação ou byte. Essa distinção será importante porque o custo de atenção, o tamanho do contexto e a velocidade de geração são medidos em tokens, não em palavras.

> **!** Modelo probabilístico não é banco de dados
>
> Os parâmetros armazenam regularidades distribuídas aprendidas durante o treinamento. Eles não constituem uma tabela confiável

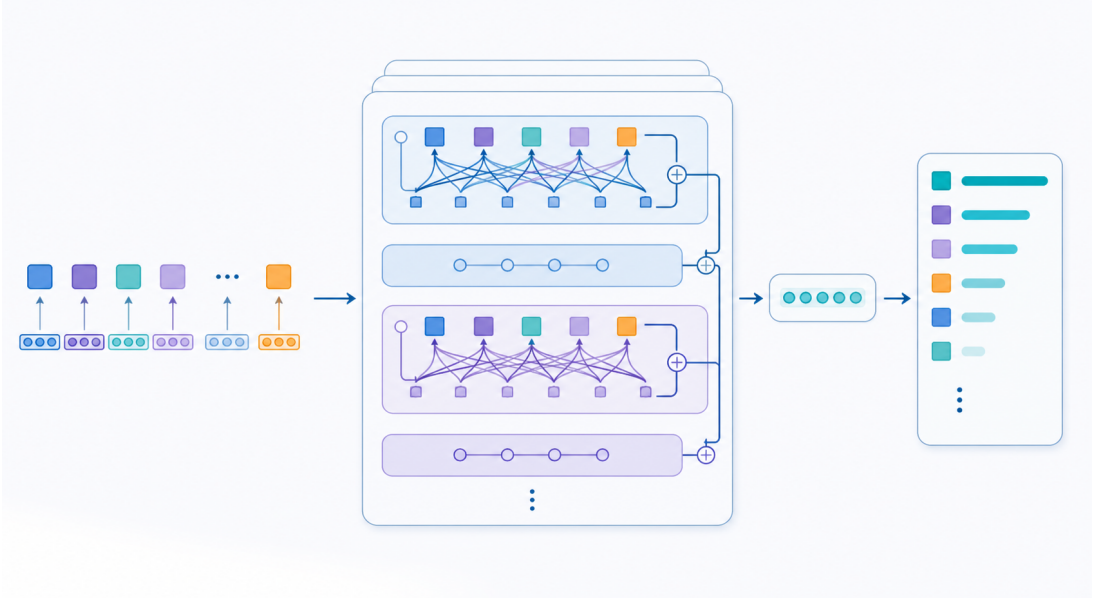


Figura 2.1: Fluxo conceitual de tokens entrando em camadas de atenção e produzindo possibilidades para o próximo token.

de fatos nem garantem que a continuação mais provável seja verdadeira. Fluência e correção factual são propriedades diferentes.

2.2 O que são Redes Neurais

Nas últimas décadas, sistemas avançados de inteligência artificial — como modelos de tradução automática, reconhecimento de imagens e geração de texto e os próprios LLMs — passaram a ser dominados por uma classe específica de técnicas conhecida como **aprendizado profundo (deep learning)**. Embora o termo possa sugerir algo excessivamente complexo ou obscuro, sua base conceitual é relativamente direta: trata-se do uso sistemático de **redes neurais artificiais** para aproximar funções complexas a partir de dados. Veja na figura abaixo:

Em ciência da computação clássica, estamos acostumados a resolver problemas explicitando regras e algoritmos: fornecemos uma função bem definida e esperamos que ela produza a saída correta para qualquer entrada válida. Entretanto, muitos problemas relevantes do mundo real

não admitem uma formulação algorítmica exata. Nesses casos, o que temos não é a função em si, mas apenas exemplos do seu comportamento. Redes neurais surgem exatamente como um mecanismo matemático para lidar com esse cenário.

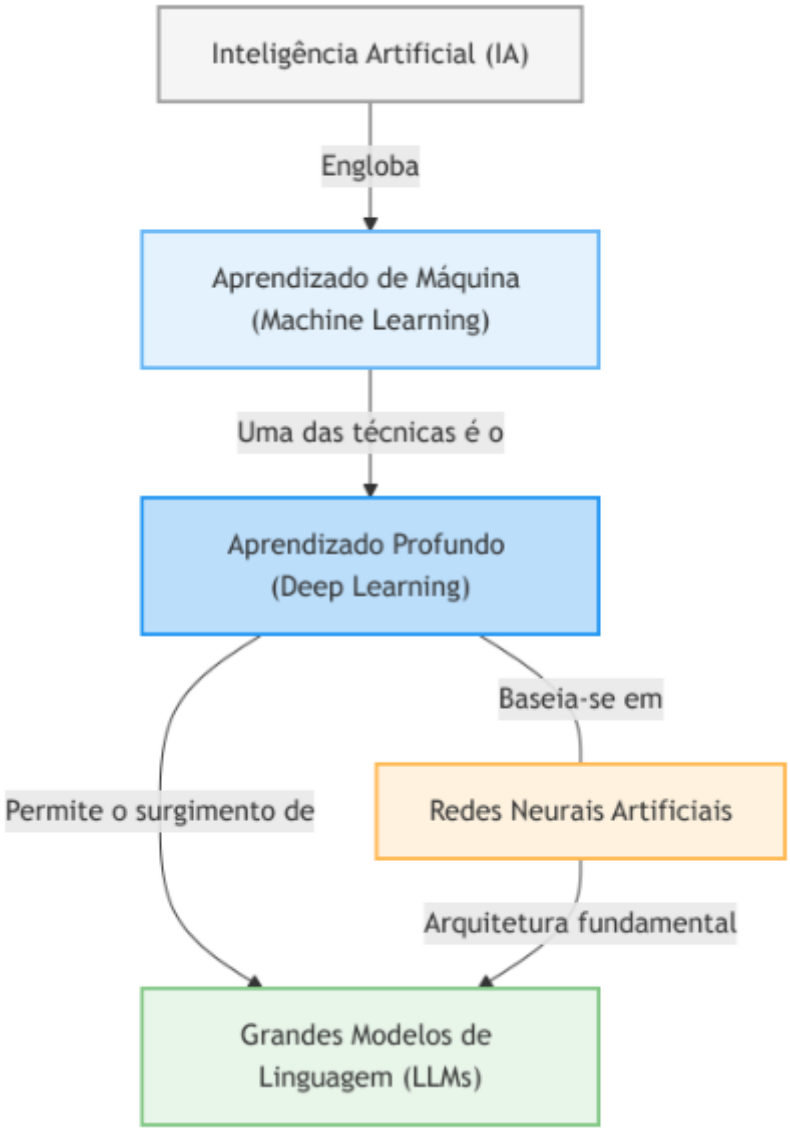


Figura 2.2: Diagrama conceitual

Historicamente, a ideia de redes neurais remonta a 1944, com o trabalho seminal de Warren McCullough e Walter Pitts, que propuseram um modelo matemático simplificado de neurônios biológicos. Desde então, a área passou por ciclos de entusiasmo e frustração. O que explica seu sucesso contemporâneo não é uma mudança conceitual radical, mas a

convergência de três fatores: disponibilidade massiva de dados, avanços em métodos de otimização e aumento expressivo do poder computacional, especialmente com o uso de GPUs.

Do ponto de vista funcional, uma rede neural pode ser entendida como um aproximador universal de funções. Dado um conjunto suficientemente grande de exemplos representativos, a rede é capaz de aprender uma função que mapeia entradas em saídas com erro controlado. Esse é o motivo pelo qual ela pode, por exemplo, estimar a probabilidade de um tumor ser maligno a partir de exames clínicos, mesmo sem existir uma equação fechada que descreva tal decisão.

A estrutura interna de uma rede neural é organizada em **camadas** compostas por unidades chamadas neurônios artificiais. Cada neurônio executa uma operação matemática elementar: recebe um conjunto de valores de entrada, multiplica cada um por um peso associado, soma esses produtos e adiciona um termo constante denominado **viés (bias)**. O resultado dessa soma é então transformado por uma **função de ativação**.

A função de ativação desempenha um papel conceitualmente crucial. Sem ela, toda a rede se reduziria a uma única transformação linear, independentemente do número de camadas. Ao introduzir **não linearidade**, funções como ReLU, sigmoide ou tanh permitem que a rede represente relações altamente complexas entre variáveis. Em termos intuitivos, a função de ativação decide quais sinais devem ser propagados adiante e quais devem ser atenuados ou descartados.

A maioria das arquiteturas clássicas de redes neurais é do tipo **feed-forward**, no qual a informação flui apenas no sentido da entrada para a saída, sem ciclos. Os dados atravessam sucessivamente as camadas, sendo transformados passo a passo, até que a última camada produza a saída final do modelo. Essa saída pode representar uma classe, um valor contínuo ou mesmo uma distribuição de probabilidades, dependendo do problema.

O processo de aprendizado consiste em ajustar iterativamente os pesos e vieses da rede. Inicialmente, esses parâmetros são definidos com

valores aleatórios. A rede então produz uma saída para cada exemplo de treinamento, que é comparada com a saída correta esperada. A diferença entre essas duas quantidades é quantificada por uma **função de perda (loss)**, que mede o quão distante o modelo está do comportamento desejado.

A partir dessa perda, algoritmos de otimização — como o gradiente descendente e suas variantes — ajustam os parâmetros da rede de forma a reduzir progressivamente o erro. Em termos conceituais, o treinamento pode ser visto como uma busca em um espaço de altíssima dimensionalidade por uma configuração de parâmetros que minimize a perda média sobre os dados de treinamento, mantendo capacidade de generalização.

Essa perspectiva permite desmistificar redes neurais. Elas não "pensam" nem "entendem" no sentido humano, mas implementam um mecanismo matemático elegante para capturar regularidades estatísticas em dados. Sua força reside menos em cada neurônio individual e mais no comportamento coletivo emergente de milhares ou milhões dessas unidades trabalhando em conjunto.

É importante enfatizar que esse arcabouço conceitual serve como base para arquiteturas mais sofisticadas, como redes convolucionais, recorrentes e modelos baseados em atenção. Contudo, todas essas variações compartilham o mesmo princípio fundamental aqui descrito: aprender uma função complexa a partir de exemplos por meio de composições sucessivas de transformações simples.

2.3 A Escala dos Modelos Modernos de LLM

Nos modelos de linguagem grande (em inglês Large Language Model), o termo "grande" não é meramente decorativo. Estes modelos são chamados assim porque foram treinados com quantidades massivas de dados textuais e possuem bilhões ou até trilhões de parâmetros ajustáveis. Um parâmetro, neste contexto, é um valor numérico que o modelo aprende

durante o treinamento e que determina como ele processa e transforma a informação.

Para entender a magnitude envolvida, considere que modelos públicos da família Llama já foram disponibilizados em escalas de bilhões a centenas de bilhões de parâmetros. Em modelos proprietários, a contagem e a organização dos parâmetros nem sempre são divulgadas; por isso, é melhor evitar estimativas tratadas como fatos. Cada parâmetro é uma variável numérica pequena, mas o comportamento emerge da interação entre todos eles, dos dados, do objetivo de treinamento e do procedimento de alinhamento.

Um **parâmetro** é aprendido, como um elemento de uma matriz de pesos. Um **hiperparâmetro** é escolhido pelo projetista, como taxa de aprendizado, número de camadas ou dimensão dos embeddings. Separar os dois conceitos ajuda a ler artigos e configurações de modelos sem confundir o que o treinamento descobre com o que o engenheiro define.

2.4 Por Que LLMs São Importantes

A relevância dos modelos de linguagem de grande escala transcende o mero fascínio tecnológico. Eles representam uma mudança na forma como humanos e máquinas interagem. Antes dos LLMs, a comunicação com computadores exigia que os humanos aprendessem linguagens de programação ou interfaces específicas. Com os LLMs, a linguagem natural torna-se o meio primário de comunicação, democratizando o acesso à informação e à automação.

Esses modelos têm aplicações que abrangem praticamente todos os setores da atividade humana. Na medicina, auxiliam no diagnóstico e na pesquisa científica. Na educação, funcionam como tutores personalizados. Na escrita criativa, servem como colaboradores e fontes de inspiração. No atendimento ao cliente, operam como assistentes disponíveis vinte e quatro horas por dia. A versatilidade dos LLMs é uma conse-

quência direta da generalidade do problema que eles resolvem: entender e gerar linguagem.

2.5 A Evolução do Processamento de Linguagem Natural (PLN)

2.5.1 Os Primórdios: Modelos Estatísticos de Linguagem

Para compreender a revolução representada pelos LLMs modernos, é instrutivo recuar no tempo e examinar as abordagens que os precederam. Nas décadas de 1980 e 1990, os modelos de linguagem eram predominantemente estatísticos. Modelos como os n-gramas dominavam o campo, onde a probabilidade de uma palavra aparecer era calculada com base nas n palavras anteriores.

Esses modelos estatísticos tinham limitações severas. Quanto maior o valor de n necessário para capturar contexto relevante, maior a quantidade de dados necessária para estimar todas as combinações possíveis. A maldição da dimensionalidade tornava impossível capturar dependências de longo alcance de forma eficiente. Além disso, esses modelos não conseguiam generalizar para sequências nunca vistas no treinamento,pois eram essencialmente tabelas de lookup probabilístico.

2.5.2 A Era das Redes Neurais Recorrentes

O salto seguinte veio com a introdução das redes neurais recorrentes (RNNs), especialmente as Long Short-Term Memory (LSTM) e Gated Recurrent Units (GRUs). Essas arquiteturas foram capazes de superar as limitações dos n-gramas ao aprender representações densas e contínuas de palavras, os chamados embeddings, e ao processar sequências de forma sequencial, mantendo um estado interno que capturava informação contextual.

As RNNs permitiram, pela primeira vez, que modelos de linguagem capturassem dependências de longo prazo em textos. Uma RNN pode, em teoria, "lembrar" de informações vistas centenas de passos atrás na sequência. Na prática, no entanto, as RNNs sofriam de problemas de vanishing gradients (gradientes que desaparecem durante o treinamento), que dificultavam o aprendizado de dependências muito longas.

- **Processamento Sequencial:** As RNNs processam palavras uma de cada vez, em uma ordem estrita (da esquerda para a direita). O *hidden state* (estado oculto) do passo t depende da entrada atual e do estado oculto do passo $t-1$.
- **O Gargalo (Bottleneck): Impossibilidade de Paralelização:** Como o cálculo do passo t requer o resultado de $t-1$, não é possível utilizar massivamente o paralelismo das GPUs modernas. **Esquecimento de Longo Prazo:** Mesmo com LSTMs, o contexto se dilui em sequências muito longas. O modelo "esquece" o início da frase quando chega ao final.

2.5.3 A Revolução Transformer

Em 2017, um artigo acadêmico intitulado "Attention Is All You Need", escrito por pesquisadores do Google e da Universidade de Toronto, apresentou uma arquitetura que transformaria irreversivelmente o campo da processamento de linguagem natural. O Transformer, como foi denominado, abandonou completamente a ideia de processamento sequencial em favor de um mecanismo denominado auto-atenção, ou self-attention.

O artigo demonstrou que, utilizando exclusivamente mecanismos de atenção, era possível alcançar resultados superiores aos das arquiteturas baseadas em recorrência, com a vantagem adicional de ser altamente paralelizável, permitindo treinamento em hardware gráfico (GPUs) e tensores (TPUs) de forma muito mais eficiente. Portanto, a história moderna do PLN pode ser dividida em duas eras principais: a era pré-

Transformer (dominada por redes recorrentes) e a era pós-Transformer (dominada por mecanismos de atenção).

A arquitetura **Transformer** descartou a recorrência em favor do mecanismo de **self-attention** (autoatenção).

- **Paralelização Total:** O Transformer processa toda a sequência de entrada simultaneamente, não sequencialmente.
- **Mecanismo de Atenção:** Permite que o modelo "olhe" para todas as outras palavras na frase ao mesmo tempo para entender o contexto de uma palavra específica, independentemente da distância entre elas.
- **Positional Encoding:** Como não há recorrência, a ordem das palavras é injetada matematicamente através de vetores de posição.

Após o artigo seminal de 2017, uma nova classe de modelos emergiu: os modelos generativos pré-treinados, conhecidos pela sigla GPT (Generative Pre-trained Transformer). O primeiro GPT, apresentado pela OpenAI em 2018, demonstrou que era possível pré-treinar um modelo de linguagem em grandes volumes de texto não supervisionado e, em seguida, refiná-lo (fine-tunar) para tarefas específicas com relativamente poucos exemplos.

Essa abordagem de pré-treinamento seguido de refinamento tornou-se o paradigma dominante e inaugurou uma era de escalabilidade sem precedentes. Cada geração de modelos trouxe aumentos exponenciais no número de parâmetros e na quantidade de dados de treinamento, resultando em capacidades cada vez mais impressionantes.

2.5.4 Diagrama de Evolução e Arquitetura

O diagrama abaixo ilustra o fluxo completo de um modelo de linguagem N-gramas, desde a fase de treinamento até a fase de inferência (previsão/geração de texto)

O diagrama acima tem duas fases

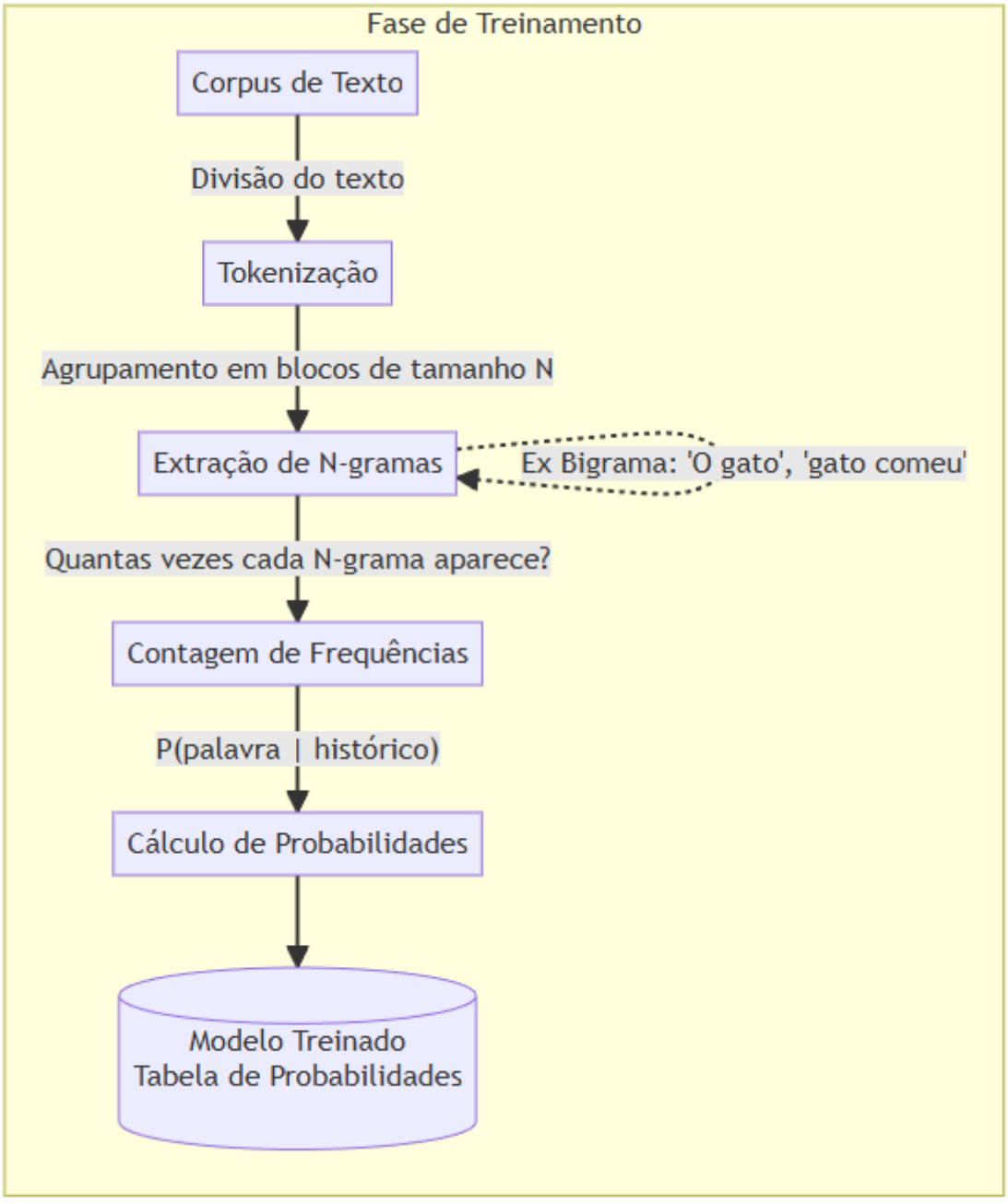


Figura 2.3: N-grams fase treinamento

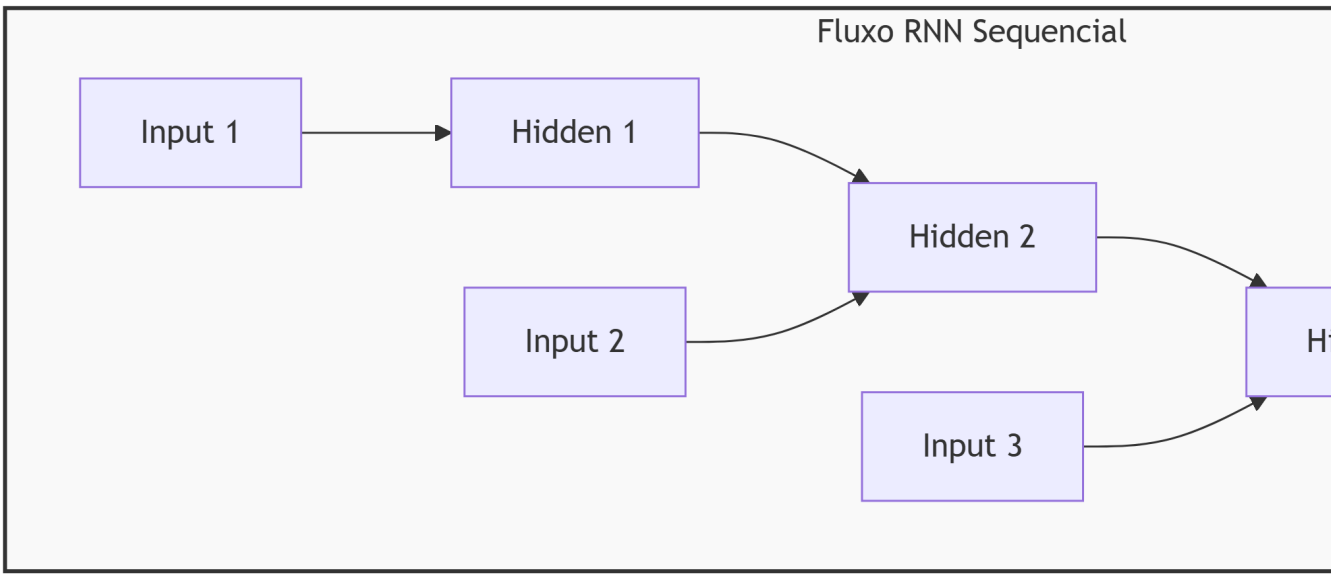
Fase de Treinamento:

- Recebe um volume de textos brutos (Corpus).
- Divide-os em pedaços básicos (Tokens).
- Agrupa esses tokens no tamanho desejado, por exemplo, de 2 em 2 para um modelo Bigrama (Extração de N-gramas).
- Conta as repetições e gera probabilidades estatísticas para saber "qual palavra é mais provável de aparecer depois de X" (Cálculo de Probabilidades).

Fase de Inferência / Geração:

- Para gerar um texto ou fazer o autocompletar, o modelo pega a frase atual (Contexto).
- Olha apenas para as palavras anteriores restritas ao tamanho do N-grama (Extração do Histórico).
- Consulta a sua "Tabela Mágica" (Busca no Modelo).
- Cospe a palavra que estatisticamente faz mais sentido a seguir (Previsão)

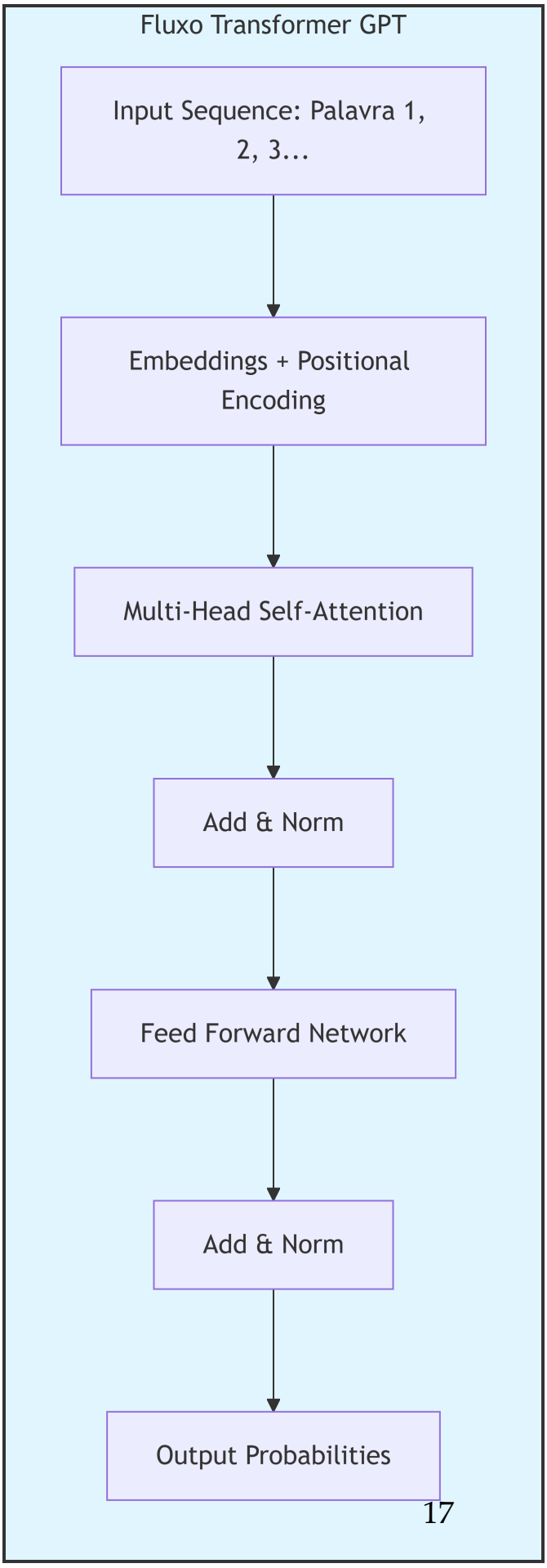
O diagrama abaixo ilustra o processamento linear de uma Rede Neural Recorrente (RNN).



No diagrama acima é possível visualizar o “Gargalo” destas redes.

- O diagrama flui da esquerda para a direita , representando o tempo passando enquanto o modelo lê uma frase, palavra por palavra:
- Entradas (Inputs 1, 2 e 3): Representam os tokens da sua sequência sendo inseridos um por vez. Por exemplo, em uma frase como “O gato dormiu”, Input 1 seria “O”, Input 2 seria “gato”, etc.
- Estados Ocultos (Hidden 1, 2 e 3): Esta é a “memória” de curto prazo da rede. Ela acumula o conhecimento do que já foi lido.
- Quando o Input 1 entra, ele gera o estado Hidden 1.
- Quando o Input 2 entra, observe que há duas setas apontando para Hidden 2: uma vindo do Input 2 e outra vindo do Hidden 1. Isso significa que o modelo processa a palavra atual combinada com a “lembrança” da palavra anterior.
- O Hidden 3 faz o mesmo, dependendo do Input 3 e de toda a memória acumulada lá atrás no passo do Hidden 2.
- Saída (Output): No final da cadeia, o estado oculto final (Hidden 3) possui (em teoria) o contexto encapsulado de toda a sequência lida, e é usado para gerar o resultado final, como prever a próxima palavra.
- O diagrama serve visualmente para provar o gargalo de processamento: a impossibilidade de paralelização.
- Fica evidente graças às setas conectando os sucessivos nós Hidden que você não pode calcular matematicamente o bloco Hidden 3 sem antes ter terminado de calcular o bloco Hidden 2, que precisa do bloco Hidden 1.

- Computadores modernos e Placas de Vídeo (GPUs) são extremamente rápidos e construídos para processar milhares de coisas ao mesmo tempo (“em paralelo”). No entanto, essa técnica quebra esse poder, pois força a máquina a esperar pacientemente o passo anterior terminar (fila indiana).


Fluxo Transformer GPT
Input Sequence: Palavra 1, 2, 3...
Embeddings + Positional Encoding
Multi-Head Self-Attention
Add & Norm
Feed Forward Network
Add & Norm
Output Probabilities

2.6 Tipos de LLMs Baseados em Transformers e Causal Language Modeling (CLM)

Os LLMs modernos geralmente utilizam variações da arquitetura Transformer original:

1. **Encoder-Only (ex: BERT):** Especialistas em "entender" o texto. Ótimos para classificação, reconhecimento de entidades e análise de sentimento. O modelo vê o contexto bidirecional (esquerda e direita).
2. **Decoder-Only (ex: GPT, LLaMA):** Especialistas em "gerar" texto. Treinados para prever a próxima palavra (causal language modeling). O modelo vê apenas o contexto à esquerda.
3. **Encoder-Decoder (ex: T5, BART):** Utilizados para tradução e resumo, onde uma sequência é transformada em outra.

Causal Language Modeling (CLM) é o objetivo de treinamento em que o modelo aprende a **prever o próximo token** de uma sequência, usando apenas os tokens anteriores como contexto.

Em termos simples, dado um texto como:
`Eu gosto de estudar`
o modelo aprende algo como:

- depois de `Eu` ◎ prever `gosto`
- depois de `Eu gosto` ◎ prever `de`
- depois de `Eu gosto de` ◎ prever `estudar`

Ou seja, ele sempre prevê **da esquerda para a direita**.

A palavra **causal** aqui não significa "causa" no sentido filosófico. Significa que o modelo respeita uma **máscara causal**: ao prever o token atual, ele **não pode olhar para o futuro**. Ele só enxerga os tokens anteriores.

Formalmente, para uma sequência (x_1, x_2, …, x_T), o CLM maximiza a probabilidade:

$$P(x_1, x_2, ..., x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t})$$

Isso quer dizer que a probabilidade da sequência inteira é decomposta como uma cadeia de previsões do próximo token.

Durante o treinamento, normalmente usa-se **teacher forcing**: a sequência correta é dada ao modelo, e ele tenta prever cada próximo token. A perda costuma ser **cross-entropy** entre a distribuição prevista e o token verdadeiro seguinte.

Exemplo com deslocamento:

- Entrada: `O gato dorme`
- Alvo: `gato dorme <eos>`

Então o modelo recebe a frase deslocada e aprende a prever o próximo elemento em cada posição.

Esse objetivo é típico de modelos **autoregressivos**, tais como GPT, LLaMA, Mistral, Gemma e semelhantes.

Ele é diferente de objetivos como o **Masked Language Modeling (MLM)**, usado em BERT, onde alguns tokens são mascarados e o modelo tenta recuperá-los usando contexto dos dois lados.

Na prática, o CLM é muito adequado para geração de texto, autocompletar, chatbots, resumo generativo, - tradução autoregressiva, geração de código e outras tarefas de PLN.

A principal intuição é: **o modelo aprende a continuar sequências plausíveis**.

Exemplo em pseudoformato:

```
Sequência:  [Eu] [gosto] [de] [Python]

Passos de treino:
entrada: [Eu]                alvo: [gosto]
entrada: [Eu gosto]          alvo: [de]
entrada: [Eu gosto de]       alvo: [Python]
```

Resumo

Compreendemos que a limitação de processamento sequencial das RNNs levou ao desenvolvimento dos Transformers, que utilizam mecanismos de atenção e processamento paralelo.

Verifique sua compreensão

Se um modelo atribui probabilidades 0,6, 0,3 e 0,1 a três possíveis próximos tokens, isso não significa que ele "sabe" qual é o correto. Significa apenas que, condicionado ao contexto e aos parâmetros atuais, o primeiro token recebeu maior massa de probabilidade. Explique por que uma estratégia de amostragem ainda poderia escolher qualquer um dos três.

2.7 Da aprendizagem de uma função à aprendizagem em contexto

Uma rede neural tradicional aprende uma função durante o treinamento e mantém seus parâmetros fixos quando recebe novos exemplos. Um LLM também funciona assim, mas a janela de contexto acrescenta uma possibilidade interessante: exemplos fornecidos no próprio prompt podem modificar temporariamente o comportamento sem alterar os pesos. Esse fenômeno é chamado de **aprendizagem em contexto** (*in-context le-*

arning).

Considere uma tarefa de classificar comandos como `leitura` ou `escrita`. Podemos fornecer dois pares de exemplo e, em seguida, um novo comando:

```
"mostre o arquivo" -> leitura
"salve o relatório" -> escrita
"liste a pasta" -> ?
```

O modelo usa os exemplos como parte da sequência e tenta continuar o padrão com `leitura`. Com **zero exemplos**, a tarefa é *zero-shot*; com um exemplo, *one-shot*; com alguns exemplos, *few-shot*. Nada foi gravado permanentemente: ao iniciar outro contexto, essa adaptação desaparece.

Essa distinção separa três momentos. No **pré-treinamento**, os parâmetros absorvem regularidades de grandes coleções textuais. No **ajuste fino**, parte ou todos os parâmetros são modificados para uma finalidade. Na **aprendizagem em contexto**, apenas as ativações mudam enquanto o modelo processa o prompt. Confundir esses mecanismos leva a expectativas erradas sobre memória e aprendizado.

> ⚠ Demonstração não é garantia
>
> Um exemplo no prompt orienta a continuação, mas não cria uma regra formal. Mudanças de ordem, formulação ou contexto podem alterar o resultado; por isso, tarefas importantes ainda exigem testes e validação.

Capítulo 3

Configuração do Ambiente de Desenvolvimento

Para trabalhar com LLMs, é necessário um ambiente robusto capaz de lidar com computação numérica intensiva (os tensores). Utilizaremos **Python** e **PyTorch**, o framework padrão da indústria para pesquisa em Deep Learning.

3.1 Pré-requisitos

- **Sistema Operacional:** Linux (recomendado Ubuntu 20.04+) ou Windows com WSL2.
- **Hardware:** GPU NVIDIA recomendada (com drivers CUDA instalados) para treinamento ou inferência eficiente.
- **Python:** Versão 3.8 a 3.11.

3.2 Gerenciamento de Ambientes Virtuais

Nunca instale bibliotecas no ambiente global do Python. Utilize `venv` ou `Conda`.

```
# Criação do ambiente virtual (Linux/Mac)
# python3 -m venv .venv
# Ativação do ambiente
# source .venv/bin/activate

# No Windows (PowerShell):
python -m venv .venv
.\venv\Scripts\Activate.ps1
```

Exemplo simples em Python

```
print("Olá Mundo")
```

3.3 Instalação do PyTorch

A instalação do PyTorch depende da sua versão de CUDA. Visite pytorch.org para o comando exato. Abaixo, um exemplo padrão para CUDA 11.8:

```
# Instalação com suporte a GPU (NVIDIA)
pip install torch torchvision torchaudio
  --index-url https://download.pytorch.org/whl/cu118

# Se você não possui GPU (apenas CPU - muito lento para LLMs):
# pip install torch torchvision torchaudio
```

3.4 Ecossistema Hugging Face

A biblioteca `transformers` da Hugging Face é a interface padrão para acessar modelos pré-treinados.

```
pip install transformers accelerate datasets torch
```

- **transformers:** Carrega modelos e tokenizadores.
- **accelerate:** Otimiza o treinamento e inferência em hardware variado (multi-GPU, mixed precision).
- **datasets:** Facilita o download e processamento de grandes corpora de texto.

3.5 Script de Verificação (Sanity Check)

Crie um arquivo chamado `check_env.py` para validar se o PyTorch consegue acessar a GPU e se a biblioteca Transformers está funcional.

```
import torch
from transformers import pipeline

def check_environment():
    print("=== Verificação de Ambiente ===")

    # 1. Verificar Python e PyTorch
    print(f"PyTorch Version: {torch.__version__}")

    # 2. Verificar disponibilidade de CUDA (GPU)
    cuda_available = torch.cuda.is_available()
    print(f"CUDA Available: {cuda_available}")

    if cuda_available:
        print(f"GPU Device: {torch.cuda.get_device_name(0)}")
        print(f"VRAM: {torch.cuda.get_device_properties(0).tota
            / 1e9:.2f} GB")
    else:
```

```
        print("AVISO: Executando em CPU. O desempenho será limi

    # 3. Teste rápido de inferência com um modelo pequeno (GPT-
    print("\n=== Teste de Inferência (Hugging Face) ===")
    try:
        # Pipeline abstrai a complexidade de tokenização e mode
        generator = pipeline('text-generation', model='gpt2', d
            if cuda_available else -1)

        prompt = "The future of AI is"
        result = generator(prompt, max_length=30, max_new_token
            truncation=True)

        print(f"Prompt: {prompt}")
        print(f"Output: {result[0]['generated_text']}")
        print("\nSUCESSO: Ambiente configurado corretamente.")

    except Exception as e:
        print(f"""\nERRO: Falha ao carregar modelo ou executar
                  \nDetalhes: {e}""")

if __name__ == "__main__":
    check_environment()
```

Aqui estão alguns exemplos simples que ilustram conceitos básicos relacionados ao Transformer usando a biblioteca transformers da Hugging Face:

```
from transformers import AutoTokenizer, AutoModel
import torch

# Carregar tokenizer e modelo pré-treinado
```

```
model_name = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name)

# Texto de exemplo
text = "Transformers são incríveis para PLN!"

# Tokenização
inputs = tokenizer(text, return_tensors="pt")

# Obter embeddings do modelo
outputs = model(**inputs)
last_hidden_states = outputs.last_hidden_state

print(last_hidden_states.shape)  # (batch_size, sequence_length
```

```
last_hidden_states
```

Este código mostra como carregar um modelo Transformer pré-treinado (BERT), tokenizar um texto e obter as representações internas (embeddings) do modelo.

Outro exemplo simples para usar um modelo Transformer para classificação de texto:

```
from transformers import pipeline

# Criar pipeline de classificação de sentimento
classifier = pipeline("sentiment-analysis")

# Classificar texto
result = classifier("Eu adoro aprender sobre Transformers!")
print(result)
```

Este exemplo usa um pipeline pronto para análise de sentimento, que internamente utiliza um modelo Transformer para classificar o texto.

Resumo

Configuramos um ambiente Python isolado, instalamos o PyTorch com suporte a CUDA e validamos a instalação executando um modelo GPT-2 básico.

3.6 Como tornar o experimento reproduzível

Um ambiente funcionar hoje não garante que funcionará após uma atualização. Registre a versão do Python, do PyTorch, do driver e das bibliotecas instaladas. Em projetos acadêmicos, salve as dependências em `requirements.txt` ou `pyproject.toml` e anote a semente aleatória usada nos experimentos. A semente não elimina todas as fontes de não determinismo em GPU, mas reduz variações difíceis de investigar.

Antes de treinar, estime a memória. Os parâmetros são apenas uma parte do consumo: ativações, gradientes e estados do otimizador também ocupam espaço. Um modelo em `float32` usa quatro bytes por parâmetro somente para os pesos; durante o treinamento com AdamW, o total pode ser várias vezes maior. Começar com lote e sequência pequenos evita confundir falta de memória com erro no código.

Checklist de diagnóstico

Execute primeiro uma operação pequena na CPU, depois na GPU, confirme `tensor.device` e inspecione os formatos com `tensor.shape`. Se ocorrer *out of memory*, reduza `batch_size` ou `seq_len` antes de alterar a arquitetura.

3.7 Entendendo o caminho até a GPU

Uma GPU acelera o treinamento porque executa muitas operações numéricas semelhantes em paralelo. Entretanto, o ganho só aparece quando há trabalho suficiente para ocupar seus núcleos e quando a movimentação de dados não domina o tempo total. Um programa pode possuir GPU disponível e ainda ser lento por causa de lotes muito pequenos, carregamento de dados bloqueante ou cópias frequentes entre CPU e dispositivo.

Três recursos disputam espaço na memória: pesos, estados necessários à retropropagação e dados do lote. Precisão reduzida, acumulação de gradientes e *gradient checkpointing* atacam partes diferentes desse consumo. A precisão reduzida diminui bytes por número; a acumulação simula um lote maior com vários microlotes; o checkpointing economiza ativações e as recalcula durante o backward.

```
import torch

device = torch.device("cuda" if torch.cuda.is_available() else
x = torch.randn(32, 128, 512, device=device)
print("dispositivo:", x.device)
print("memória do tensor (MiB):", x.nelement() * x.element_size
```

Ao medir desempenho, sincronize a GPU antes e depois do trecho cronometrado, pois as operações CUDA são assíncronas. Faça algumas iterações de aquecimento e compare o mesmo cálculo; medir apenas a primeira execução inclui inicialização e compilação de kernels.

> **Diagnóstico orientado por evidências**
>
> Se a GPU estiver pouco utilizada, investigue alimentação de dados e tamanho das operações. Se estiver sem memória, investigue formatos, ativações e lote. Se estiver totalmente utilizada, otimizações de

> kernel e precisão podem ser mais relevantes.

Capítulo 4

Fundamentos Matemáticos

4.1 A linguagem como problema matemático

Para que uma máquina possa processar linguagem, é necessário traduzir texto em representações matemáticas. A linguagem humana, com sua riqueza de nuances, ambiguidades e estruturas complexas, não é diretamente interpretável por algoritmos matemáticos. portanto, uma das primeiras tarefas na construção de um LLM é criar representações numéricas adequadas.

O texto é transformado em números através de um processo chamado tokenização, onde palavras, subpalavras ou caracteres são mapeados para inteiros. Esses inteiros são então convertidos em vetores densos de números reais através de camadas de embedding. Cada token recebe um vetor associado que captura certas propriedades semânticas e sintáticas desse token.

4.2 Espaços Vetoriais e Embeddings

Um conceito central nos LLMs é o de espaço vetorial. Cada palavra, após o processamento pelo modelo, existe como um ponto — ou mais precisamente, como um vetor — em um espaço de alta dimensionalidade. A

dimensionalidade desses espaços varia, mas comumente está na ordem de centenas ou milhares de dimensões. A dimensão refere-se ao número de valores independentes que definem cada vetor.

A magia dos embeddings reside no fato de que relações semânticas entre palavras são refletidas como relações geométricas entre seus vetores. Se você pegar o vetor da palavra "rei", subtrair o vetor de "homem" e adicionar o vetor de "mulher", obterá um vetor muito próximo do vetor de "rainha". Essa propriedade, conhecida como analogia aritmética de word2vec, demonstra que os embeddings capturam relações semânticas de forma quantificável.

4.3 Probabilidade e Previsão

No coração de todo modelo de linguagem está o princípio da probabilidade condicional. Dado uma sequência de tokens anteriores, o modelo calcula a probabilidade de cada token possível do vocabulário ser o próximo. Matematicamente, se denotarmos uma sequência de tokens como $x_1, x_2, \ldots, x_n$, o modelo calcula $P(x_{n+1} \mid x_1, x_2, \ldots, x_n)$.

Essa distribuição de probabilidade é calculada através de múltiplas camadas de transformação neural, cada uma processando a representação intermediária da sequência e refinando-a progressivamente. Durante o treinamento, o modelo ajusta seus parâmetros para maximizar a probabilidade dos tokens reais que aparecem no texto de treinamento.

A saída final do modelo é tipicamente uma distribuição de probabilidade sobre todo o vocabulário. Durante a geração de texto, diferentes estratégias de amostragem podem ser usadas para selecionar o próximo token, desde a escolha determinística do token mais provável até métodos estocásticos que introduzemvariabilidade criativa.

4.4 Funções de Ativação e Normalização

As redes neurais que compõem os LLMs utilizam diversas funções matemáticas para transformar dados entre camadas. As funções de ativação, como ReLU (Rectified Linear Unit), GELU (Gaussian Error Linear Unit) e SwiGLU, introduzem não-linearidades que permitem ao modelo aprender relações complexas entre entrada e saída.

A normalização é outro componente crucial. Técnicas como Layer Normalization e RMS Normalization estabilizam o treinamento ao garantir que os valores intermediários mantenham magnitudes adequadas, evitando problemas numéricos que poderiam dificultar ou impossibilitar o aprendizado.

4.5 Dimensões: a ferramenta de depuração mais útil

Considere um lote $X \in \mathbb{R}^{B\times T\times d}$, em que B é o tamanho do lote, T é o número de tokens e d é a dimensão do modelo. Uma projeção linear com $W \in \mathbb{R}^{d\times d_k}$ produz $XW \in \mathbb{R}^{B\times T\times d_k}$. Acompanhar essas dimensões permite detectar uma grande classe de erros antes mesmo de executar o programa.

O produto escalar mede compatibilidade: $q \cdot k = \sum_i q_i k_i$. Na atenção, produtos maiores indicam que uma consulta e uma chave estão mais alinhadas. Já a *softmax* transforma um vetor de escores z em probabilidades $p_i = e^{z_i} / \sum_j e^{z_j}$. Subtrair $\max(z)$ antes da exponenciação preserva o resultado e melhora a estabilidade numérica.

Exemplo rápido

Para os logits $[2, 1, 0]$, a softmax atribui mais probabilidade ao primeiro item, mas mantém probabilidade positiva para os demais. Somar a mesma constante a todos os logits não altera essa distribuição. Verifique essa propriedade algebricamente ou com `torch.softmax`.

4.6 Regra da cadeia e retropropagação

Uma rede é uma composição de funções. Se $z = Wx + b$, $a = f(z)$ e $L = g(a)$, a influência de W sobre a perda é obtida pela regra da cadeia:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial W}.$$

A retropropagação percorre o grafo computacional no sentido inverso, reutilizando derivadas intermediárias. Bibliotecas de diferenciação automática registram as operações do forward e calculam produtos vetor-Jacobiano; elas não "descobrem" a matemática, mas automatizam sua aplicação.

```
import torch

w = torch.tensor(2.0, requires_grad=True)
x = torch.tensor(3.0)
y = w * x
loss = (y - 5) ** 2
loss.backward()
print(w.grad)  # 2 * (w*x - 5) * x = 6
```

O sinal do gradiente indica a direção de crescimento local da perda; o otimizador caminha na direção oposta. Gradiente zero pode representar

um mínimo, um ponto plano, saturação de uma ativação ou um caminho desconectado do grafo. Por isso, inspecionar apenas a perda não basta.

4.7 Entropia cruzada e divergência entre distribuições

Para uma distribuição verdadeira p e uma aproximação q, a entropia cruzada é $H(p, q) = -\sum_i p_i \log q_i$. Quando o alvo é um único token, somente o logaritmo da probabilidade desse token contribui. A divergência de Kullback–Leibler satisfaz $D_{KL}(p\|q) = H(p, q) - H(p)$; se p é fixo, minimizar entropia cruzada também minimiza essa divergência.

> ⚠ KL não é distância
>
> $D_{KL}(p\|q)$ não é simétrica e não obedece, em geral, à desigualdade triangular. Ela quantifica informação perdida ao usar q no lugar de p, não uma distância geométrica comum.

Capítulo 5

Tokenização e Vocabulários

Objetivo

Este capítulo detalha a ponte fundamental entre a linguagem humana e a representação numérica compreendida por redes neurais. Exploraremos a teoria da tokenização *subword* (subpalavra), implementaremos o algoritmo **Byte-Pair Encoding (BPE)** do zero e discutiremos a arquitetura de preparação de dados para treinamento de Large Language Models (LLMs).

5.1 Por Que Tokenizar é Necessário

Redes neurais não processam strings; elas processam tensores de números (ponto flutuante ou inteiros). O processo de converter texto em números é chamado de **Tokenização**.

Modelos de linguagem não processam texto diretamente como strings de caracteres. Eles operam em unidades discretas chamadas tokens. A tokenização é o processo de converter texto em uma sequência de tokens que o modelo pode processar matematicamente.

A granularidade da tokenização varia entre abordagens. Em um extremo, temos tokenização por caractere, onde cada caractere é um token. No outro extremo, temos tokenização por palavra, onde cada palavra é um token. A maioria dos LLMs modernos utiliza uma abordagem intermediária chamada tokenização por subpalavras, que divide palavras incomuns em unidades menores enquanto mantém palavras frequentes como tokens únicos.

5.2 Visão Geral do Byte Pair Encoding (BPE)

O algoritmo mais amplamente utilizado para tokenização em LLMs é o Byte Pair Encoding (BPE), ou alguma variante dele. Originalmente desenvolvido para compressão de dados, o BPE foi adaptado para tokenização de linguagem natural e oferece um equilíbrio excelente entre granularidade e tamanho de vocabulário.

O BPE funciona começando com um vocabulário de caracteres individuais e iterativamente mesclando o par de tokens mais frequente para criar novos tokens. O processo continua até que o vocabulário atinja um tamanho desejado. O resultado é um vocabulário que contém tanto caracteres quanto sequências de caracteres de tamanhos variados, das palavras inteiras mais frequentes às subpalavras menos frequentes.

Por exemplo, a palavra “desproporcionalidade” seria tokenizada em unidades menores como [“des”, “pro”, “por”, “cional”, “idade”], pois é improvável que apareça frequentemente como uma palavra única. Já a palavra “o” seria um único token, pois aparece com altíssima frequência.

5.3 Vocabulários Típicos e Seus Tamanhos

Os vocabulários utilizados em LLMs modernos variam em tamanho, tipicamente entre 30.000 e 200.000 tokens únicos. Vocabulários maiores

capturam mais nuances, mas exigem mais memória e computação. Vocabulários menores são mais eficientes, mas podem forçar a divisão de muitas palavras em múltiplos tokens, aumentando a profundidade da sequência de entrada.

A tokenização também participa do custo de inferência. Processar texto para produzir tokens e, no sentido inverso, converter tokens novamente em texto são operações realizadas em cada interação com o modelo.

5.4 O Impacto da Tokenização na Geração

A forma como o texto é tokenizado tem implicações profundas para o comportamento do modelo. Diferentes idiomas tokenizam de formas radicalmente diferentes: idiomas como o inglês tokenizam relativamente bem em BPE, enquanto idiomas como o chinês ou idiomas com sistemas de escrita complexos podem requerer tokenizadores especializados para atingir eficiência comparável.

O número de tokens em um texto também impacta diretamente o custo computacional de processá-lo. Muitos serviços de API que utilizam LLMs cobram com base no número de tokens processados, refletindo o custo computacional real da tokenização e do processamento subsequente.

5.4.1 Dilema da Granularidade

Historicamente, existiam duas abordagens principais, ambas com limitações severas para modelos modernos:

1. **Tokenização por Palavra (Word-level):**

    - Divide o texto por espaços.

- *Problema:* O vocabulário se torna gigantesco (milhões de palavras possíveis). Palavras não vistas durante o treino tornam-se tokens `<UNK>` (Unknown), perdendo informação semântica.

2. **Tokenização por Caractere (Character-level):**

- Divide o texto em letras individuais.
- *Problema:* O vocabulário é pequeno, mas as sequências tornam-se extremamente longas, dificultando a modelagem de dependências de longo prazo e aumentando o custo computacional.

5.4.2 A Solução: Tokenização Subword (Subpalavra)

A abordagem moderna (utilizada pelo GPT-4, Llama 3, BERT) é a tokenização *subword*. Ela baseia-se no princípio de que palavras frequentes não devem ser divididas, mas palavras raras devem ser decompostas em unidades menores e significativas.

- **Exemplo:** A palavra "tokenização" pode ser dividida em `["token", "iza", "ção"]`.
- **Benefício:** Permite um vocabulário de tamanho fixo (ex: 32k a 100k tokens) com capacidade de representar virtualmente qualquer palavra através da composição de subunidades.

5.5 Treinamento do Byte Pair Encoding (BPE)

O BPE é um algoritmo iterativo que constrói um vocabulário fundindo os pares de caracteres (ou bytes) mais frequentes no corpus de treinamento.

5.5.1 2.1 Fluxo de Treinamento do BPE

Abaixo, o diagrama ilustra o ciclo de vida do treinamento de um tokenizador BPE:

```
flowchart TD
    A[Corpus de Texto Bruto] --> B[Pré-tokenização]
    B -->|Divisão por espaços/regras| C[Vocabulário Inicial]
    C -->|Caracteres individuais| D{Loop de Treinamento}

    D --> E[Contar Frequência de Pares Adjacentes]
    E --> F[Identificar Par Mais Frequente]
    F --> G[Criar Nova Regra de Fusão 'Merge']
    G --> H[Atualizar Vocabulário e Corpus]
    H --> I{Atingiu Tamanho Limite?}

    I -- Não --> D
    I -- Sim --> J[Vocabulário Final & Regras de Merge]

    style D stroke:#f66,stroke-width:2px
    style J stroke:#0f0,stroke-width:2px
```

5.5.2 2.2 Exemplo de Execução (Trace)

Suponha o corpus: `("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4)`

1. **Base:** `h, u, g, p, n, b`
2. **Iteração 1:** O par `u, n` aparece em "pun" e "bun" (12 + 4 = 16 vezes). É o mais frequente.
    - Novo token: `un`.
    - Vocabulário: `h, u, g, p, n, b, un`.

3. **Iteração 2:** O par `u, g` aparece em "hug" e "pug" (10 + 5 = 15 vezes).
    - Novo token: `ug`.
    - Vocabulário: `h, u, g, p, n, b, un, ug`.

5.6 3. Implementação Prática: BPE do Zero em Python

A seguir, implementamos um BPE simplificado para demonstrar a lógica interna de manipulação de strings e contagem estatística.

```
import collections
import re

class SimpleBPE:
    def __init__(self, vocab_size=300):
        self.vocab_size = vocab_size
        self.merges = {} # Armazena as regras de fusão
        self.vocab = {}  # Mapeamento token -> ID

    def get_stats(self, vocab_counts):
        """Conta a frequência de todos os pares adjacentes."""
        pairs = collections.defaultdict(int)
        for word, freq in vocab_counts.items():
            symbols = word.split()
            for i in range(len(symbols) - 1):
                pairs[symbols[i], symbols[i+1]] += freq
        return pairs

    def merge_vocab(self, pair, v_in):
```

```
        """Aplica a fusão do par mais frequente no vocabulário
        v_out = {}
        bigram = re.escape(' '.join(pair))
        p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')

        for word in v_in:
            # Substitui 'a b' por 'ab'
            w_out = p.sub(''.join(pair), word)
            v_out[w_out] = v_in[word]
        return v_out

    def train(self, text):
        # 1. Pré-tokenização simples (dividir por espaços) e co
        word_freqs = collections.defaultdict(int)
        for word in text.split():
            # Adiciona espaços entre chars para representar div
            # Ex: "hello" -> "h e l l o"
            word_freqs[" ".join(list(word))] += 1

        num_merges = self.vocab_size - len(set("".join(text.spl

        print(f"Iniciando treinamento BPE para {num_merges} mer

        # 2. Loop de Treinamento
        for i in range(num_merges):
            pairs = self.get_stats(word_freqs)
            if not pairs:
                break

            # Encontra o par mais frequente
            best = max(pairs, key=pairs.get)
```

```
            self.merges[best] = "".join(best)

            # Atualiza o vocabulário aplicando o merge
            word_freqs = self.merge_vocab(best, word_freqs)

            if i % 10 == 0:
                print(f"Iteração {i}: Merge {best} -> {''.join(

        # Construir vocabulário final (Token -> ID)
        # Na prática, incluiríamos tokens especiais aqui
        unique_tokens = set()
        for word in word_freqs:
            unique_tokens.update(word.split())

        self.vocab = {token: i for i, token in enumerate(sorted
        print(f"Treinamento concluído. Tamanho do vocabulário:

    def encode(self, text):
        """Tokeniza um novo texto usando as regras aprendidas."
        # Esta é uma versão simplificada. Um encoder real aplic
        # as regras de merge na mesma ordem em que foram aprend
        tokens = []
        for word in text.split():
            word_spaced = " ".join(list(word))

            # Aplica merges iterativamente (ineficiente, apenas
            # Em produção, usa-se uma árvore ou hash map otimiz
            for pair, merged in self.merges.items():
                bigram = " ".join(pair)
                if bigram in word_spaced:
                    word_spaced = word_spaced.replace(bigram, m
```

```
            word_tokens = word_spaced.split()
            ids = [self.vocab.get(t, self.vocab.get("<UNK>", 0)
            tokens.extend(ids)
        return tokens

# --- Exemplo de Uso ---
corpus = "low low low low low lower lower newest newest newest
tokenizer = SimpleBPE(vocab_size=20) # Vocabulário pequeno para
tokenizer.train(corpus)

print("\nVocabulário:", tokenizer.vocab)
print("Encode 'lower newest':", tokenizer.encode("lower newest"
```

5.7 4. Preparação do Dataset de Treinamento

Ter um tokenizador é apenas o primeiro passo. Para treinar um LLM, os dados devem ser estruturados em tensores com tokens especiais e janelas de contexto.

5.7.1 4.1 Tokens Especiais

Um vocabulário robusto deve incluir tokens de controle que não aparecem no texto natural:

Token	Significado	Função
[PAD]	Padding	Preenche sequências curtas para manter o tamanho fixo do lote (batch). Geralmente ID 0.
[UNK]	Unknown	Representa caracteres ou subpalavras que o modelo nunca viu.
[BOS]	Beginning of Sentence	Sinaliza o início de uma sequência (crucial para geração).
[EOS]	End of Sentence	Sinaliza onde a geração deve parar.
[MASK]	Mask	Usado em modelos tipo BERT para preenchimento de lacunas.

5.7.2 4.2 Pipeline de Transformação: De Texto a Tensor

O processo final envolve normalização, tokenização e estruturação em janelas deslizantes (Sliding Windows) ou empacotamento.

```
sequenceDiagram
    participant Raw as Texto Bruto
    participant Norm as Normalizador
    participant Tok as Tokenizador
    participant Post as Pós-Processador
    participant Tensor as Tensor Final
```

```
    Raw->>Norm: "Olá, MUNDO!"
    Note right of Norm: Unicode NFKC, Lowercase (opcional)
    Norm->>Tok: "olá, mundo!"
    Note right of Tok: Aplica BPE
    Tok->>Post: [104, 2201, 15, 889, 2]
    Note right of Post: Adiciona [BOS] e [EOS]
    Post->>Tensor: [1, 104, 2201, 15, 889, 2, 3]
    Note right of Tensor: Padding até Context Length (ex: 10)
    Tensor->>Tensor: [1, 104, 2201, 15, 889, 2, 3, 0, 0, 0]
```

5.7.3 4.3 Estratégias de Contexto (Context Window)

Ao preparar o dataset, o texto contínuo deve ser fatiado para caber na janela de contexto do modelo (ex: 4096 tokens).

1. **Truncation:** Cortar o texto se exceder o limite.
2. **Sliding Window:** Criar exemplos sobrepostos.
    - Exemplo: Texto `[A, B, C, D, E]`, Janela 3.
    - Amostra 1: `[A, B, C]`
    - Amostra 2: `[B, C, D]` (Stride=1)
3. **Packing:** Concatenar múltiplos textos curtos em uma única sequência, separados por `[EOS]`, para maximizar a eficiência da GPU (evitando excesso de `[PAD]`).

5.8 5. Considerações de Arquitetura

Ao projetar este pipeline em produção:

- **Normalização Unicode:** Utilize a forma **NFKC** para garantir que caracteres visualmente idênticos tenham a mesma representação de bytes.

- **Byte-Level BPE:** Modelos como GPT-2/3/4 operam em nível de bytes UTF-8, não caracteres Unicode. Isso elimina o token `<UNK>`, pois qualquer string pode ser decomposta em bytes.
- **Eficiência:** A implementação Python acima é educativa. Em produção, utilize bibliotecas escritas em **Rust** (como Hugging Face `tokenizers`) que paralelizam a contagem de pares e a aplicação de merges.

5.9 A tokenização muda o problema computacional

Dois textos com o mesmo número de caracteres podem produzir quantidades muito diferentes de tokens. Código, palavras raras, emojis e idiomas pouco representados no corpus de treinamento tendem a ser fragmentados de maneira distinta. Isso afeta custo, latência e quanto conteúdo cabe na janela de contexto.

Considere `transformador`, que poderia ser um token frequente ou ser dividido em partes como `transform` e `ador`. A segunda representação permite reutilizar subpalavras em termos nunca vistos, mas aumenta T, o comprimento da sequência. Como a atenção densa constrói uma matriz $T \times T$, pequenas mudanças na fragmentação podem ter efeito quadrático nessa parte do custo.

> **Experimento**
>
> Tokenize a mesma frase em português e em inglês com um tokenizer real. Compare os identificadores, o número de tokens e a reconstrução com `decode`. Nunca assuma que espaços ou caracteres acentuados correspondem um a um aos tokens.

5.10 O contrato entre tokenizador e modelo

O tokenizador faz parte do modelo tanto quanto as matrizes de pesos. Cada ID seleciona uma linha específica da tabela de embeddings; trocar o vocabulário sem adaptar essa tabela muda o significado de todos os índices. Por isso, checkpoint, configuração e arquivos do tokenizador devem ser versionados juntos.

Tokens especiais também precisam de semântica consistente. `<bos>` pode marcar início, `<eos>` sinalizar término, `<pad>` completar lotes e tokens de função separar mensagens de sistema, usuário e assistente. Se o template de conversa coloca esses marcadores em ordem diferente daquela vista no treinamento, a qualidade pode cair mesmo que o texto visível pareça correto.

5.10.1 Um exemplo de preparação causal

Para IDs `[11, 27, 42, 9]`, o treino autoregressivo pode usar:

```
entrada: [11, 27, 42]
alvo:    [27, 42,  9]
```

Em um lote com padding, as posições artificiais não devem contribuir para a perda. Uma máscara de atenção controla quais tokens podem ser consultados; um `ignore_index` na função de perda controla quais alvos serão avaliados. São responsabilidades diferentes.

5.10.2 Robustez no nível de bytes

Tokenizadores baseados em bytes conseguem representar qualquer sequência UTF-8 sem depender de um token desconhecido. A vantagem é cobertura universal; a desvantagem é que textos com padrões raros po-

dem se fragmentar em muitas unidades. Essa eficiência desigual deve ser medida no idioma e no domínio da aplicação.

! Teste de ida e volta

Para entradas válidas, verifique se `decode(encode(texto))` reconstrói o texto conforme as regras de normalização adotadas. Faça testes explícitos com acentos, emojis, quebras de linha, código e espaços consecutivos.

Capítulo 6

Representação Vetorial: Embeddings e Positional Encoding

Em modelos de Processamento de Linguagem Natural (NLP) baseados em Deep Learning, especificamente na arquitetura Transformer, os dados brutos (texto) não podem ser processados diretamente. O primeiro passo crítico é a transformação de *tokens* discretos (índices inteiros de um vocabulário) em representações vetoriais contínuas e densas, enriquecidas com informações sobre a ordem da sequência.

Este capítulo detalha a arquitetura da camada de entrada, composta por dois subcomponentes principais: **Input Embeddings** e **Positional Encoding**.

6.1 1. Input Embeddings (Camada de Embeddings)

A camada de Embeddings atua como uma tabela de consulta (*lookup table*) aprendível. Enquanto a representação *One-Hot Encoding* gera veto-

res esparsos e de altíssima dimensionalidade (tamanho do vocabulário), os Embeddings projetam esses tokens em um espaço vetorial denso de dimensão inferior (d_{model}), onde a proximidade geométrica reflete a similaridade semântica.

6.1.1 Características Técnicas

- **Mapeamento:** Cada ID de token x é mapeado para um vetor $v \in \mathbb{R}^{d_{model}}$.
- **Dimensionalidade (d_{model}):** Um hiperparâmetro da arquitetura (ex: 512 no Transformer original, 4096 no GPT-3).
- **Escalonamento de Variância:** No artigo original *"Attention Is All You Need"*, os pesos dos embeddings são multiplicados por $\sqrt{d_{model}}$. Isso é feito para contrabalancear a magnitude do produto escalar na camada de atenção subsequente, auxiliando na estabilidade dos gradientes durante o treinamento.

6.2 2. Positional Encoding (Codificação Posicional)

Diferente de Redes Neurais Recorrentes (RNNs) ou LSTMs, a arquitetura Transformer não processa dados sequencialmente; ela processa a sequência inteira em paralelo. Consequentemente, o mecanismo de *Self-Attention* é **invariante à permutação**. Sem uma injeção explícita de posição, o modelo veria as frases "O cão mordeu o homem" e "O homem mordeu o cão" como idênticas em termos de composição de tokens.

Para resolver isso, injetamos um vetor de **Positional Encoding (PE)** somando-o ao vetor de Embedding.

6.2.1 Implementação Matemática

A codificação posicional utiliza frequências de ondas senoidais e cossenos de diferentes comprimentos de onda. Para uma posição pos na sequência e uma dimensão i dentro do vetor de embedding:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

Por que esta fórmula? 1. **Valores Determinísticos:** Não requer parâmetros aprendíveis extras (embora embeddings posicionais aprendíveis também sejam usados em modelos modernos como BERT). 2. **Relatividade Linear:** Para qualquer deslocamento fixo k, PE_{pos+k} pode ser representado como uma função linear de PE_{pos}. Isso facilita para o modelo aprender a atender posições relativas (ex: o token anterior ou o próximo). 3. **Extrapolação:** Permite que o modelo processe sequências mais longas do que as vistas durante o treinamento.

6.3 3. Arquitetura do Fluxo de Dados

A combinação dessas duas camadas resulta na entrada final para os blocos do Transformer. A operação é uma soma elemento a elemento (element-wise addition), seguida geralmente por uma camada de Dropout para regularização.

6.3.1 Diagrama de Fluxo

```
graph TD
    subgraph "Pré-processamento"
    RawText[Texto Bruto] --> Tokenizer[Tokenizador]
```

```
    Tokenizer --> TokenIDs[IDs dos Tokens (Inteiros)]
    end

    subgraph "Camada de Representação Vetorial"
    TokenIDs --> EmbedLayer[Embedding Lookup Table]
    EmbedLayer --> Scale[Escalar por sqrt(d_model)]

    PosIndex[Índices de Posição 0..N] --> PosEncCalc[Cálculo Se
    PosEncCalc --> PosVector[Vetor Positional Encoding]

    Scale --> Sum((Soma Element-wise))
    PosVector --> Sum

    Sum --> Dropout[Dropout Layer]
    end

    Dropout --> TransformerBlock[Bloco Transformer]

    style Sum fill:#f9f,stroke:#333,stroke-width:2px
    style EmbedLayer fill:#bbf,stroke:#333,stroke-width:2px
    style PosEncCalc fill:#bbf,stroke:#333,stroke-width:2px
```

6.4 4. Implementação de Referência (PyTorch)

Abaixo apresentamos uma implementação robusta e anotada, seguindo as especificações padrão da indústria.

```
import torch
import torch.nn as nn
import math
```

```python
class InputEmbeddings(nn.Module):
    def __init__(self, d_model: int, vocab_size: int):
        """
        Args:
            d_model (int): Dimensão do vetor de embedding.
            vocab_size (int): Tamanho do vocabulário.
        """
        super().__init__()
        self.d_model = d_model
        self.vocab_size = vocab_size
        # Camada de Embedding padrão do PyTorch
        self.embedding = nn.Embedding(vocab_size, d_model)

    def forward(self, x):
        # Escalonamento por sqrt(d_model) conforme paper origin
        return self.embedding(x) * math.sqrt(self.d_model)

class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, seq_len: int, dropout: flo
        """
        Args:
            d_model (int): Dimensão do modelo.
            seq_len (int): Comprimento máximo da sequência.
            dropout (float): Taxa de dropout.
        """
        super().__init__()
        self.dropout = nn.Dropout(dropout)

        # Cria uma matriz de (seq_len, d_model) com zeros
        pe = torch.zeros(seq_len, d_model)
```

```
        # Cria um vetor de posições (0, 1, ... seq_len-1)
        # Shape: (seq_len, 1)
        position = torch.arange(0, seq_len, dtype=torch.float).

        # Termo divisor para as frequências (10000^(2i/d_model)
        # Implementado em log-space para estabilidade numérica
        div_term = torch.exp(torch.arange(0, d_model, 2).float(

        # Aplica Seno aos índices pares (2i)
        pe[:, 0::2] = torch.sin(position * div_term)

        # Aplica Cosseno aos índices ímpares (2i+1)
        pe[:, 1::2] = torch.cos(position * div_term)

        # Adiciona dimensão de batch para facilitar o broadcast
        pe = pe.unsqueeze(0)

        # Registra como buffer (não é um parâmetro aprendível,
        self.register_buffer('pe', pe)

    def forward(self, x):
        """
        Args:
            x: Embeddings de entrada. Shape: (Batch_Size, Seq_L
        """
        # Soma o embedding com o positional encoding (até o com
        # x.requires_grad_(False) não é necessário pois pe é bu
        x = x + self.pe[:, :x.shape[1], :]
        return self.dropout(x)

# Exemplo de uso integrado
```

```
class TransformerInputLayer(nn.Module):
    def __init__(self, vocab_size, d_model, max_len, dropout=0.
        super().__init__()
        self.embeddings = InputEmbeddings(d_model, vocab_size)
        self.positional_encoding = PositionalEncoding(d_model,

    def forward(self, x):
        x = self.embeddings(x)
        x = self.positional_encoding(x)
        return x
```

6.4.1 Análise do Código

1. **Estabilidade Numérica:** No cálculo do `div_term`, utilizamos `torch.exp` e `math.log`. Matematicamente, $e^{\ln(x)} = x$. Isso evita potências diretas de números muito grandes ou muito pequenos, mantendo a precisão em ponto flutuante.
2. **Buffers:** O uso de `register_buffer` garante que a matriz de Positional Encoding seja salva junto com o modelo (`state_dict`), mas não seja atualizada pelo otimizador (Backpropagation), pois é fixa.
3. **Broadcasting:** A forma do tensor `pe` é `(1, seq_len, d_model)`. Isso permite que ele seja somado automaticamente a um batch de entradas `(Batch, seq_len, d_model)` sem necessidade de duplicação de memória.

6.5 5. Embedding não é significado isolado

A linha da tabela de embeddings fornece apenas o ponto de partida de um token. Depois de cada bloco, sua representação se torna **contextual**. Assim, ocorrências de “banco” em “banco de dados” e “banco da praça”

começam com o mesmo vetor, mas terminam com estados ocultos diferentes por causa dos tokens vizinhos.

Se os IDs têm formato (B, T), `nn.Embedding` produz $(B,T,d_{model}). A codificação posicional deve ser compatível com as duas últimas dimensões; a soma não concatena posição e conteúdo, portanto o formato permanece inalterado. Esse é um bom ponto para usar`assert x.shape == (B,T,d_model)‘ durante a implementação.

Além das senoides, modelos atuais podem usar posições aprendidas, RoPE ou vieses relativos. O critério não é apenas representar ordem: é também generalizar para comprimentos diferentes e permitir que a atenção expresse distâncias relevantes.

> **Exercício**
>
> Troque dois tokens de uma sequência e compare a soma `embedding + posição`. Depois, remova a codificação posicional e explique por que a autoatenção deixa de distinguir as duas ordens.

6.6 Posições relativas com RoPE

Embeddings posicionais absolutos somam um vetor que representa o índice. A codificação posicional rotativa, conhecida como **RoPE**, segue outra ideia: pares de coordenadas de consultas e chaves são girados por ângulos dependentes da posição. Como o produto escalar entre vetores girados depende da diferença entre os ângulos, a atenção recebe informação de distância relativa.

Para um par (x_{2i}, x_{2i+1}) e ângulo θ, a rotação é

$$\begin{bmatrix} x'_{2i} \\ x'_{2i+1} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_{2i} \\ x_{2i+1} \end{bmatrix}.$$

Frequências diferentes são usadas ao longo das dimensões: algumas

variam rapidamente e distinguem posições próximas; outras variam lentamente e preservam relações em escalas maiores. A rotação é aplicada a Q e K, não necessariamente a V, porque sua finalidade é alterar o cálculo de compatibilidade.

RoPE não torna o contexto ilimitado. Comprimentos muito além dos observados no treinamento podem produzir padrões angulares pouco familiares. Técnicas de escalonamento ajustam frequências ou posições, mas precisam ser avaliadas quanto a recuperação de informação, estabilidade e perda de resolução local.

i Conteúdo e posição continuam separados conceitualmente

O embedding representa o token; a transformação posicional modifica como tokens em lugares diferentes se relacionam. Embora sejam combinados no cálculo, essas funções ajudam a depurar o modelo separadamente.

6.7 Visualizando embeddings com cuidado

PCA, t-SNE e UMAP projetam vetores de alta dimensão para duas ou três dimensões. Essas figuras ajudam a formular hipóteses sobre agrupamentos, mas inevitavelmente distorcem relações. t-SNE enfatiza vizinhanças locais; UMAP tenta preservar mais estrutura global; PCA é linear e mais fácil de interpretar. Nenhum gráfico bidimensional prova, sozinho, que o modelo aprendeu um conceito.

Capítulo 7

O Mecanismo de Atenção (Self-Attention)

O mecanismo de **Self-Attention** (Auto-Atenção) é o componente fundamental que distingue a arquitetura Transformer das redes recorrentes (RNNs) e convolucionais (CNNs) anteriores. Enquanto as RNNs processam sequências passo a passo (o que dificulta a paralelização e a retenção de dependências de longo prazo), o Self-Attention permite que o modelo relacione cada palavra na sequência de entrada com todas as outras palavras simultaneamente.

Este capítulo disseca a **Scaled Dot-Product Attention**, a fórmula matemática que rege essa operação, e implementa o mascaramento necessário para modelos de linguagem generativos (como a família GPT).

7.1 1. Conceitos Fundamentais: Query, Key e Value

Para calcular a atenção, projetamos a entrada (embeddings) em três vetores distintos para cada token. Esta abordagem é inspirada em sistemas de recuperação de informação:

1. **Query (Q - Consulta):** O que o token atual está procurando? (Representa o foco atual).
2. **Key (K - Chave):** O que o token atual oferece? (Serve como um identificador para correspondência).
3. **Value (V - Valor):** Qual é o conteúdo real da informação? (A informação que será agregada se a chave corresponder à consulta).

Em um mecanismo de *Self-Attention*, Q, K e V originam-se da mesma fonte (a saída da camada anterior), multiplicados por matrizes de pesos aprendíveis (W^Q, W^K, W^V).

7.2 2. A Fórmula Matemática

A atenção é calculada como uma soma ponderada dos valores (V), onde o peso atribuído a cada valor é determinado pela compatibilidade entre a consulta (Q) e a chave correspondente (K).

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

7.2.1 Passo a Passo do Cálculo:

1. **MatMul (QK^T):** Calcula o produto escalar entre a Query e a Key transposta. Isso resulta em uma **Matriz de Scores** (ou afinidade). Se o produto escalar é alto, os vetores estão alinhados, indicando alta relevância entre os tokens.
2. **Scale ($\frac{1}{\sqrt{d_k}}$):** Dividimos os scores pela raiz quadrada da dimensão das chaves (d_k).
    - *Por que escalar?* Sem isso, para valores altos de d_k, o produto escalar cresce em magnitude, empurrando a função Softmax para regiões onde os gradientes são extremamente pequenos (vanishing gradients), dificultando o treinamento.

3. **Mask (Opcional):** Aplica-se uma máscara (substituindo valores por $-\infty$) para impedir que o modelo "veja" certos tokens (ex: tokens futuros ou padding).
4. **Softmax:** Converte os scores em probabilidades (pesos de atenção) que somam 1.
5. **MatMul ($ ◎V) : ∗ ∗ $Multiplica - seospesospelosValores$(V$). Isso filtra as informações irrelevantes e preserva as relevantes.

7.3 3. Fluxo de Dados Visual

O diagrama abaixo ilustra o fluxo de tensores dentro do bloco de atenção escalar.

```
graph TD
    subgraph Inputs
    Q[Query (Q)]
    K[Key (K)]
    V[Value (V)]
    end

    Q --> MatMul1[MatMul (Q x K^T)]
    K --> MatMul1

    MatMul1 --> Scale[Scale (1 / sqrt(d_k))]

    Scale --> MaskNode{Aplicar Máscara?}
    Mask[Máscara Causal / Padding] -.-> MaskNode

    MaskNode -- Sim --> MaskFill[Fill com -inf]
    MaskNode -- Não --> Softmax
```

```
    MaskFill --> Softmax[Softmax]

    Softmax --> MatMul2[MatMul (Weights x V)]
    V --> MatMul2

    MatMul2 --> Output[Output Contextualizado]

    style MatMul1 fill:#f9f,stroke:#333,stroke-width:2px
    style Softmax fill:#ff9,stroke:#333,stroke-width:2px
    style MatMul2 fill:#f9f,stroke:#333,stroke-width:2px
```

7.4 4. Mascaramento (Masking)

Existem dois tipos críticos de máscaras no Transformer:

1. **Padding Mask:** Ignora tokens de preenchimento (pad tokens) usados para igualar o tamanho das sentenças em um batch.
2. **Look-ahead (Causal) Mask:** Essencial para decodificadores (como no GPT). Ao prever o token na posição t, o modelo não pode ter acesso aos tokens nas posições $t + 1, t + 2, ...$

A máscara causal é uma matriz triangular superior (excluindo a diagonal) preenchida com $-\infty$. Quando somada aos scores antes do Softmax:

$$\text{softmax}(x + (-\infty)) = 0$$

Isso garante que a atenção para tokens futuros seja zero.

7.5 5. Implementação em PyTorch

Abaixo, implementamos uma classe robusta para a Scaled Dot-Product Attention, capaz de lidar com máscaras causais.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class ScaledDotProductAttention(nn.Module):
    """
    Calcula a atenção escalonada por produto escalar.
    Fórmula: Softmax(QK^T / sqrt(d_k)) * V
    """

    def __init__(self, dropout_rate=0.1):
        super(ScaledDotProductAttention, self).__init__()
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, query, key, value, mask=None):
        """
        Args:
            query: Tensor de forma (Batch, Heads, Seq_Len_Q, He
            key:   Tensor de forma (Batch, Heads, Seq_Len_K, He
            value: Tensor de forma (Batch, Heads, Seq_Len_K, He
            mask:  Tensor de máscara (Batch, 1, Seq_Len_Q, Seq_
                   Valores True indicam posições a serem mascar

        Returns:
            context: Tensor de saída ponderado
            attention_weights: Pesos de atenção (para visualiza
        """

        # 1. Obter dimensões
        d_k = query.size(-1)
```

```
        # 2. Calcular Scores (Q * K^T)
        # Transpomos as duas últimas dimensões de K para permit
        scores = torch.matmul(query, key.transpose(-2, -1))

        # 3. Escalonamento (Scaling)
        scores = scores / math.sqrt(d_k)

        # 4. Aplicação da Máscara (Se fornecida)
        if mask is not None:
            # Substitui valores onde a máscara é 0 (ou True, de
            # Aqui assumimos que mask=0 significa "esconder"
            scores = scores.masked_fill(mask == 0, float('-inf'

        # 5. Softmax para obter probabilidades
        # Aplicado na última dimensão (dimensão da sequência al
        attention_weights = F.softmax(scores, dim=-1)

        # 6. Dropout (Opcional, mas comum para regularização)
        attention_weights = self.dropout(attention_weights)

        # 7. Multiplicação pelos Valores (Weights * V)
        context = torch.matmul(attention_weights, value)

        return context, attention_weights

# --- Exemplo de Uso com Máscara Causal ---

def create_causal_mask(seq_len):
    """Cria uma máscara triangular inferior para impedir 'olhar
    # tril retorna a parte triangular inferior da matriz
    mask = torch.tril(torch.ones(seq_len, seq_len))
```

```
    return mask

# Configuração de teste
batch_size = 1
heads = 1
seq_len = 4
d_k = 8

# Dados fictícios
q = torch.randn(batch_size, heads, seq_len, d_k)
k = torch.randn(batch_size, heads, seq_len, d_k)
v = torch.randn(batch_size, heads, seq_len, d_k)

# Criar máscara causal (1s visíveis, 0s ocultos)
causal_mask = create_causal_mask(seq_len)
# Expande dimensões para broadcasting: (1, 1, seq_len, seq_len)
causal_mask = causal_mask.unsqueeze(0).unsqueeze(0)

# Instanciar e rodar
attention_layer = ScaledDotProductAttention()
output, weights = attention_layer(q, k, v, mask=causal_mask)

print("Pesos de Atenção (com máscara causal):")
print(weights[0][0])
# Note que a parte superior direita será 0.
```

7.5.1 Análise da Saída do Código

Se executarmos o código acima, a matriz de pesos de atenção (`weights`) terá uma estrutura triangular inferior. Por exemplo, para uma sequência de tamanho 4:

- O token 1 só atende ao token 1.
- O token 2 atende aos tokens 1 e 2.
- O token 3 atende aos tokens 1, 2 e 3.
- As posições onde $j > i$ (futuro) terão peso 0 (após o softmax de $-\infty$).

7.6 Conclusão do Capítulo

O mecanismo de Scaled Dot-Product Attention é eficiente computacionalmente, pois se baseia em operações de matriz altamente otimizadas. A introdução do fator de escala $\sqrt{d_k}$ garante estabilidade numérica, e o uso de máscaras permite que a mesma arquitetura seja utilizada tanto para codificação (entender todo o contexto) quanto para decodificação (gerar texto sequencialmente). No próximo capítulo, veremos como expandir este conceito para **Multi-Head Attention**, permitindo que o modelo foque em diferentes subespaços de representação simultaneamente.

7.7 Auditoria de formas e invariantes

Para $Q, K \in \mathbb{R}^{B\times H\times T\times d_k}$, o produto $QK^\top$ tem forma (B, H, T, T). A softmax deve ser aplicada na última dimensão, pois cada consulta distribui peso sobre as chaves. Multiplicar por $V \in \mathbb{R}^{B\times H\times T\times d_v}$ devolve (B, H, T, d_v).

Três testes simples capturam muitos bugs: cada linha dos pesos deve somar aproximadamente 1; posições futuras devem receber zero após a máscara causal; e o gradiente deve alcançar as projeções de Q, K e V. Máscaras invertidas e softmax no eixo errado frequentemente produzem código executável, porém semanticamente incorreto.

Máscara antes da softmax

Substitua escores proibidos por um valor muito negativo **antes** da softmax. Zerar pesos depois dela quebra a normalização, a menos que a linha seja renormalizada.

7.8 Autoatenção, atenção cruzada e custo

Na **autoatenção**, consultas, chaves e valores são obtidos da mesma sequência. Na **atenção cruzada**, as consultas vêm de uma sequência, enquanto chaves e valores vêm de outra. Esse segundo caso aparece em arquiteturas codificador-decodificador e em modelos multimodais, nos quais representações de imagem ou áudio orientam a geração textual.

A matriz de escores possui $T \times T$ elementos por cabeça. Portanto, a atenção densa tem custo quadrático em memória em relação ao comprimento T. Atenção local, esparsa, por blocos e aproximações lineares procuram reduzir esse gargalo, geralmente trocando alcance global, exatidão ou simplicidade por eficiência.

Exemplo numérico mínimo

Para $q = [1, 0]$, $k_1 = [1, 0]$ e $k_2 = [0, 1]$, os escores são 1 e 0. A softmax dá maior peso ao primeiro valor, mas não transforma a operação em uma escolha rígida: a saída continua sendo uma combinação ponderada dos dois vetores de valor.

7.9 Atenção eficiente sem mudar a fórmula

Uma implementação ingênua materializa toda a matriz $T \times T$ na memória global da GPU. **FlashAttention** reorganiza o cálculo em blocos pequenos que cabem em memória rápida, calcula estatísticas da softmax

incrementalmente e evita armazenar grandes intermediários. O resultado matemático permanece sendo atenção exata; o ganho vem do padrão de acesso à memória.

Essa distinção é importante: complexidade assintótica e tempo real não são sinônimos. Uma operação com o mesmo número de multiplicações pode ser muito mais rápida ao reduzir transferências entre níveis de memória. Por isso, analisar apenas FLOPs não explica todo o desempenho.

7.9.1 Janela deslizante e alcance global

Na atenção de janela, cada token consulta somente uma vizinhança de largura w. O custo passa de aproximadamente T^2 para Tw, mas uma dependência distante precisa atravessar várias camadas ou usar tokens globais. A escolha de w define um compromisso entre memória, latência e velocidade de propagação do contexto.

Modelos híbridos podem alternar camadas locais e globais. Outra possibilidade é usar atenção linear, que reorganiza ou aproxima o cálculo para evitar a matriz quadrática. Essas variantes não são substitutas universais: tarefas de recuperação distante devem fazer parte da avaliação.

Teste de contexto longo

Insira uma informação única no começo de um documento e faça perguntas em distâncias crescentes. Meça acerto e latência; apenas declarar o tamanho máximo da janela não revela quanto contexto o modelo utiliza efetivamente.

Capítulo 8

Multi-Head Attention e Projeções Lineares

Expansão do mecanismo de atenção para múltiplas cabeças (heads), permitindo que o modelo foque em diferentes partes da sequência simultaneamente.

8.1 1. Introdução

No capítulo anterior, exploramos o mecanismo de *Self-Attention* (Autoatenção), que permite ao modelo ponderar a importância de diferentes palavras em uma sequência. No entanto, uma única operação de atenção limita a capacidade do modelo de focar em diferentes tipos de relacionamentos simultaneamente (por exemplo, concordância gramatical *versus* contexto semântico).

O **Multi-Head Attention (MHA)** resolve essa limitação executando múltiplos mecanismos de autoatenção em paralelo. Cada "cabeça" (head) aprende a focar em diferentes subespaços de representação, permitindo que o modelo capture nuances complexas e variadas da linguagem ao mesmo tempo.

8.2 2. O Conceito de Projeções Lineares

Antes de dividir o processamento em múltiplas cabeças, os vetores de entrada (Embeddings) passam por **Projeções Lineares**.

Matematicamente, uma projeção linear é uma multiplicação de matrizes que transforma o vetor de entrada original em um novo espaço vetorial. No contexto do Transformer, para cada cabeça i, treinamos três matrizes de pesos independentes:

1. W_i^Q **(Pesos para Query)**
2. W_i^K **(Pesos para Key)**
3. W_i^V **(Pesos para Value)**

Se a dimensão do modelo é d_{model} e queremos h cabeças, a dimensão de cada cabeça será tipicamente $d_k = d_{model}/h$. As projeções reduzem a dimensionalidade para cada cabeça, mantendo o custo computacional total similar ao de uma única cabeça com dimensão total.

8.3 3. Arquitetura do Multi-Head Attention

O processo ocorre em quatro etapas principais:

1. **Projeção:** Os vetores de entrada (Queries, Keys, Values) são projetados linearmente h vezes com matrizes de pesos diferentes e aprendidas.
2. **Scaled Dot-Product Attention:** O mecanismo de atenção é aplicado em paralelo a cada uma dessas projeções.
3. **Concatenação:** As saídas de todas as cabeças são concatenadas.
4. **Projeção Final:** A concatenação passa por uma última camada linear (W^O) para restaurar a dimensão original e misturar as informações aprendidas pelas diferentes cabeças.

8.3.1 Formulação Matemática

Dado um conjunto de matrizes de entrada Q, K, V:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

Onde cada cabeça é calculada como:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

E a função de Atenção é a padrão:

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

8.4 4. Diagrama de Fluxo de Dados

O diagrama abaixo ilustra como o fluxo de dados é dividido, processado e reunificado.

```
graph TD
    subgraph Inputs
    X[Input Embeddings]
    end

    subgraph "Linear Projections (Split)"
    LP_Q[Linear Proj Q]
    LP_K[Linear Proj K]
    LP_V[Linear Proj V]
    end

    subgraph "Heads (Parallel Processing)"
    H1[Head 1: Scaled Dot-Product]
```

```
    H2[Head 2: Scaled Dot-Product]
    H_dots[...]
    Hn[Head h: Scaled Dot-Product]
    end

    subgraph "Aggregation"
    Concat[Concatenation]
    LinearOut[Linear Proj Output (Wo)]
    end

    X --> LP_Q
    X --> LP_K
    X --> LP_V

    LP_Q -- Split --> H1 & H2 & H_dots & Hn
    LP_K -- Split --> H1 & H2 & H_dots & Hn
    LP_V -- Split --> H1 & H2 & H_dots & Hn

    H1 --> Concat
    H2 --> Concat
    H_dots --> Concat
    Hn --> Concat

    Concat --> LinearOut
    LinearOut --> Output[Final Context Vectors]

    style Inputs fill:#f9f,stroke:#333,stroke-width:2px
    style H1 fill:#bbf,stroke:#333
    style H2 fill:#bbf,stroke:#333
    style Hn fill:#bbf,stroke:#333
    style LinearOut fill:#dfd,stroke:#333
```

8.5 5. Implementação Técnica (PyTorch)

Na prática, não criamos h camadas lineares separadas por ineficiência. Em vez disso, projetamos para a dimensão total (d_{model}) e usamos operações de *reshape* e *transpose* para separar as cabeças logicamente.

Abaixo, uma implementação robusta e anotada:

```
import torch
import torch.nn as nn
import math

class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        assert d_model % num_heads == 0, "d_model deve ser divi

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        # Projeções Lineares (W_q, W_k, W_v)
        # Note que fazemos uma única matriz grande para eficiên
        self.w_q = nn.Linear(d_model, d_model)
        self.w_k = nn.Linear(d_model, d_model)
        self.w_v = nn.Linear(d_model, d_model)

        # Projeção Final (W_o)
        self.w_o = nn.Linear(d_model, d_model)

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        # Q, K, V shape: [batch_size, num_heads, seq_len, d_k]
```

```
        # 1. Matmul Q e K transposto
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sq

        # 2. Aplicar máscara (opcional, usado no Decoder)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        # 3. Softmax
        attn_weights = torch.softmax(scores, dim=-1)

        # 4. Multiplicar pelos Values
        output = torch.matmul(attn_weights, V)
        return output, attn_weights

    def forward(self, q, k, v, mask=None):
        batch_size = q.size(0)

        # 1. Projeções Lineares e Reshape para separar cabeças
        # Transformação: [batch, seq_len, d_model] -> [batch, s
        # Transpose: -> [batch, num_heads, seq_len, d_k] para f
        Q = self.w_q(q).view(batch_size, -1, self.num_heads, se
        K = self.w_k(k).view(batch_size, -1, self.num_heads, se
        V = self.w_v(v).view(batch_size, -1, self.num_heads, se

        # 2. Aplicar Atenção em todas as cabeças simultaneament
        attn_output, _ = self.scaled_dot_product_attention(Q, K

        # 3. Concatenação
        # Transpose reverso: [batch, num_heads, seq_len, d_k] -
        # Contiguous + View: -> [batch, seq_len, d_model]
        attn_output = attn_output.transpose(1, 2).contiguous().
```

```
        # 4. Projeção Linear Final
        output = self.w_o(attn_output)

        return output
```

8.6 6. Por que Múltiplas Cabeças? (Intuição)

Imagine a frase: *"O banco negou o empréstimo porque ele estava sem fundos."*

Para entender a palavra **"ele"**, o modelo precisa relacioná-la a outras palavras. * **Cabeça 1 (Sintática):** Pode focar na estrutura gramatical, ligando "ele" ao sujeito da oração anterior. * **Cabeça 2 (Semântica):** Pode focar no contexto financeiro ("banco", "empréstimo", "fundos") para desambiguar se "ele" se refere ao banco ou ao solicitante (neste caso, o contexto sugere o banco).

Se tivéssemos apenas uma cabeça de atenção, o modelo teria que fazer uma média ponderada de todos esses relacionamentos, potencialmente diluindo informações cruciais. Com o Multi-Head Attention, cada cabeça se especializa em um aspecto diferente da linguagem, resultando em uma representação muito mais rica e robusta.

8.7 7. Resumo

- **Multi-Head Attention** permite o processamento paralelo de diferentes subespaços de representação.
- **Projeções Lineares** (W^Q, W^K, W^V) são usadas para transformar a entrada em dimensões específicas para cada cabeça.
- A saída é uma combinação linear (**concatenação +** W^O) das visões de todas as cabeças, fornecendo ao restante da rede uma visão contextual completa da sequência.

8.8 8. Contabilidade de dimensões

Com $d_{model} = 512$ e $H = 8$, cada cabeça costuma usar $d_k = 64$. Depois das projeções, reorganizamos $(B, T, 512)$ para $(B, 8, T, 64)$. A atenção opera de forma independente por cabeça; em seguida, o `transpose` e o `reshape` reconstroem $(B, T, 512)$ antes de W^O.

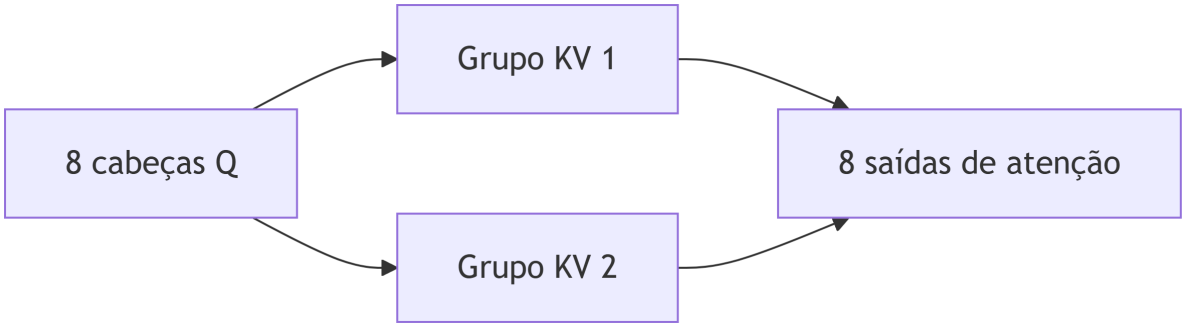
Esse rearranjo não cria informação nem altera a quantidade de elementos. Ele apenas muda a interpretação dos eixos. Em PyTorch, uma transposição pode produzir memória não contígua; por isso, `reshape` ou `contiguous().view(...)` deve ser usado com consciência.

> **!** Especialização é uma hipótese observável
>
> Dizer que uma cabeça é "sintática" é uma intuição útil, não uma regra imposta pela arquitetura. A função de cada cabeça emerge do treinamento e pode ser redundante, distribuída ou variar entre entradas.

8.9 MHA, MQA e GQA

Na atenção multicabeças convencional (**MHA**), cada cabeça possui suas próprias chaves e valores. Durante a geração, esses tensores de todos os tokens anteriores são mantidos no cache, o que pode consumir muita memória. A atenção multi-query (**MQA**) compartilha um único conjunto de chaves e valores entre todas as cabeças de consulta. A atenção de consultas agrupadas (**GQA**) ocupa o meio-termo: várias cabeças de consulta compartilham cada grupo de K e V.

Se há $H_q = 8$ cabeças de consulta e $H_{kv} = 2$ grupos de chaves e valores, quatro consultas compartilham cada grupo. A saída ainda contém oito perspectivas de consulta, mas o cache armazena somente duas versões de K e V por camada. Isso reduz memória e largura de banda durante o decode.



Compartilhar demais pode reduzir capacidade; compartilhar pouco economiza menos. O número de grupos é, portanto, um hiperparâmetro arquitetural avaliado junto com qualidade e velocidade.

8.10 Contagem aproximada de parâmetros

Com projeções densas e $d_{model} = d$, o MHA usual possui aproximadamente $4d^2$ parâmetros: três projeções para Q, K, V e uma projeção de saída. Dividir em cabeças não multiplica automaticamente esse total quando as dimensões concatenadas continuam somando d. O custo muda quando as dimensões de chaves e valores ou o número de grupos são alterados.

Capítulo 9

A Arquitetura do Bloco Transformer

O "Bloco Transformer" é a unidade atômica fundamental de arquiteturas baseadas em *Large Language Models* (LLMs). Enquanto o mecanismo de *Self-Attention* (discutido em capítulos anteriores) é responsável por contextualizar tokens entre si, o Bloco Transformer é a estrutura que encapsula esse mecanismo, processa a informação extraída e estabiliza o fluxo de gradientes através da rede profunda.

Um modelo moderno (como GPT-4, LLaMA ou Claude) é essencialmente uma pilha vertical de dezenas ou centenas desses blocos idênticos.

9.1 1. Visão Geral da Arquitetura

O Transformer original combina um **codificador**, que constrói representações bidirecionais da entrada, e um **decodificador**, que gera a saída autoregressivamente e consulta o codificador por atenção cruzada. Modelos como BERT usam principalmente a pilha codificadora; modelos no estilo GPT usam somente blocos decodificadores com máscara causal; tarefas de tradução e transformação sequência-a-sequência frequentemente usam as duas pilhas.

Independentemente da família, os blocos reutilizam quatro ideias: atenção para misturar informação entre posições, redes feed-forward para transformar cada posição, conexões residuais para preservar caminhos de sinal e normalização para estabilizar o treinamento.

A arquitetura moderna do bloco divergiu ligeiramente do paper original "Attention is All You Need" (2017). A mudança mais significativa é a adoção da configuração **Pre-Norm** (normalização antes das subcamadas) em vez de **Post-Norm**, para garantir estabilidade no treinamento de modelos profundos.

O fluxo de dados dentro de um único bloco segue esta ordem lógica: 1. Entrada (x) 2. Normalização 1 3. Self-Attention 4. Conexão Residual 1 (Soma com a entrada) 5. Normalização 2 6. Feed-Forward Network (FFN) 7. Conexão Residual 2 (Soma com o resultado anterior)

9.1.1 Diagrama de Fluxo de Dados (Pre-Norm)

```
graph TD
    Input[Entrada do Bloco <br/> Tensor: Batch, Seq, Dim] --> S

    %% Caminho Residual 1
    Split1 -->|Caminho Residual| Add1((Soma))

    %% Caminho de Atenção
    Split1 --> Norm1[RMSNorm / LayerNorm]
    Norm1 --> Attn[Multi-Head Self-Attention]
    Attn --> Add1

    %% Caminho Residual 2
    Add1 --> Split2(( ))
    Split2 -->|Caminho Residual| Add2((Soma))
```

```
    %% Caminho FFN
    Split2 --> Norm2[RMSNorm / LayerNorm]
    Norm2 --> FFN[Feed-Forward Network <br/> Projeção -> Ativaç
    FFN --> Add2

    Add2 --> Output[Saída do Bloco]

    style Input fill:#f9f,stroke:#333,stroke-width:2px
    style Output fill:#f9f,stroke:#333,stroke-width:2px
    style Add1 fill:#bbf,stroke:#333
    style Add2 fill:#bbf,stroke:#333
```

9.2 2. Conexões Residuais (Skip Connections)

As conexões residuais são vitais para o treinamento de redes neurais profundas. Matematicamente, se considerarmos uma subcamada (como a Atenção) como uma função $F(x)$, a saída do bloco não é apenas $F(x)$, mas sim:

$$Output = x + F(x)$$

9.2.1 Função Técnica

1. **Mitigação do Desvanecimento de Gradiente (Vanishing Gradient):** Durante a retropropagação (*backpropagation*), o gradiente pode fluir diretamente através da conexão "skip" (x) sem ser atenuado pelas operações complexas da função $F(x)$. Isso cria uma "superestrada" para o gradiente fluir das últimas camadas até as primeiras.

2. **Preservação de Informação:** Permite que o modelo retenha informações da entrada original se a transformação $F(x)$ não for necessária ou for destrutiva naquele estágio específico.

9.3 3. Normalização: LayerNorm vs. RMSNorm

A normalização é responsável por manter a estabilidade numérica dos valores dos neurônios (ativações), garantindo que eles não explodam nem desapareçam.

9.3.1 Layer Normalization (LayerNorm)

A abordagem clássica. Para um vetor de entrada x, a LayerNorm calcula a média μ e a variância σ^2 de *todas as dimensões* daquele vetor específico (independente do batch).

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \cdot \gamma + \beta$$

Onde γ e β são parâmetros aprendíveis de escala e deslocamento (bias).

9.3.2 Root Mean Square Normalization (RMSNorm)

Adotada por modelos modernos como LLaMA, Gopher e T5. A RMSNorm baseia-se na hipótese de que a recentralização (subtrair a média μ) não é fundamental para a convergência, apenas a reescala.

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\text{RMS}(x)^2 + \epsilon}} \cdot \gamma$$

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

Vantagens da RMSNorm: * **Eficiência Computacional:** Remove o cálculo da média, economizando operações, o que é significativo em modelos com bilhões de parâmetros. * **Simplicidade:** Mantém a invariância de escala sem a complexidade do deslocamento de média.

9.4 4. Camadas Feed-Forward (FFN)

Enquanto a Atenção agrega informações de diferentes tokens (mistura temporal/contextual), a FFN processa essas informações para cada token individualmente (processamento de características). É frequentemente descrita como a "memória associativa" do modelo.

9.4.1 Estrutura Clássica (MLP)

Uma FFN padrão consiste em duas transformações lineares com uma função de ativação não-linear no meio. Geralmente, ela expande a dimensão oculta (frequentemente por um fator de 4) e depois a projeta de volta.

$$\text{FFN}(x) = \text{Ativação}(xW_1 + b_1)W_2 + b_2$$

9.4.2 Estrutura Moderna (SwiGLU / Gated Linear Units)

Modelos de ponta (como PaLM e LLaMA) utilizam variantes "Gated" (com portas), especificamente a **SwiGLU**.

Nesta configuração, existem três matrizes de peso em vez de duas, e a ativação funciona como um portão lógico suave:

$$\text{SwiGLU}(x) = (\text{Swish}(xW_{gate}) \otimes (xW_{in}))W_{out}$$

- W_{in}: Projeta para a dimensão expandida.
- W_{gate}: Cria uma "máscara" de ativação.
- $\otimes$: Multiplicação elemento a elemento (Hadamard product).
- W_{out}: Projeta de volta para a dimensão do modelo.

Isso permite um controle mais refinado sobre quais informações fluem através da rede.

9.5 5. Implementação de Referência (PyTorch)

Abaixo, uma implementação de um Bloco Transformer moderno utilizando **Pre-Norm**, **RMSNorm** e uma estrutura **MLP** padrão para clareza.

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

class RMSNorm(nn.Module):
    def __init__(self, dim: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        # Parâmetro de escala aprendível (gamma)
        self.weight = nn.Parameter(torch.ones(dim))

    def forward(self, x):
        # Cálculo do Root Mean Square
        var = torch.mean(x ** 2, dim=-1, keepdim=True)
```

```
        x_norm = x * torch.rsqrt(var + self.eps)
        return self.weight * x_norm

class FeedForward(nn.Module):
    def __init__(self, dim: int, hidden_dim: int, dropout: floa
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(dim, hidden_dim),
            nn.GELU(), # Ativação moderna (Gaussian Error Linea
            nn.Linear(hidden_dim, dim),
            nn.Dropout(dropout)
        )

    def forward(self, x):
        return self.net(x)

class TransformerBlock(nn.Module):
    def __init__(self, dim: int, num_heads: int, mlp_ratio: flo
        super().__init__()
        self.norm1 = RMSNorm(dim)
        self.attn = nn.MultiheadAttention(embed_dim=dim, num_he

        self.norm2 = RMSNorm(dim)
        hidden_dim = int(dim * mlp_ratio)
        self.ffn = FeedForward(dim, hidden_dim, dropout)

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # 1. Pre-Norm + Atenção + Residual
        # Nota: A entrada 'x' é preservada para a soma (skip co
```

```
        norm_x = self.norm1(x)
        attn_out, _ = self.attn(norm_x, norm_x, norm_x)
        x = x + self.dropout(attn_out)

        # 2. Pre-Norm + FFN + Residual
        # Nota: O 'x' atualizado é preservado novamente
        norm_x = self.norm2(x)
        ffn_out = self.ffn(norm_x)
        x = x + self.dropout(ffn_out)

        return x

# Exemplo de instanciação
# Dimensão do modelo: 512, Cabeças: 8
block = TransformerBlock(dim=512, num_heads=8)
dummy_input = torch.randn(1, 10, 512) # Batch=1, Seq=10, Dim=51
output = block(dummy_input)
print(f"Shape de saída: {output.shape}")
```

9.6 Resumo do Capítulo

A eficácia do Bloco Transformer reside na orquestração cuidadosa de seus componentes: 1. **RMSNorm (Pre-Norm):** Estabiliza a entrada antes do processamento pesado. 2. **Atenção:** Mistura informações entre tokens. 3. **FFN:** Processa e "raciocina" sobre as informações misturadas, expandindo a dimensionalidade para capturar relações não-lineares complexas. 4. **Residuais:** Garantem que o sinal original nunca seja perdido e que os gradientes fluam saudavelmente durante o treinamento.

9.7 Contrato de um bloco bem implementado

O bloco deve preservar a forma (B, T, d_{model}), respeitar a máscara causal e aceitar diferentes comprimentos de sequência. Dropout deve atuar apenas em modo de treino, e os parâmetros de normalização precisam acompanhar o dispositivo e o tipo numérico do restante do modelo.

No esquema *pre-norm*, podemos escrever $y = x + \text{Attention}(\text{Norm}(x))$ e $z = y + \text{FFN}(\text{Norm}(y))$. Essa notação evidencia os dois caminhos residuais e ajuda a conferir se a implementação somou cada saída à entrada correta.

> **Teste unitário sugerido**
>
> Passe um tensor aleatório, confira forma e ausência de `NaN`, execute `output.sum().backward()` e verifique se os principais pesos receberam gradientes. Depois aplique uma máscara e confirme que alterar um token futuro não muda a saída de posições anteriores.

9.7.1 O tensor ao atravessar o bloco

O formato externo permanece (B, T, d_{model}). A atenção mistura informações **entre tokens**; a FFN transforma as características de cada token **independentemente**, usando pesos compartilhados entre posições. Empilhar blocos preserva essas dimensões, mas produz representações progressivamente mais contextualizadas.

9.8 Escolhas modernas dentro do bloco

Transformers iniciais usavam LayerNorm e redes feed-forward com ReLU. Blocos modernos frequentemente combinam **RMSNorm**, organização *pre-norm* e ativações com portas, como SwiGLU. Essas

escolhas não mudam a função geral do bloco, mas afetam estabilidade, capacidade e custo.

A RMSNorm normaliza pela magnitude quadrática sem subtrair a média:

$$\text{RMSNorm}(x) = g \odot \frac{x}{\sqrt{\frac{1}{d}\sum_i x_i^2 + \epsilon}},$$

onde g é um vetor aprendido. Ela possui menos operações que LayerNorm, embora a vantagem prática dependa do kernel e do hardware.

Na SwiGLU, duas projeções seguem caminhos distintos; uma é transformada pela função SiLU e atua como porta sobre a outra:

$$\text{SwiGLU}(x) = (\text{SiLU}(xW_g) \odot xW_u)W_d.$$

O mecanismo permite controlar quais características seguem adiante. Como há projeções adicionais, a largura intermediária costuma ser escolhida para equilibrar parâmetros com uma FFN convencional.

9.8.1 Por que as conexões residuais ajudam

Em $y = x + F(x)$, a derivada contém um caminho identidade além da derivada de F. Isso oferece ao gradiente uma rota direta através de muitas camadas e permite que uma subcamada aprenda apenas a correção necessária. A conexão residual não impede todo problema de otimização, mas reduz a dependência de transformar perfeitamente o sinal em cada etapa.

> ⚠ Normalizações não são intercambiáveis
>
> BatchNorm usa estatísticas do lote e é sensível ao tamanho do batch. LayerNorm e RMSNorm operam sobre características de cada token, comportamento mais adequado a sequências de comprimentos e lotes variados.

Capítulo 10

Montando o Modelo Completo (Estilo GPT)

Integração de todos os componentes desenvolvidos anteriormente para criar uma arquitetura Decoder-Only funcional pronta para gerar texto.

Este capítulo integra um **Transformer somente decodificador**, paradigma usado por modelos da família GPT e Llama. Diferentemente do Transformer codificador-decodificador original, essa organização é voltada à modelagem autoregressiva: prever cada próximo token apenas com o contexto anterior.

10.1 1. Arquitetura em Alto Nível

A arquitetura contém embeddings aprendidos, uma pilha de blocos decodificadores e uma projeção final que mapeia as representações internas de volta ao espaço do vocabulário.

10.1.1 Diagrama da Arquitetura

O diagrama mostra o caminho dos IDs de tokens até os logits. Observe a normalização anterior às subcamadas (*pre-norm*), comum em implemen-

tações modernas por favorecer a estabilidade de redes profundas.

```
graph TD
    subgraph Inputs
        I[Input Token IDs] --> TE[Token Embeddings]
        P[Position IDs] --> PE[Positional Embeddings]
    end

    TE & PE --> Sum((+))
    Sum --> Drop1[Dropout]

    subgraph "Transformer Block (Repeated N times)"
        Drop1 --> LN1[Layer Norm 1]
        LN1 --> MSA[Masked Multi-Head Attention]
        MSA --> Res1((+))
        Drop1 --> Res1

        Res1 --> LN2[Layer Norm 2]
        LN2 --> MLP[Feed-Forward Network<br/>(GELU Activation)]
        MLP --> Res2((+))
        Res1 --> Res2
    end

    Res2 --> LNF[Final Layer Norm]
    LNF --> Head[Linear Head<br/>(Project to Vocab Size)]
    Head --> Logits[Logits]
    Logits --> Soft[Softmax]
    Soft --> Prob[Next Token Probabilities]

    style Inputs fill:#f9f,stroke:#333,stroke-width:2px
    style MSA fill:#bbf,stroke:#333,stroke-width:2px
    style MLP fill:#bbf,stroke:#333,stroke-width:2px
```

```
    style Head fill:#bfb,stroke:#333,stroke-width:2px
```

10.2 2. Integração dos Componentes

To assemble the model, we integrate the components defined in previous chapters. The architecture is defined by hyperparameters: d_{model} (embedding dimension), N_{layers} (depth), N_{heads} (attention heads), and V (vocabulary size).

10.2.1 A. Camada de Embeddings

É a entrada do modelo: converte índices discretos em vetores densos e incorpora informação de posição. * **Token Embeddings:** A lookup table of size (V, d_{model}). * **Positional Embeddings:** A lookup table of size $(ContextLen, d_{model})$. In GPT, these are typically learned parameters rather than fixed sinusoidal functions. * **Combination:** The two embeddings are summed element-wise.

10.2.2 B. Bloco Decodificador

A maior parte da computação acontece aqui. Cada bloco contém normalização, autoatenção com máscara causal, uma rede feed-forward e duas conexões residuais. A máscara garante que a posição t consulte apenas posições até t, preservando o objetivo autoregressivo.

10.2.3 C. Cabeça de Saída

After passing through N blocks: 1. **Normalização final:** estabiliza os estados ocultos finais. 2. **Projeção linear:** mapeia d_{model} para V, o tama-

nho do vocabulário. 3. **Compartilhamento de pesos:** opcionalmente, a projeção usa a mesma matriz dos embeddings de entrada, reduzindo parâmetros.

10.3 3. Lógica da Implementação

Below is a structural representation of the complete model using a PyTorch-like class structure. This demonstrates how the components interact in code.

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

class GPT(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config

        # 1. Embeddings
        self.token_embedding = nn.Embedding(config.vocab_size,
        self.position_embedding = nn.Embedding(config.block_siz
        self.drop = nn.Dropout(config.dropout)

        # 2. Stack of Decoder Blocks
        # Assuming 'Block' is a class defined in previous chapt
        self.blocks = nn.ModuleList([
            Block(config) for _ in range(config.n_layer)
        ])
```

```
        # 3. Final Normalization
        self.ln_f = nn.LayerNorm(config.n_embd)

        # 4. Output Head
        self.head = nn.Linear(config.n_embd, config.vocab_size,

        # Weight tying: embedding weights == output head weight
        self.token_embedding.weight = self.head.weight

        self.apply(self._init_weights)

    def _init_weights(self, module):
        """ Initialize weights (typically normal distribution w
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=

    def forward(self, idx, targets=None):
        # idx shape: (Batch, Seq_Len)
        B, T = idx.shape

        # Create position indices: [0, 1, 2, ..., T-1]
        pos = torch.arange(0, T, dtype=torch.long, device=idx.d

        # Forward pass through embeddings
        tok_emb = self.token_embedding(idx) # (B, T, n_embd)
        pos_emb = self.position_embedding(pos) # (T, n_embd)
        x = self.drop(tok_emb + pos_emb)
```

```
        # Forward pass through transformer blocks
        for block in self.blocks:
            x = block(x)

        # Final Norm
        x = self.ln_f(x)

        # Project to vocabulary
        logits = self.head(x) # (B, T, vocab_size)

        loss = None
        if targets is not None:
            # Flatten for CrossEntropyLoss
            # logits: (B*T, vocab_size), targets: (B*T)
            loss = F.cross_entropy(logits.view(-1, logits.size(

        return logits, loss
```

10.4 4. Forward de Treino versus Geração

It is crucial to distinguish how this model operates during training versus generation.

10.4.1 Treinamento em paralelo

No treinamento, a sequência correta inteira está disponível. Para entrada `[A, B, C]`, o alvo é `[B, C, D]`. A máscara causal permite calcular simultaneamente as previsões de `B` dado `A`, de `C` dado `AB` e de `D` dado `ABC`,

sem revelar o futuro. A entropia cruzada é calculada em todas as posições válidas.

10.4.2 Inferência sequencial

Na geração, o futuro é desconhecido. Partimos de `[A]`, selecionamos `B` a partir dos logits, anexamos o token e repetimos com `[A,B]`. O ciclo termina ao gerar `<EOS>` ou atingir o limite configurado. Um cache de chaves e valores evita recalcular toda a atenção do prefixo em cada passo.

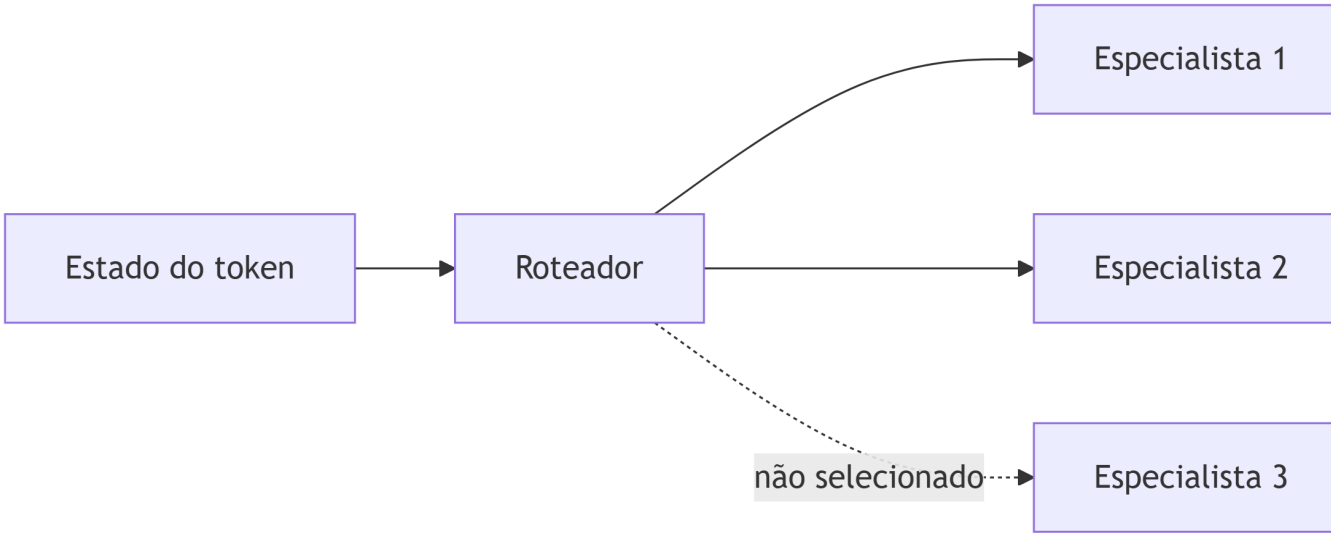
10.5 5. Resumo da Integração

O modelo no estilo GPT combina embeddings, blocos causais e uma cabeça de vocabulário em um caminho inteiramente diferenciável. Durante o treino, aprende regularidades úteis para prever o próximo token; durante a inferência, aplica essa mesma função repetidamente para produzir uma continuação.

> **Teste de causalidade**
>
> Altere o último token de uma sequência e execute o modelo em modo de avaliação. Os logits das posições anteriores devem permanecer iguais, salvo pequenas diferenças numéricas. Se mudarem, a máscara está permitindo vazamento do futuro.

10.6 Modelos densos e mistura de especialistas

No Transformer denso, todos os tokens atravessam a mesma FFN de cada bloco. Uma camada de **mistura de especialistas** (*Mixture of Experts*, MoE) contém várias FFNs e um roteador que escolhe um pequeno subconjunto para cada token. Assim, o número total de parâmetros pode crescer sem que todos sejam ativados em cada passagem.



O roteador produz escores e seleciona, por exemplo, os dois especialistas mais relevantes. A saída combina esses resultados com pesos normalizados. O modelo continua diferenciável nas rotas selecionadas, mas o treinamento precisa de termos auxiliares para evitar que quase todos os tokens sejam enviados aos mesmos especialistas.

MoE separa **parâmetros disponíveis** de **parâmetros ativos por token**. Isso pode aumentar capacidade mantendo custo de cálculo controlado, porém adiciona comunicação entre dispositivos, balanceamento de carga e limites de capacidade por especialista. Um modelo com mais parâmetros totais não é automaticamente mais lento ou melhor; a comparação deve considerar parâmetros ativos, memória e arquitetura.

10.7 Contabilidade do caminho completo

Para IDs com forma (B, T), os embeddings produzem (B, T, d). A pilha conserva essa forma, a normalização final também, e a cabeça de linguagem gera (B, T, V). No treino, todas as posições contribuem em paralelo. Na geração, normalmente só os logits da última posição decidem o novo token.

! Configuração é parte do checkpoint

Número de camadas, dimensão, cabeças, grupos KV, vocabulário, tipo de posição e normalização devem acompanhar os pesos. Matrizes com formas compatíveis ainda podem representar arquiteturas semanticamente incompatíveis.

Capítulo 11

O Loop de Treinamento e Otimização

Implementação do ciclo de treinamento, cálculo da função de perda (Cross-Entropy), Backpropagation e configuração do otimizador AdamW.

O "coração" de qualquer sistema de Deep Learning é o loop de treinamento. É neste ciclo iterativo que os parâmetros do modelo (pesos e vieses) são ajustados matematicamente para minimizar o erro entre as previsões do modelo e os dados reais.

Neste capítulo, dissecamos a arquitetura do ciclo de treinamento, a matemática por trás da função de perda *Cross-Entropy*, a mecânica da *Backpropagation* e a justificativa para o uso do otimizador **AdamW** em arquiteturas baseadas em Transformers.

11.1 1. Visão Geral do Ciclo de Treinamento

No pré-treinamento autossupervisionado, os próprios textos fornecem os rótulos: cada sequência de entrada é deslocada para criar os tokens-alvo. Isso permite aprender com grandes corpora sem anotação manual, mas transfere ao modelo padrões, lacunas e vieses presentes nas fon-

tes. Deduplicação, normalização, remoção de dados pessoais e balanceamento entre domínios fazem parte do problema de modelagem, não apenas de limpeza.

Qualidade e diversidade importam tanto quanto volume. Um corpus deve representar idiomas, gêneros e conhecimentos relevantes para a aplicação, respeitando licenças e privacidade. Conjuntos contaminados por exemplos de avaliação tornam métricas posteriores pouco confiáveis.

O treinamento de uma rede neural é um processo de otimização estocástica. O objetivo é encontrar um conjunto de parâmetros θ que minimize uma função de custo $J(\theta)$.

O fluxo de dados ocorre em duas direções: 1. **Forward Pass (Propagação):** Os dados fluem da entrada para a saída, gerando previsões (logits). 2. **Backward Pass (Retropropagação):** O erro flui da saída para a entrada, calculando o gradiente da perda em relação a cada parâmetro.

11.1.1 Diagrama de Fluxo do Treinamento

O diagrama abaixo ilustra a interação entre os dados, o modelo, a função de perda e o otimizador.

```
flowchart TD
    subgraph Data_Preparation [Preparação de Dados]
        Batch[Batch de Treinamento]
        Targets[Labels/Targets]
    end

    subgraph Forward_Pass [Forward Pass]
        Params[(Parâmetros do Modelo)]
        Model[Arquitetura Transformer]
        Logits[Logits / Previsões]
```

```
        Params --> Model
        Batch --> Model
        Model --> Logits
    end

    subgraph Optimization_Cycle [Ciclo de Otimização]
        LossFn{Cross-Entropy Loss}
        Gradients[Gradientes]
        Optimizer[Otimizador AdamW]

        Logits --> LossFn
        Targets --> LossFn
        LossFn -- "loss.backward()" --> Gradients
        Gradients --> Optimizer
        Optimizer -- "step()" --> Params
    end

    style LossFn fill:#f96,stroke:#333,stroke-width:2px
    style Optimizer fill:#69f,stroke:#333,stroke-width:2px
    style Params fill:#eee,stroke:#333,stroke-width:2px
```

11.2 2. A Função de Perda: Cross-Entropy

Para modelos de linguagem (LLMs), o objetivo é prever o próximo *token* em uma sequência. Isso é tratado como um problema de classificação multiclasse, onde o número de classes é igual ao tamanho do vocabulário (V).

A função de perda padrão é a **Cross-Entropy (Entropia Cruzada)**. Ela mede a divergência entre duas distribuições de probabilidade: 1.

P **(Distribuição Real):** Uma distribuição *one-hot* representando o token verdadeiro. 2. Q **(Distribuição Predita):** As probabilidades geradas pelo modelo (após a aplicação de *Softmax* nos logits).

A fórmula matemática para um único exemplo é:

$$H(P, Q) = -\sum_{x} P(x) \log Q(x)$$

No contexto de PyTorch, a classe `nn.CrossEntropyLoss` combina duas operações para estabilidade numérica: `LogSoftmax` e `NLLLoss` (Negative Log Likelihood Loss).

Por que Cross-Entropy? Ela penaliza fortemente previsões confiantes, porém erradas. Se o modelo atribui uma probabilidade baixa ao token correto, o logaritmo dessa probabilidade resulta em um valor negativo grande, aumentando drasticamente a perda.

11.3 3. O Otimizador: AdamW

Embora o *Stochastic Gradient Descent* (SGD) seja a base, modelos modernos (especialmente Transformers) utilizam variantes adaptativas. O padrão da indústria atual é o **AdamW**.

11.3.1 Adam vs. AdamW

- **Adam (Adaptive Moment Estimation):** Calcula taxas de aprendizado adaptativas para cada parâmetro. Ele armazena médias móveis exponenciais dos gradientes passados (m_t, primeiro momento) e dos gradientes ao quadrado (v_t, segundo momento).
- **O Problema do Adam:** A implementação original do Adam aplicava a regularização L2 (Weight Decay) *junto* com o cálculo do gradiente adaptativo. Isso acoplava a regularização à magnitude dos

gradientes, o que muitas vezes levava a uma generalização subótima.

- **A Solução AdamW (Decoupled Weight Decay):** O AdamW desacopla o decaimento de peso da atualização do gradiente. O decaimento de peso é aplicado diretamente aos parâmetros *após* o passo de otimização principal.

A atualização de parâmetros no AdamW segue a lógica:

1. Atualiza parâmetros baseados nos gradientes adaptativos (Adam clássico).
2. Aplica decaimento de peso: $\theta_t = \theta_t - \eta\lambda\theta_{t-1}$ (onde λ é o fator de decaimento).

Isso permite usar taxas de decaimento de peso mais agressivas sem prejudicar o aprendizado, resultando em modelos que generalizam melhor.

11.4 4. Implementação Técnica

Abaixo apresentamos uma implementação robusta do loop de treinamento utilizando PyTorch.

11.4.1 Estrutura do Código

```
import torch
import torch.nn as nn
from torch.optim import AdamW
from torch.utils.data import DataLoader
```

```
def train_epoch(
    model: nn.Module,
    dataloader: DataLoader,
    optimizer: torch.optim.Optimizer,
    device: torch.device,
    clip_grad: float = 1.0
) -> float:
    """
    Executa uma época de treinamento.

    Args:
        model: O modelo Transformer.
        dataloader: Iterador de dados em batches.
        optimizer: Otimizador configurado (AdamW).
        device: CPU ou CUDA.
        clip_grad: Valor para recorte de gradiente (Gradient Cl

    Returns:
        Média da perda (loss) na época.
    """
    model.train() # Coloca o modelo em modo de treinamento (ati
    total_loss = 0.0

    # Definição da Função de Perda
    # ignore_index é usado para ignorar tokens de padding no cá
    criterion = nn.CrossEntropyLoss(ignore_index=0)

    for batch_idx, (inputs, targets) in enumerate(dataloader):
        # 1. Mover dados para o dispositivo (GPU/TPU)
        inputs = inputs.to(device)
        targets = targets.to(device)
```

```
        # 2. Zerar os gradientes acumulados
        # Essencial: PyTorch acumula gradientes por padrão.
        optimizer.zero_grad(set_to_none=True)

        # 3. Forward Pass
        # O modelo retorna logits de forma (Batch, Seq_Len, Voc
        logits = model(inputs)

        # 4. Reshape para cálculo da Loss
        # CrossEntropy espera (N, C) ou (N, C, d1...)
        # Achatamos Batch e Seq_Len para tratar cada token como
        B, T, C = logits.shape
        logits = logits.view(B * T, C)
        targets = targets.view(B * T)

        # 5. Cálculo da Perda
        loss = criterion(logits, targets)

        # 6. Backward Pass (Backpropagation)
        # Calcula dLoss/dWeights para todos os parâmetros
        loss.backward()

        # 7. Gradient Clipping
        # Crucial em Transformers para evitar explosão de gradi
        torch.nn.utils.clip_grad_norm_(model.parameters(), clip

        # 8. Passo do Otimizador
        # Atualiza os pesos: w = w - lr * grad
        optimizer.step()

        total_loss += loss.item()
```

```
        # (Opcional) Logging intra-época
        if batch_idx % 100 == 0:
            print(f"Batch {batch_idx} | Loss: {loss.item():.4f}

    return total_loss / len(dataloader)

# Exemplo de Configuração do Otimizador
# model = MyTransformer(...)
# optimizer = AdamW(model.parameters(), lr=3e-4, betas=(0.9, 0.
```

11.5 5. Detalhes Críticos de Arquitetura

11.5.1 Gradient Accumulation (Acumulação de Gradientes)

Se a memória da GPU não suportar o tamanho de batch desejado (ex: 512), usamos acumulação de gradientes. Executamos o *forward* e *backward* várias vezes com micro-batches menores, mas só chamamos `optimizer.step()` após N passos.

11.5.2 Gradient Clipping

Transformers são profundos e sensíveis. Gradientes podem crescer exponencialmente (*exploding gradients*), desestabilizando o treino. O `clip_grad_norm_` reescala o vetor de gradientes global se sua norma exceder um limiar (geralmente 1.0), mantendo a direção mas limitando a magnitude.

11.5.3 `set_to_none=True`

Ao zerar gradientes, usar `optimizer.zero_grad(set_to_none=True)` é ligeiramente mais eficiente em termos de memória do que definir os tensores como zero, pois evita operações de leitura/escrita de memória desnecessárias.

11.6 Resumo do Capítulo

O sucesso do treinamento de um LLM depende do equilíbrio delicado entre a arquitetura do modelo e o algoritmo de otimização. Utilizamos **Cross-Entropy** para guiar o modelo probabilisticamente e **AdamW** para navegar na superfície de erro complexa de forma eficiente, desacoplando a regularização da adaptação do passo. A implementação correta do loop, incluindo salvaguardas como *Gradient Clipping*, é mandatória para a convergência do modelo.

11.7 Instrumentação e depuração

Registre perda, taxa de aprendizado, norma dos gradientes, tokens processados por segundo e uso de memória. Uma perda que se torna `NaN` pode indicar taxa alta, overflow em precisão reduzida, dados inválidos ou máscara incorreta. A norma dos gradientes ajuda a distinguir explosão de gradiente de um problema na entrada.

Um *batch* efetivo com acumulação é `micro_batch × passos_de_acumulaçã` `× número_de_dispositivos`. Divida a perda pelos passos de acumulação antes de `backward()` para manter a escala do gradiente equivalente à média do lote maior.

> ⚠ Ordem das operações
>
> Com *mixed precision*, desfaça a escala dos gradientes antes do clipping. Só então execute o passo do otimizador, atualize o scaler e zere os gradientes.

11.8 Escala de modelo, dados e computação

Resultados empíricos mostram relações previsíveis entre perda, quantidade de dados, parâmetros e computação, mas aumentar apenas uma dimensão pode desperdiçar recursos. O planejamento deve equilibrar capacidade do modelo e quantidade de tokens, além de considerar inferência, energia e custo de avaliação. Capacidades observadas em escala precisam ser medidas diretamente; não devem ser presumidas apenas pela contagem de parâmetros.

11.9 Overfitting, underfitting e regularização

Underfitting ocorre quando o modelo ou o procedimento de treino não consegue reduzir adequadamente nem a perda de treinamento. Pode indicar capacidade insuficiente, poucos passos, taxa de aprendizado inadequada ou dados difíceis. **Overfitting** aparece quando o desempenho no treino melhora, mas a validação deixa de acompanhar ou piora.

Regularização não é uma única técnica. *Weight decay* desencoraja pesos excessivos; dropout perturba ativações durante o treino; parada antecipada escolhe um checkpoint anterior à degradação de validação; diversidade de dados reduz a dependência de padrões estreitos. Em LLMs, deduplicação e controle da proporção entre fontes também funcionam como regularização do corpus.

```
loss de treino:      3.2 -> 2.4 -> 1.8 -> 1.3
loss de validação:   3.3 -> 2.6 -> 2.5 -> 2.8
                                       ^ melhor checkpoint
```

> ⚠ Treinar por mais tempo pode piorar
>
> Uma loss de treino menor não compensa perda de generalização. Critérios de parada e escolha de checkpoint devem ser definidos antes de observar o conjunto de teste.

11.10 Ajuste de hiperparâmetros

Pesquisa em grade avalia combinações predeterminadas, mas cresce exponencialmente com o número de dimensões. Pesquisa aleatória costuma explorar melhor quando poucos hiperparâmetros dominam o resultado. Métodos bayesianos usam resultados anteriores para sugerir novas tentativas, sendo úteis quando cada treinamento é caro.

Comece por faixas logarítmicas para taxa de aprendizado e weight decay. Faça ensaios curtos, elimine configurações instáveis e só então invista em execuções longas. Compare sob o mesmo orçamento de tokens e registre configuração, código, semente e hardware.

11.11 Leis de escala como ferramenta de planejamento

Curvas de escala ajustam relações empíricas entre perda e recursos. Elas ajudam a estimar se o próximo incremento deve ir para dados, parâmetros ou passos, mas são válidas no regime medido. Mudanças de arquitetura, qualidade de dados e objetivo podem deslocar a curva. Extrapolação é uma hipótese quantitativa, não uma promessa de novas capaci-

dades.

Capítulo 12

Estratégias de Inferência e Geração de Texto

Desenvolvimento de métodos de geração de texto, explorando Greedy Search, Beam Search, amostragem Top-k, Top-p (Nucleus) e Temperatura.

Este capítulo detalha os mecanismos fundamentais pelos quais os Grandes Modelos de Linguagem (LLMs) transformam probabilidades estatísticas em texto coerente. A inferência em modelos autoregressivos envolve prever o próximo *token* (w_t) dado uma sequência de *tokens* anteriores ($w_{1:t-1}$).

Embora o modelo forneça uma distribuição de probabilidade sobre todo o vocabulário (os *logits*), a escolha de qual *token* selecionar — a **Estratégia de Decodificação** — determina a criatividade, coerência e qualidade do texto gerado.

12.1 1. O Processo Autoregressivo

Antes de explorar as estratégias, é crucial entender o fluxo de dados. O modelo recebe um contexto, processa através de camadas de *attention*, e a camada final projeta o estado oculto para o tamanho do vocabulário.

Uma função **Softmax** converte esses valores brutos (*logits*) em probabilidades.

$$P(w_t|w_{1:t-1}) = \text{softmax}(h_t)$$

A estratégia de decodificação intervém exatamente após a geração dos *logits* e antes (ou durante) a seleção final do *token*.

O processo continua até um token de fim, um limite de comprimento ou outro critério de parada. A janela de contexto limita quantos tokens podem participar do cálculo; quando prompt e geração ultrapassam esse limite, o sistema precisa truncar, resumir ou recuperar somente informações relevantes.

Em desempenho, separe **prefill** e **decode**. O prefill processa o prompt em paralelo e preenche o cache de chaves e valores. O decode gera tokens sequencialmente e reutiliza esse cache. Por isso, tempo para o primeiro token e tokens por segundo medem aspectos distintos da experiência.

12.2 2. Métodos Determinísticos

Métodos determinísticos sempre produzem a mesma saída para a mesma entrada, focando na maximização da probabilidade.

12.2.1 2.1 Greedy Search (Busca Gulosa)

A estratégia mais simples. Em cada passo, o modelo seleciona o *token* com a maior probabilidade absoluta.

- **Algoritmo:** $w_t = \text{argmax } P(w|w_{1:t-1})$
- **Vantagens:** Computacionalmente eficiente; bom para tarefas que exigem respostas exatas (aritmética, tradução curta).
- **Desvantagens:** Tende a gerar textos repetitivos e genéricos. Sofre de "miopia": escolhe o melhor *token* agora, mas pode levar a um "beco sem saída" probabilístico mais tarde.

```
# Exemplo simplificado de Greedy Search
import torch

def greedy_decode(model, input_ids, max_length):
    for _ in range(max_length):
        outputs = model(input_ids)
        next_token_logits = outputs.logits[:, -1, :]
        # Seleciona o índice com o maior valor
        next_token = torch.argmax(next_token_logits, dim=-1).un
        input_ids = torch.cat([input_ids, next_token], dim=1)
    return input_ids
```

12.2.2 2.2 Beam Search (Busca em Feixe)

O Beam Search mitiga a miopia do Greedy Search mantendo as k sequências mais prováveis (onde k é o `num_beams`) em cada passo, em vez de apenas uma.

- **Funcionamento:**

  1. Gera a distribuição para o próximo passo.
  2. Considera todas as extensões possíveis das k sequências atuais.
  3. Calcula a probabilidade acumulada (score) de cada caminho.
  4. Mantém apenas as top-k sequências para o próximo passo.

- **Vantagens:** Gera textos mais coerentes e gramaticalmente corretos que o Greedy.
- **Desvantagens:** Computacionalmente mais caro. Ainda pode sofrer de repetição em textos longos.

12.2.2.1 Diagrama de Fluxo: Greedy vs Beam Search

```
graph TD
    subgraph Greedy_Search
    A((Start)) --> B{Top 1?}
    B --> C[Token A: 0.6]
    C --> D{Top 1?}
    D --> E[Token B: 0.7]
    end

    subgraph Beam_Search_Width_2
    S((Start)) --> P1[Token A: 0.6]
    S --> P2[Token B: 0.3]
    P1 --> P1_1[Seq A-A: 0.4]
    P1 --> P1_2[Seq A-B: 0.5]
    P2 --> P2_1[Seq B-A: 0.2]
    P2 --> P2_2[Seq B-C: 0.6]

    P1_2 -.-> Keep1(Manter: Score 0.5)
    P2_2 -.-> Keep2(Manter: Score 0.6)
    P1_1 -.-> Discard1(Descartar)
    P2_1 -.-> Discard2(Descartar)
    end
```

12.3 3. Métodos Estocásticos (Amostragem)

Para geração de texto criativo (chatbots, histórias), a maximização da probabilidade é indesejável, pois a linguagem humana não é estritamente previsível. Métodos estocásticos introduzem aleatoriedade controlada.

12.3.1 3.1 Temperatura (Temperature)

A temperatura é um hiperparâmetro que escala os *logits* antes da aplicação da Softmax.

$$P_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

- $T < 1$ **(Baixa Temperatura):** Aumenta a diferença entre *logits* altos e baixos. A distribuição fica "pontiaguda". O modelo torna-se mais confiante e conservador.
- $T = 1$**:** Distribuição original do modelo.
- $T > 1$ **(Alta Temperatura):** Achata a distribuição. *Tokens* menos prováveis ganham chance de serem escolhidos. Aumenta a criatividade, mas também o risco de alucinação ou incoerência.

12.3.2 3.2 Top-k Sampling

Em vez de amostrar de todo o vocabulário (que pode ter 50.000+ *tokens*), limitamos a amostragem aos k *tokens* com maior probabilidade.

1. Ordena-se o vocabulário por probabilidade.
2. Mantém-se apenas os top-k *tokens*.
3. A massa de probabilidade restante é zerada.
4. A distribuição é re-normalizada.
5. Amostra-se um *token* dessa nova distribuição.

- **Problema:** k é fixo. Se a distribuição for plana (muitas palavras válidas), k pode cortar opções boas. Se for pontiaguda (poucas palavras válidas), k pode incluir palavras sem sentido.

12.3.3 3.3 Top-p (Nucleus) Sampling

Introduzido para resolver a rigidez do Top-k. Em vez de um número fixo de *tokens*, seleciona-se o menor conjunto de *tokens* cuja probabilidade

acumulada excede um limiar p (ex: 0.9 ou 90%).

- **Dinâmica:** O tamanho do conjunto de candidatos (k) varia dinamicamente a cada passo.
  - Se o modelo está incerto (distribuição plana), muitos *tokens* são necessários para somar 90%, aumentando a variedade.
  - Se o modelo está certo (distribuição pontiaguda), poucos *tokens* somam 90%, reduzindo o risco de erros.

12.3.3.1 Comparação Visual: Top-k vs Top-p

```
graph TD
    subgraph Vocabulario_Ordenado
    T1[Token 1: 0.4]
    T2[Token 2: 0.3]
    T3[Token 3: 0.15]
    T4[Token 4: 0.05]
    T5[Token 5: 0.05]
    T6[Token 6: 0.01]
    end

    subgraph Top_K_k_is_2
    K_Select[Seleciona Top 2] --> T1
    K_Select --> T2
    style T3 fill:#f9f9f9,stroke:#333,stroke-dasharray: 5 5
    end

    subgraph Top_P_p_is_0_85
    P_Calc[Soma Cumulativa >= 0.85]
    T1 --0.4--> Sum1(0.4)
    T2 --0.3--> Sum2(0.7)
```

```
    T3 --0.15--> Sum3(0.85 - STOP)

    P_Select[Seleciona Conjunto] --> T1
    P_Select --> T2
    P_Select --> T3
    end
```

12.4 4. Resumo Comparativo e Implementação

A maioria dos sistemas modernos utiliza uma combinação dessas estratégias (ex: Top-k seguido de Top-p, com Temperatura ajustada).

Estratégia	Determinístico?	Caso de Uso Ideal	Risco Principal
Greedy	Sim	Respostas curtas, factuais, matemática.	Repetição, falta de criatividade.
Beam Search	Sim	Tradução, resumo de texto.	Custo computacional, repetição.
Temperatura	Não (se $T \neq 0$)	Controle global de "risco".	T alto: Alucinação. T baixo: Repetição.
Top-k	Não	Chatbots simples.	Corte arbitrário de palavras válidas.

Estratégia	Determinístico?	Caso de Uso Ideal	Risco Principal
Top-p	Não	Escrita criativa, diálogo humano.	Imprevisibilidade em contextos técnicos.

12.4.1 Exemplo de Configuração de Geração (Hugging Face Transformers)

```
# Configuração típica para um Chatbot criativo mas coerente
generation_config = {
    "max_new_tokens": 200,
    "do_sample": True,          # Ativa métodos estocásticos
    "temperature": 0.7,         # Equilíbrio entre foco e criativ
    "top_k": 50,                # Limita a cauda longa de tokens
    "top_p": 0.95,              # Nucleus sampling para adaptação
    "repetition_penalty": 1.2 # Penaliza repetição de tokens já
}

# model.generate(**inputs, **generation_config)
```

12.5 5. Critérios de parada e avaliação

Uma geração precisa de condições explícitas: token de fim, limite de novos tokens e, em aplicações estruturadas, sequências de parada. O limite deve contar apenas os novos tokens quando o tamanho do prompt varia. Penalidades de repetição podem ajudar, mas valores altos também deformam nomes, código e termos que precisam reaparecer.

Não existe configuração universalmente melhor. Compare estratégias no conjunto representativo da aplicação e meça correção, diversidade, latência e taxa de respostas inválidas. Para geração factual, a verificação por fontes pode ser mais importante que qualquer ajuste de temperatura.

> **Experimento controlado**
>
> Fixe prompt e semente, varie um único parâmetro por vez e registre a distribuição antes e depois do filtro. Isso torna visível se a mudança veio da temperatura, de top-k ou de top-p.

Técnicas como *speculative decoding* usam um modelo menor para propor vários tokens e o modelo principal para verificá-los. Quando as propostas são aceitas com frequência, a latência diminui sem alterar a distribuição final produzida pelo modelo principal.

12.6 Cache de chaves e valores

Sem cache, ao gerar o token t, o modelo recalcularia chaves e valores dos $t-1$ tokens anteriores em todas as camadas. O **KV cache** guarda esses tensores. A nova consulta é comparada com as chaves armazenadas, e somente o novo par de chave e valor é anexado.

Para L camadas, H_{kv} cabeças KV, dimensão d_h, comprimento T e b bytes por elemento, uma estimativa para um item do lote é

$$\text{bytes} \approx 2LTH_{kv}d_hb,$$

em que o fator 2 representa chaves e valores. A fórmula explica por que contextos longos e lotes concorrentes pressionam a memória, e por que GQA reduz o cache ao diminuir H_{kv}.

O cache acelera o decode, mas não remove sua dependência sequencial. Também não deve ser reutilizado entre prompts incompatíveis: as

posições, máscaras e tokens precisam corresponder exatamente ao prefixo armazenado.

12.7 Quantização na inferência

Quantizar é representar pesos ou ativações com menos bits. Um peso em FP32 ocupa quatro bytes; em int8, aproximadamente um byte, antes de metadados de escala. Na quantização simétrica, um intervalo centrado em zero é mapeado para inteiros. Na assimétrica, um ponto zero permite representar intervalos deslocados.

PTQ quantiza um modelo já treinado usando calibração; **QAT** simula a quantização durante o treino para adaptar os pesos. Reduzir bits economiza memória e pode acelerar kernels compatíveis, mas introduz erro de arredondamento. Camadas sensíveis podem permanecer em maior precisão.

> **Avalie o sistema, não apenas o tamanho**
>
> Compare perplexidade ou tarefas, tokens por segundo, tempo para o primeiro token e memória. Um arquivo menor não garante aceleração se o hardware precisar desquantizar valores ou não possuir kernels adequados.

Capítulo 13

Fine-tuning e Personalização

13.1 O Que é Fine-tuning

Após o pré-treinamento, o modelo possui conhecimento geral sobre linguagem, mas pode não se comportar da forma desejada para tarefas específicas. O fine-tuning (ajuste fino) é o processo de continuar o treinamento do modelo em um conjunto de dados menor e mais específico, orientando seu comportamento para um domínio ou aplicação particular.

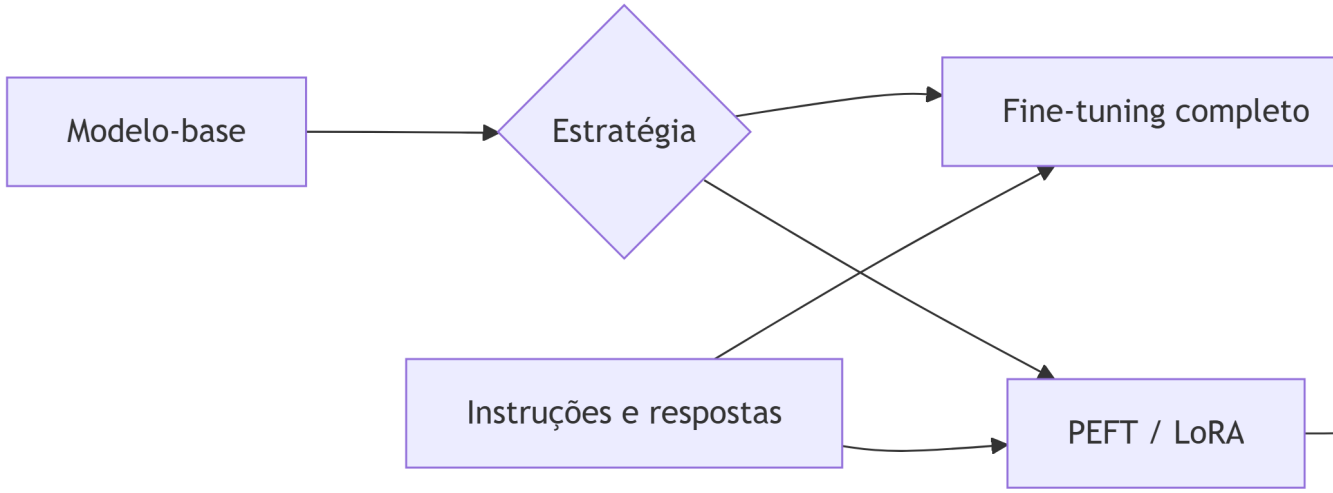
O fine-tuning é consideravelmente mais barato que o pré-treinamento, pois requer muito menos dados e computação. Enquanto o pré-treinamento pode levar semanas ou meses, o fine-tuning pode ser concluído em horas ou dias. Isso torna possível adaptar modelos grandes para aplicações específicas mesmo com recursos computacionais limitados.

13.2 Fine-tuning Supervisionado

A forma mais direta de fine-tuning é o ajuste fino supervisionado (SFT), onde o modelo é treinado em um conjunto de exemplos de entrada-saída desejados. Por exemplo, para criar um assistente de programação, poderíamos fine-tunar o modelo com pares de perguntas sobre programação e respostas corretas.

Durante o SFT, os parâmetros do modelo são atualizados para maximizar a probabilidade de gerar as respostas desejadas dado o prompt de entrada. O modelo aprende o formato e o estilo das respostas de exemplo, adquirindo novos comportamentos sem esquecer completamente o conhecimento do pré-treinamento.

No **instruction tuning**, os exemplos são organizados como instrução, contexto opcional e resposta. Essa mudança ensina o modelo a interpretar pedidos e produzir formatos úteis, em vez de apenas continuar texto bruto. A qualidade, diversidade e consistência das respostas supervisionadas são mais importantes que repetir muitas vezes exemplos quase idênticos.



13.3 Aprendizado por Reforço a Partir de Feedback Humano (RLHF)

Uma técnica particularmente influente é o aprendizado por reforço a partir de feedback humano (RLHF). O RLHF combina fine-tuning supervisionado com aprendizado por reforço para alinhar o modelo às preferências humanas.

O processo RLHF tipicamente envolve três etapas. Primeiro, modelos de recompensa são treinados com base em comparações humanas de pares de respostas. Segundo, o modelo de linguagem é otimizado usando o modelo de recompensa como função de objetivo através de algoritmos de aprendizado por reforço como PPO (*Proximal Policy Optimization*). Terceiro, o ciclo pode ser repetido para refinar continuamente as preferências.

O RLHF foi fundamental para criar modelos como o ChatGPT, que demonstram comportamento mais alinhado com as intenções do usuário e geram respostas mais úteis e menos problemáticas do que modelos apenas pré-treinados.

13.4 LoRA e Outras Técnicas de Eficiência

Antes de comparar métodos eficientes, é útil separar três objetivos. *Instruction tuning* ensina formato e comportamento a partir de pares de instrução e resposta; adaptação de domínio aproxima vocabulário e padrões de uma área; alinhamento por preferências favorece respostas comparativamente desejadas. Nenhum deles garante fatos corretos ou elimina a necessidade de avaliação.

O fine-tuning completo de modelos grandes é proibitivamente caro para muitos casos de uso, pois envolve atualizar bilhões de parâmetros. Técnicas de eficiência computacional foram desenvolvidas para resolver

esse problema.

LoRA (Low-Rank Adaptation) é uma técnica que adiciona pequenas matrizes treináveis às camadas do modelo, mantendo os parâmetros originais congelados. Isso reduz drasticamente o número de parâmetros que precisam ser otimizados durante o fine-tuning, mantendo resultados comparáveis ao fine-tuning completo.

Outras técnicas incluem QLoRA, que combina LoRA com quantização para reduzir ainda mais os requisitos de memória, e adapter methods, que introduzem módulos pequenos e treináveis em pontos estratégicos da arquitetura.

13.4.1 O que realmente muda no LoRA

Para uma matriz congelada $W \in \mathbb{R}^{d_{out} \times d_{in}}$, o LoRA aprende uma atualização de baixa ordem $\Delta W = BA$, com posto r muito menor que as dimensões originais. O forward passa a usar $Wx + BAx$. Assim, treinamos poucas variáveis sem descartar o conhecimento do modelo-base.

> ⚠ Evite vazamento de avaliação
>
> Separe treino, validação e teste por documento ou fonte quando exemplos semelhantes puderem se repetir. Uma divisão aleatória por linha pode colocar quase duplicatas nos três conjuntos e produzir uma estimativa excessivamente otimista.

13.5 Decidindo o que adaptar

Fine-tuning completo oferece liberdade máxima, mas exige memória para gradientes e estados do otimizador de todos os pesos. PEFT restringe a atualização a uma fração pequena. Antes de escolher, pergunte se o objetivo é ensinar um formato, incorporar terminologia, mudar preferências ou introduzir conhecimento factual atualizável.

Para o último caso, recuperação externa pode ser mais fácil de corrigir e auditar.

No LoRA, o posto r controla a capacidade da atualização. Valores muito pequenos podem limitar a adaptação; valores maiores consomem mais memória e podem facilitar sobreajuste. O fator de escala regula a contribuição de BAx, e o dropout no caminho LoRA pode ajudar em conjuntos pequenos.

Após o treino, os adaptadores podem permanecer separados ou ser fundidos a W. Mantê-los separados facilita trocar especializações; fundi-los simplifica a inferência. Em ambos os casos, o modelo-base exato e os módulos-alvo devem ser registrados.

13.6 Destilação de conhecimento

Na destilação, um modelo-aluno aprende com saídas de um modelo-professor. Em vez de receber apenas o token correto, o aluno pode aproximar toda a distribuição suavizada do professor. Relações entre alternativas — por exemplo, dois tokens quase igualmente plausíveis — carregam informação que um rótulo rígido descarta.

Uma função de objetivo pode combinar a perda supervisionada com uma divergência entre distribuições:

$$L = (1 - \lambda)L_{alvo} + \lambda T^2 D_{KL}(p_T^{professor} \| p_T^{aluno}),$$

onde T é a temperatura usada para suavizar os logits. A destilação pode produzir modelos menores e mais baratos, mas também transmite erros e vieses do professor. Dados de transferência, cobertura e avaliação independente continuam essenciais.

! Ajuste não substitui segurança

Comportamento desejado no conjunto de treino não garante robustez fora dele. Avalie instruções conflitantes, entradas longas, idiomas diferentes e tentativas de contornar políticas.

Capítulo 14

Avaliação, Checkpoints e Escala

Este capítulo apresenta métricas de linguagem, avaliação em múltiplas camadas e checkpoints reproduzíveis.

Após o treinamento, é preciso quantificar desempenho, comparar configurações de forma justa e preservar o estado necessário para retomar experimentos. Essas práticas conectam pesquisa, reprodutibilidade e operação.

14.1 1. Métricas de Avaliação: Perplexidade e Loss

Ao contrário de tarefas de classificação binária onde a "acurácia" é uma métrica clara, modelos generativos requerem métricas que avaliem a probabilidade das sequências geradas.

14.1.1 1.1. Cross-Entropy Loss (Perda de Entropia Cruzada)

A função de perda fundamental durante o treinamento é a *Cross-Entropy Loss*. Ela mede a diferença entre a distribuição de probabilidade prevista pelo modelo e a distribuição real (o próximo token correto).

14.1.2 1.2. Perplexidade (Perplexity - PPL)

A Perplexidade é a métrica padrão-ouro para avaliar modelos de linguagem autoregressivos. Intuitivamente, ela mede o quão "surpreso" o modelo fica ao ver novos dados. * **Definição Matemática:** A perplexidade é a exponenciação da perda de entropia cruzada média.

$$PPL = e^{Loss}$$

* **Interpretação:** * Uma PPL baixa indica que o modelo atribui alta probabilidade ao próximo token correto. * Uma PPL de 1 seria um modelo perfeito (certeza absoluta). * Uma PPL igual ao tamanho do vocabulário indica que o modelo está "chutando" aleatoriamente.

14.1.2.1 Exemplo de Implementação (PyTorch)

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

def calculate_perplexity(logits, targets):
    """
    Calcula a perplexidade dado os logits do modelo e os tokens

    Args:
```

```
        logits: Saída crua do modelo [batch_size, seq_len, voca
        targets: Tokens reais [batch_size, seq_len]

    Returns:
        float: Valor da perplexidade
    """
    # Redimensionar para compatibilidade com cross_entropy
    # [batch_size * seq_len, vocab_size]
    logits_flat = logits.view(-1, logits.size(-1))
    targets_flat = targets.view(-1)

    # Calcular Cross Entropy Loss
    loss = F.cross_entropy(logits_flat, targets_flat)

    # Calcular Perplexidade
    perplexity = torch.exp(loss)

    return perplexity.item()
```

14.1.3 Fluxo de Avaliação

```
graph LR
    A[Dataset de Validação] --> B(Modelo)
    B --> C{Logits}
    C --> D[Cálculo de Loss]
    D --> E[Cálculo de Perplexidade]
    E --> F[Métrica Final]
    style F fill:#90EE90,stroke:#333,stroke-width:2px
```

14.2 2. Salvamento de Checkpoints (Persistência)

O treinamento de LLMs é computacionalmente custoso e propenso a falhas de hardware. O salvamento de *checkpoints* não serve apenas para salvar o modelo final, mas para permitir a retomada do treinamento (*resuming*) e a análise de *overfitting*.

14.2.1 O que salvar?

Um checkpoint robusto deve conter mais do que apenas os pesos do modelo (`state_dict`).

1. **Pesos do Modelo:** Parâmetros aprendidos.
2. **Estado do Otimizador:** Momentos e variância (essencial para AdamW).
3. **Estado do Scheduler:** Learning rate atual.
4. **Metadados:** Número da época, step atual e valor da loss.

14.2.1.1 Estratégia de Salvamento

```
import os

def save_checkpoint(model, optimizer, scheduler, epoch, loss, p
    if not os.path.exists(path):
        os.makedirs(path)

    checkpoint = {
        'epoch': epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
```

```
        'scheduler_state_dict': scheduler.state_dict(),
        'loss': loss,
    }

    file_name = f"{path}/checkpoint_epoch_{epoch}.pt"
    torch.save(checkpoint, file_name)
    print(f"Checkpoint salvo em: {file_name}")

# Exemplo de uso no loop de treino
# save_checkpoint(model, optimizer, scheduler, current_epoch, c
```

14.3 Conclusão do Capítulo

Perplexidade e perda ajudam a acompanhar a modelagem de tokens, mas devem ser complementadas por tarefas, revisão humana e testes de sistema. Checkpoints preservam o progresso e tornam comparações e retomadas mais confiáveis.

14.4 Avaliação em múltiplas camadas

Perplexidade compara a previsão de tokens sob o mesmo tokenizer e corpus, mas não mede sozinha utilidade, segurança ou factualidade. Uma avaliação completa combina métricas intrínsecas, tarefas automáticas, revisão humana e testes do sistema integrado. Compare modelos com a mesma preparação de dados para não atribuir ao modelo uma diferença causada pela tokenização.

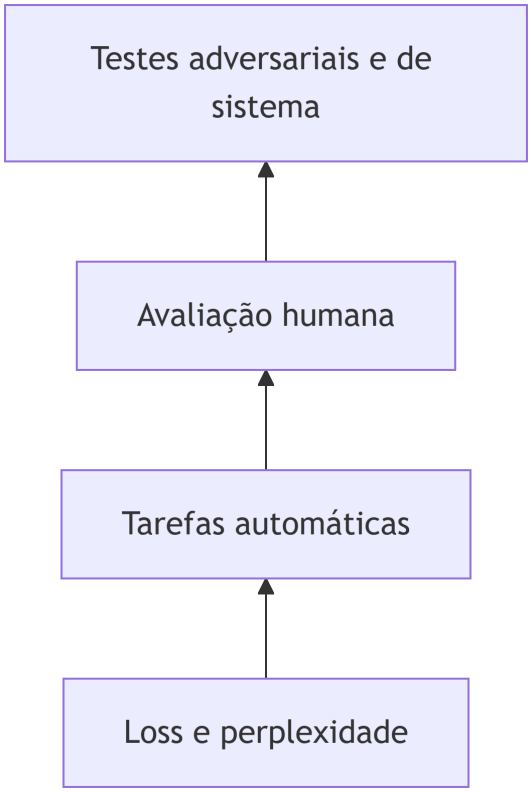
Separe um conjunto de teste antes de ajustar hiperparâmetros. Use validação durante o desenvolvimento e consulte o teste apenas nas decisões finais. Documente versão do checkpoint, prompt, parâmetros de

geração, dependências e semente; sem isso, uma pontuação isolada não é reproduzível.

> **!** Checkpoint não é apenas o modelo
>
> Para retomar treinamento de forma fiel, salve também estado do otimizador e scheduler, scaler de precisão mista, passo global e estados aleatórios. Para inferência, pesos, configuração e tokenizer compatível são o mínimo necessário.

14.5 Uma pirâmide de avaliação

Avaliações baratas e frequentes devem ficar na base; avaliações caras e contextualizadas, no topo. Durante o treino, loss e perplexidade detectam regressões rapidamente. Depois, tarefas automáticas medem capacidades específicas. Amostras humanas avaliam utilidade, clareza e critérios difíceis de codificar. Por fim, testes do sistema verificam recuperação, ferramentas, permissões, latência e falhas em produção.



Nenhuma camada substitui as demais. Uma perplexidade menor pode não melhorar instruções; uma preferência humana pode favorecer estilo sem detectar erro factual; um benchmark pode estar contaminado pelos dados de treino.

14.5.1 Incerteza e intervalos

Uma média sem variabilidade pode induzir conclusões frágeis. Para acurácia em n exemplos, use intervalos de confiança ou bootstrap. Ao comparar dois modelos, avalie-os nos mesmos itens e examine a distribuição das diferenças. Pequenos ganhos podem desaparecer com outra semente, prompt ou amostra.

14.5.2 Avaliação de geração aberta

Textos podem ter várias respostas válidas. Métricas baseadas em sobreposição lexical penalizam paráfrases corretas, enquanto avaliadores ba-

seados em modelos podem introduzir preferência por estilo, ordem ou verbosidade. Defina rubricas com critérios observáveis e calibre avaliadores automáticos contra julgamentos humanos.

Dimensão	Pergunta operacional
Correção	As afirmações podem ser verificadas?
Relevância	A resposta atende ao pedido sem desvios?
Robustez	Pequenas reformulações preservam o resultado?
Segurança	Entradas adversariais atravessam controles?
Eficiência	Latência e memória atendem ao orçamento?

14.6 Estratégia de checkpoints

Salvar a cada intervalo fixo simplifica a retomada, mas pode consumir armazenamento excessivo. Uma política prática conserva o melhor checkpoint de validação, o mais recente e alguns marcos espaçados. Grave primeiro em arquivo temporário e finalize com operação atômica para reduzir risco de corrupção.

> **Teste a restauração**
>
> Um checkpoint só é confiável depois de carregado. Automatize um teste que restaura o estado, executa um lote conhecido e confirma passo global, taxa de aprendizado e saída esperada dentro de tolerância.

Capítulo 15

Limitações, Ética e Uso Responsável

15.1 Alucinações e Fabricação de Informações

Uma das limitações mais significativas dos LLMs é o fenômeno chamado "alucinação", onde o modelo gera conteúdo que parece plausível mas é factual incorreto, inventado, ou não suportado pelos dados de treinamento.

As alucinações ocorrem porque o modelo é otimizado para gerar texto que parece coerente, não necessariamente para verificar fatos. Ele pode gerar detalhes específicos sobre eventos históricos, citações de pesquisas inexistentes, ou explicações técnicas incorretas com alta confiança, pois foram treinados para produzir texto fluente mesmo quando a precisão não pode ser verificada.

Mitigar alucinações é um campo ativo de pesquisa, envolvendo técnicas como retrieval-augmented generation (RAG), onde o modelo é combinado com bases de conhecimento externas, e técnicas de verificação que encorajam o modelo a expressar incerteza.

15.2 Viés e Discriminação

Os LLMs aprendem de dados humanos, e como tal, podem incorporar e amplificar vieses presentes nesses dados. Isso pode se manifestar como geração de texto estereotipado, discriminatório ou ofensivo em certos contextos.

O viés pode aparecer de múltiplas formas: viés de gênero em descrições de profissões, viés racial em detenções de nomes, viés cultural em piadas ou referências, e assim por diante. Mitigar esses vieses é challenging, pois diferentes intervenções podem ter efeitos colaterais inesperados e o conceito de "viés" é complexo e dependente de contexto.

15.3 Consistência e Coerência em Geração Longa

Embora LLMs sejam excelentes em gerar texto localmente coerente, manter coerência global em textos muito longos é desafiador. O modelo pode "esquecer" informações introduzidas no início do texto, contradizer afirmações anteriores, ou perder o fio condutor da narrativa à medida que gera mais conteúdo.

Esse problema é uma consequência direta das limitações de contexto e da natureza auto-regressiva da geração. Técnicas como chain-of-thought prompting (incentivar o modelo a pensar em voz alta), tree-of-thought reasoning, e arquiteturas com memória externa são exploradas como possíveis soluções.

15.4 Custo Computacional e Ambiental

O treinamento e inference de LLMs requer recursos computacionais massivos. Treinar um modelo de última geração pode custar dezenas

de milhões de dólares em recursos computacionais, sem mencionar o consumo energético significativo.

As implicações ambientais do treinamento de LLMs são objeto de preocupação crescente. A pegada de carbono do treinamento de modelos grandes pode ser substancial. Pesquisadores exploram técnicas para reduzir o custo computacional, como treinamento com eficiência energética e uso de hardware dedicado.

15.5 Limitações de Raciocínio

Uma limitação conhecida deve virar um requisito de engenharia. Se fatos podem ser inventados, conecte o modelo a fontes recuperáveis e exiba evidências. Se a saída será executada, aplique validação sintática, limites de permissão e confirmação humana. Se o domínio é sensível, mantenha trilhas de auditoria e uma alternativa segura para falhas.

Apesar de demonstrarem capacidades impressionantes, LLMs têm limitações fundamentais em tarefas que requerem raciocínio formal, matemática rigorosa, ou lógica dedutiva complexa. Eles podem produzir respostas numericamente incorretas, falhar em perceber contradições lógicas, ou ser facilmente enganados por alterações sutis em problemas de raciocínio.

A comunidade de pesquisa debate se LLMs são capazes de "raciocinar" genuinamente ou se estão simplesmente recuperando e recombinando padrões observados durante o treinamento. Essa questão permanece em aberto e é fundamental para compreender tanto as capacidades quanto os limites desses modelos.

15.6 Avaliação adversarial

Além de medir a tarefa média, teste entradas ambíguas, negações, textos longos, idiomas diferentes, dados fora de distribuição e tentativas de in-

jeção de prompt. Um único número agregado pode ocultar falhas graves em subgrupos.

> **!** Calibração de confiança
>
> O tom seguro de uma resposta não é uma medida de probabilidade nem de veracidade. Interfaces responsáveis devem comunicar incerteza com base em verificações externas, não apenas pedir ao próprio modelo que declare confiança.

15.7 Privacidade e dados pessoais

Corpora extensos podem conter informações pessoais, segredos ou conteúdo sem autorização adequada. Além da curadoria antes do treinamento, sistemas em produção precisam limitar coleta, retenção e acesso aos prompts. Logs úteis para observabilidade também são uma superfície de risco e devem seguir políticas explícitas de minimização e descarte.

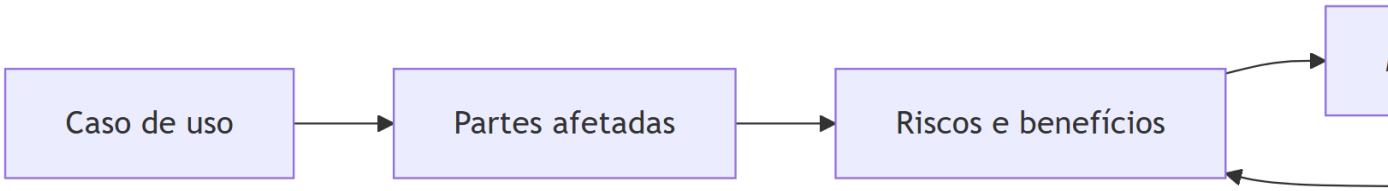
15.8 Desinformação e uso malicioso

A geração barata e em escala pode apoiar fraude, manipulação e conteúdo enganoso. Mitigações incluem autenticação, limites de uso, detecção de abuso, rastreabilidade e revisão humana em operações de alto impacto. Filtros isolados não substituem uma análise do fluxo completo em que o modelo está inserido.

15.9 Trabalho, responsabilidade e prestação de contas

Automação altera tarefas e distribui benefícios e custos de forma desigual. Uma implantação responsável define quem responde por erros, como uma decisão pode ser contestada e quando uma pessoa deve assumir o controle. O fornecedor do modelo, a organização integradora e o operador possuem responsabilidades diferentes, que devem ser documentadas.

15.10 Processo de engenharia responsável



Área	Ação mínima
Dados	Remover ou proteger informações pessoais e documentar proveniência.
Avaliação	Testar qualidade, vieses, abuso e grupos relevantes ao domínio.
Operação	Aplicar permissões mínimas, monitoramento e resposta a incidentes.
Transparência	Publicar limitações, usos previstos e responsáveis pelo sistema.

O debate sobre se modelos “entendem” linguagem não deve substituir testes de comportamento. Independentemente da interpretação fi-

losófica, decisões de implantação precisam se apoiar em capacidades e falhas observáveis.

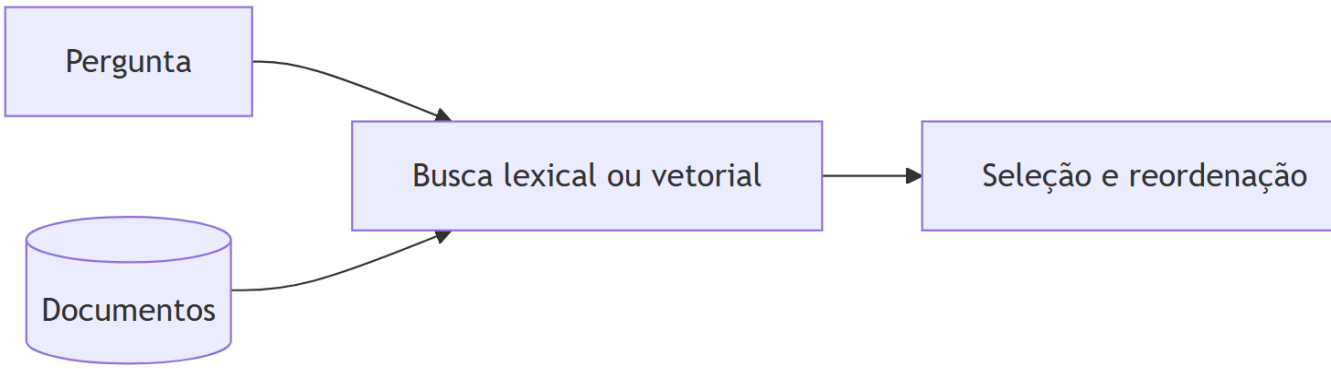
15.11 Por que alucinações acontecem

O objetivo autoregressivo recompensa continuações prováveis, não uma consulta explícita a fatos. Quando o contexto é insuficiente, contraditório ou distante dos dados de treinamento, o modelo ainda precisa distribuir probabilidade sobre o vocabulário. Estratégias de amostragem mais livres podem ampliar a variedade, mas a causa estrutural permanece: fluência não implica conexão com evidências.

Mitigações ocupam níveis diferentes. Prompts podem exigir limites e incerteza; recuperação fornece documentos; ferramentas consultam sistemas atualizados; fine-tuning ajusta comportamento; validação externa verifica formatos ou fatos. Nenhuma técnica elimina o problema em todos os domínios.

15.12 RAG como sistema, não como prompt longo

Na geração aumentada por recuperação (**RAG**), uma consulta é transformada em representação de busca, documentos relevantes são recuperados e trechos selecionados entram no contexto do gerador.



Falhas podem ocorrer em cada etapa: o documento correto pode não existir, a busca pode não recuperá-lo, o recorte pode remover contexto ou o modelo pode ignorar a evidência. Avalie recuperação e geração separadamente. Citações devem apontar para trechos realmente usados e ser conferidas, não apenas formatadas.

15.13 Injeção de prompt e fronteiras de confiança

Uma injeção ocorre quando conteúdo não confiável tenta se passar por instrução. Isso é especialmente perigoso em sistemas que leem páginas, e-mails ou documentos e depois acionam ferramentas. Separar mensagens por função ajuda, mas não torna dados externos confiáveis.

Trate a saída do modelo como entrada não confiável. Ferramentas devem validar argumentos, aplicar listas de operações permitidas, limitar escopo e pedir confirmação para ações sensíveis. Segredos não devem ser colocados no contexto se o modelo não precisa vê-los.

> ⚠ **Filtros não criam uma fronteira de segurança**
>
> Um classificador de entrada ou saída reduz alguns abusos, mas pode ser contornado. A proteção principal vem de permissões mínimas, isolamento, validação determinística e auditoria.

15.14 Explicabilidade com limites

Explicações globais procuram descrever tendências do modelo; explicações locais tratam uma entrada específica. Métodos por gradiente, ablação, sondas e análise de ativações revelam aspectos diferentes. Mapas de atenção, por exemplo, mostram pesos de mistura, mas não demonstram causalidade completa.

Uma explicação deve ser testada: remover a característica apontada muda a saída? O método é estável diante de pequenas perturbações? A narrativa continua válida em outros grupos? Explicações plausíveis, porém não fiéis, podem aumentar confiança justamente quando deveriam provocar cautela.

Capítulo 16

O Futuro dos LLMs

16.1 Modelos Multimodais

A tendência mais clara no desenvolvimento de LLMs é a expansão para além do texto. Modelos multimodais processam e geram não apenas texto, mas também imagens, áudio, vídeo, e outras modalidades de dados.

Modelos como GPT-4V, Gemini, e Claude 3 já demonstram capacidades impressionantes de entender imagens, responder perguntas sobre gráficos, e integrar informações visuais com texto. Essa convergência de modalities abre aplicações em áreas como medicina (análise de imagens diagnósticas), educação (assistentes visuais), e acessibilidade (descrição de conteúdo visual).

16.2 Raciocínio e Planejamento

Melhorar as capacidades de raciocínio é uma prioridade central na pesquisa atual. Técnicas como chain-of-thought prompting, tree-of-thought reasoning, e métodos neuro-symbolic que combinam redes neurais com raciocínio lógico formal estão sendo explorados ativamente.

O objetivo não é apenas melhorar o raciocínio matemático ou lógico,

mas também o raciocínio de senso comum, a capacidade de fazer inferências sobre o mundo físico e social, e o planejamento multi-step para alcançar objetivos complexos.

16.3 Eficiência e Acessibilidade

A pressão para desenvolver modelos mais eficientes é intensa. Técnicas como quantização, pruning (podagem de parâmetros menos importantes), knowledge distillation (treinar modelos menores para imitar modelos maiores), e arquiteturas mais eficientes são ativamente pesquisadas.

A democratização do acesso a LLMs é facilitada por modelos open-source como Llama, Mistral, e Gemma, que permitem que organizações com recursos limitados implantem modelos de linguagem capazes. Essa abertura acelera a inovação e reduz a concentração de poder em poucas organizações.

16.4 Integração com Ferramentas e Agentes

A próxima fronteira é a criação de agentes de IA que não apenas geram texto, mas podem usar ferramentas, executar código, navegar na web, e executar ações no mundo real. Modelos treinados para usar ferramentas através de técnicas como toolformer demonstram que LLMs podem aprender a interagir com o ambiente de forma instrumental.

Esses agentes de IA raise questões fascinantes sobre autonomia, controle, e segurança. Como garantir que um agente de IA que pode executar ações no mundo real faça isso de forma segura e alinhada com as intenções humanas?

16.5 Compreensão e Explicabilidade

Melhorar nossa compreensão de como LLMs funcionam internamente é crucial para desenvolver sistemas mais seguros e confiáveis. Pesquisadores utilizam técnicas de análise para mapear quais recursos e conhecimentos estão representados em diferentes partes do modelo.

O campo de interpretabilidade (interpretability) busca desenvolver métodos para explicar as decisões e comportamentos dos modelos. Embora ainda esteja em estágios iniciais, avanços na interpretabilidade podem permitir detecção mais temprana de vieses, compreensão de falhas, e desenvolvimento de sistemas mais robustos.

16.6 Arquiteturas Alternativas

Ao ler sobre uma alternativa, separe demonstração de laboratório, produto disponível e adoção em escala. Uma técnica pode mostrar boa qualidade em um benchmark e ainda enfrentar custo, latência, segurança ou integração que impedem seu uso real. Da mesma forma, um anúncio não substitui resultados reproduzíveis e comparação com baselines fortes.

Para um estudante de Ciência da Computação, a preparação mais durável não é memorizar nomes de modelos. É dominar álgebra linear, probabilidade, otimização, estruturas de dados, sistemas distribuídos e avaliação experimental. Essas ferramentas permitem compreender novas arquiteturas mesmo quando o vocabulário da área muda.

Perguntas para leitura crítica

Qual problema a proposta resolve? Com qual custo? Contra quais baselines? Os dados e métricas medem a aplicação real? Os resultados podem ser reproduzidos? Essas perguntas são mais informativas que a simples contagem de parâmetros.

Enquanto o Transformer domina o campo atual, a busca por arquiteturas alternativas continua. Arquiteturas baseadas em modelos de espaço de estados (*state-space models*), como Mamba, investigam formas mais eficientes de processar sequências longas.

A possibilidade de que uma nova arquitetura transforme o campo novamente não deve ser descartada. A história da IA contém vários exemplos em que paradigmas dominantes foram substituídos por abordagens diferentes.

16.7 Multimodalidade: mais que concatenar vetores

Modelos multimodais precisam resolver representação, alinhamento e fusão. Um codificador visual pode transformar regiões de uma imagem em tokens; uma projeção leva esses vetores à dimensão usada pelo modelo de linguagem; atenção cruzada ou concatenação permite combinar sinais. O treinamento deve associar elementos correspondentes, como regiões e palavras de uma legenda.

Os desafios incluem escalas temporais diferentes, dados ausentes e conflitos entre modalidades. Áudio possui sequência densa, imagens organizam relações espaciais e texto é discreto. Um modelo pode explorar atalhos — responder pela legenda sem observar a imagem —, portanto avaliações devem controlar quais modalidades contêm a evidência.

16.8 Modelos de espaço de estados

Um modelo de espaço de estados mantém uma memória h_t atualizada a cada entrada. Em forma discreta simplificada,

$$h_t = \bar{A}h_{t-1} + \bar{B}x_t, \qquad y_t = Ch_t.$$

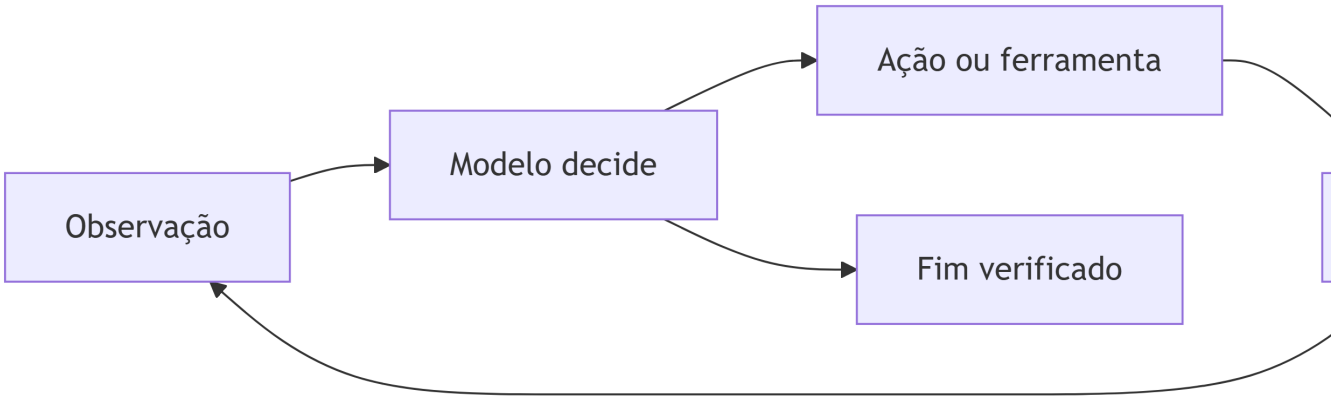
Estruturas modernas tornam parte desses parâmetros dependente da entrada, permitindo selecionar o que conservar ou esquecer. O cálculo pode ser expresso como recorrência eficiente na inferência e como operação paralelizável no treinamento. A principal promessa é processar sequências longas sem materializar atenção quadrática, mas recuperação precisa e integração com hardware continuam critérios centrais.

16.9 Modelos menores e especialização

Modelos pequenos podem ser suficientes quando o domínio é estreito, a latência é crítica ou os dados não podem sair do dispositivo. Destilação, quantização, dados sintéticos cuidadosamente filtrados e arquiteturas eficientes reduzem custo. A comparação justa deve incluir qualidade na tarefa, consumo, privacidade e facilidade de atualização — não apenas tamanho.

16.10 Agentes e ferramentas

Um agente combina um modelo com estado, ferramentas e um laço de decisão. O modelo propõe uma ação; o ambiente devolve uma observação; o processo continua até uma condição de término. Essa composição amplia capacidade, mas também acumula probabilidade de erro a cada etapa.



Agentes confiáveis precisam de limites de passos, orçamento, idempotência, registro e confirmação para efeitos externos. Planejamento verbal não substitui verificação do estado real.

16.11 Computação em tempo de teste

Em vez de produzir uma única resposta imediatamente, um sistema pode gerar candidatos, verificar restrições, comparar soluções ou refinar uma tentativa. Esse uso adicional de computação pode melhorar tarefas com critérios verificáveis, como código e matemática. O ganho depende da qualidade do verificador e da diversidade das propostas; repetir respostas semelhantes apenas aumenta custo.

O futuro é um espaço de compromissos

Atenção densa, SSMs, MoE, modelos pequenos e agentes resolvem gargalos diferentes. A questão útil não é qual tecnologia vencerá, mas qual combinação atende qualidade, custo, segurança e manutenção de uma aplicação concreta.

Capítulo 17

Glossário

Agent: Sistema de IA que pode executar ações autonomamente, frequentemente utilizando LLMs como cérebro.

Alignment (Alinhamento): O processo de garantir que os comportamentos de um modelo de IA estejam alinhados com as intenções e valores humanos.

Attention Mechanism (Mecanismo de Atenção): Técnica que permite ao modelo prestar atenção seletivamente a diferentes partes da entrada ao processar cada elemento.

BERT: Modelo baseado em codificador, desenvolvido para aprender representações bidirecionais e aplicá-las a tarefas de compreensão de texto.

Chain-of-Thought (Cadeia de Pensamento): Técnica de prompting que incentiva o modelo a mostrar seu raciocínio intermediário.

Context Window (Janela de Contexto): Quantidade máxima de texto que o modelo pode processar simultaneamente.

Decoder: Componente do Transformer que gera sequências token por token.

Embedding: Representação vetorial densa de tokens que captura relações semânticas.

Fine-tuning: Ajuste fino de um modelo pré-treinado para um domínio ou tarefa específica.

Flash Attention: Técnica para acelerar o cálculo de atenção reduzindo a memória necessária.

Generative AI (IA Generativa): Sistemas de IA capazes de criar conteúdo novo, incluindo texto, imagens, áudio e código.

Gradient: Vetor de derivadas parciais que indica a direção de maior aumento de uma função de perda.

Hallucination (Alucinação): Quando um modelo gera informação incorreta ou fabricada com aparente confiança.

Inference: Processo de usar um modelo treinado para fazer previsões ou gerar conteúdo.

In-Context Learning: Capacidade de um modelo de aprender de exemplos fornecidos no prompt sem ajustar pesos.

LLM (Large Language Model): Modelo de linguagem de grande escala treinado em vastas quantidades de texto.

LoRA (Low-Rank Adaptation): Técnica de fine-tuning eficiente que adiciona adaptadores de baixo posto.

Multimodal: Capacidade de processar múltiplas modalidades de dados, como texto e imagens.

NLP (Natural Language Processing): Campo da IA focado em processamento e compreensão de linguagem natural.

Parameter (Parâmetro): Valor ajustável em uma rede neural que é aprendido durante o treinamento.

Prompt: Entrada de texto fornecida ao modelo para elicitar uma resposta.

Quantization: Técnica para reduzir a precisão numérica dos pesos do modelo, diminuindo requisitos de memória.

Retrieval-Augmented Generation (RAG): Abordagem que combina LLMs com busca em bases de conhecimento externas.

RLHF (Reinforcement Learning from Human Feedback): Técnica de alinhamento usando feedback humano.

Self-Attention (Auto-Atenção): Mecanismo de atenção onde queries, keys e values vêm da mesma sequência.

Temperature: Parâmetro que controla a aleatoriedade da geração de texto.

Token: Unidade básica de texto processada pelo modelo.

Tokenization: Processo de converter texto em sequência de tokens.

Top-k/Top-p Sampling: Técnicas para limitar a aleatoriedade na seleção de tokens.

Transformer: Arquitetura neural baseada em mecanismos de atenção que revolucionou o NLP.

Vanilla Transformers: Versão original do Transformer com codificador e decodificador.

Backpropagation (Retropropagação): Algoritmo que aplica a regra da cadeia para calcular o gradiente da perda em relação aos parâmetros.

Checkpoint: Registro dos pesos e, quando necessário, dos estados que permitem retomar um treinamento.

Cross-Entropy (Entropia Cruzada): Função de perda que penaliza baixa probabilidade atribuída ao token-alvo.

Gradient (Gradiente): Vetor de derivadas que indica como uma pequena mudança em cada parâmetro altera a perda.

KV Cache: Cache das chaves e valores de atenção do prefixo, reutilizado para acelerar a geração autoregressiva.

Logit: Valor real não normalizado produzido antes da softmax; diferenças entre logits determinam razões de probabilidade.

LoRA (Low-Rank Adaptation): Técnica de ajuste eficiente que aprende pequenas atualizações matriciais de baixa ordem e mantém o modelo-base congelado.

Perplexity (Perplexidade): Exponencial da entropia cruzada média; sob condições comparáveis, valores menores indicam menor surpresa diante dos tokens observados.

Pre-norm: Organização de bloco que normaliza a entrada antes da atenção ou da rede feed-forward.

Softmax: Função que converte logits em valores positivos cuja soma é 1.

Weight Tying (Compartilhamento de Pesos): Uso da mesma matriz nos embeddings de entrada e na projeção de saída para reduzir parâmetros.

FlashAttention: Implementação exata da atenção que organiza o cálculo em blocos para reduzir transferências de memória e intermediários materializados.

GQA (Grouped-Query Attention): Variante em que grupos de cabeças de consulta compartilham chaves e valores, reduzindo o KV cache.

In-context Learning (Aprendizagem em Contexto): Adaptação temporária induzida por instruções ou exemplos no prompt, sem atualização dos parâmetros.

Mixture of Experts (MoE): Camada com múltiplas redes especialistas e um roteador que ativa apenas parte delas para cada token.

MQA (Multi-Query Attention): Variante que compartilha um único conjunto de chaves e valores entre todas as cabeças de consulta.

Prompt Injection (Injeção de Prompt): Entrada não confiável que tenta alterar instruções ou induzir ações fora da finalidade autorizada.

Quantization (Quantização): Representação de pesos ou ativações com menos bits para reduzir memória e, em hardware compatível, acelerar cálculos.

RAG (Retrieval-Augmented Generation): Arquitetura que recupera evidências externas e as fornece ao modelo durante a geração.

RoPE (Rotary Positional Embedding): Codificação que gira pares de coordenadas de consultas e chaves conforme a posição, introduzindo relações relativas na atenção.

SFT (Supervised Fine-Tuning): Continuação supervisionada do treinamento com pares de entrada e resposta desejada.

Speculative Decoding: Geração em que um modelo auxiliar propõe tokens e o modelo principal verifica as propostas para reduzir latência.

State-Space Model (Modelo de Espaço de Estados): Arquitetura sequencial que mantém um estado atualizado por entradas sucessivas e pode oferecer custo linear no comprimento.

SwiGLU: Rede feed-forward com uma ativação SiLU que funciona como porta sobre outra projeção linear.