IA Generativa

Python e C++

Cesar de Oliveira F. Silva

25 июня 2026 г.

Roteiro

Objetivos da aula

- Entender o ciclo completo de criação de um LLM.
- Construir intuição sobre arquitetura, dados e otimização.
- Diferenciar pré-treinamento, ajuste fino, alinhamento e inferência.
- Relacionar escolhas técnicas com custo, qualidade e latência.
- Identificar gargalos de dados, memória e serving.
- Sair com uma visão de implementação em Python e aceleração em C++.

Resultado esperado

Ao final, você deve conseguir explicar como um LLM é treinado, avaliado, otimizado e colocado em produção.

Agenda

1. O que é um LLM e o que ele não é.
2. Pipeline de dados, limpeza e tokenização.
3. Embeddings, atenção e Transformer.
4. Pré-treinamento e pós-treinamento.
5. Avaliação, segurança e alinhamento.
6. Inferência, quantização e KV cache.
7. Implementação em Python.
8. Aceleração e serving em C++.

Leitura da aula

A sequência segue o fluxo real de um sistema: dados $\rightarrow$ modelo $\rightarrow$ avaliação $\rightarrow$ produção.

Etapas da construção de um LLM

Internet, Livros, Código, Conversas

↓

Dados

↓

Tokenização

↓

Embeddings + Transformer

↓

Alinhamento (SFT, RLHF, DPO)

↓

Inferência

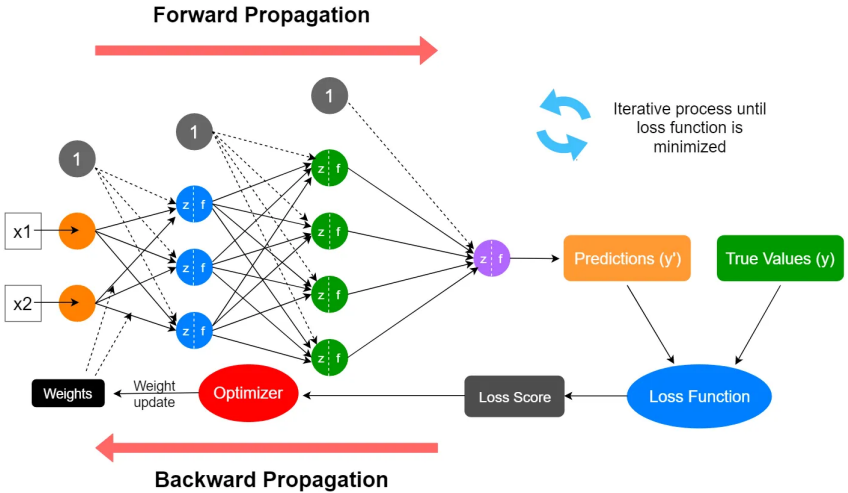
↓

Produto

↓

Usuário

Introdução


Forward Propagation
1
1
1
Iterative process until loss function is minimized
x1
x2
z f
Predictions (y')
True Values (y)
Weights
Weight update
Optimizer
Loss Score
Loss Function
Backward Propagation

Definição operacional de LLM

Um LLM é um modelo de linguagem treinado para modelar a distribuição do próximo token dado um contexto.

Ideia central

Aprender $\mathrm{P}(\mathrm{x_t} \mid \mathrm{x_{<t}})$ em corpora grandes, diversos e ruidosos, normalmente por otimização autoregressiva.

- Entrada: sequência de tokens.
- Saída: distribuição sobre o próximo token.
- Escala: parâmetros, dados e compute.
- Uso: geração, classificação, extração e diálogo.

- Capta padrões sintáticos, semânticos e procedurais.
- Pode ser adaptado para instruções, ferramentas e tarefas específicas.
- O desempenho final depende de dados, arquitetura, treino e alinhamento.

O que um LLM faz bem

Capacidades típicas

- Completar texto coerente.
- Resumir, traduzir e reescrever.
- Extrair informação e classificar texto.
- Gerar código e explicar trechos.

Por que isso acontece

- Aprende regularidades estatísticas de alto nível.
- Comprime padrões de linguagem em representações latentes.
- Generaliza por similaridade sem depender de regras explícitas.

- Funciona bem como componente de sistemas maiores, como RAG, copilotos e agentes.
- Em tarefas estruturadas, o ganho aparece mais em produtividade do que em “inteligência geral”.

Limitações estruturais

Falhas comuns

- Não possui conhecimento garantido sobre fatos atuais.
- Pode alucinar respostas plausíveis e erradas.
- É sensível ao prompt, ao contexto e à ordem das instruções.

Consequência prática

- Sempre validar em tarefas críticas.
- Usar recuperação externa quando a precisão factual importa.
- Tratar a saída como hipótese, não como verdade.

Risco operacional

O modelo pode soar confiante mesmo quando está incorreto. Em produção, isso exige checagem, filtros e monitoramento.

Pipeline de dados

1. Coleta de fontes.
2. Filtragem e deduplicação.
3. Normalização e limpeza.
4. Tokenização.
5. Empacotamento em sequências.
6. Mistura de batches e shard por dispositivo.

Pontos de controle

- Qualidade do conteúdo.
- Cobertura de domínios e idiomas.
- Privacidade e licenciamento.
- Vazamento entre treino e avaliação.
- Eficiência de batch e throughput.

Mensagem principal

Em LLMs, qualidade de dados costuma render mais retorno do que aumentar parâmetros sem critério.

Fontes de dados

Fontes típicas

- Web crawl.
- Livros e artigos.
- Código-fonte.
- Conversas e instruções.

Dados complementares

- Rótulos humanos para instrução.
- Preferências para alinhamento.
- Dados sintéticos gerados por outros modelos.
- Exemplos especializados do domínio-alvo.

- Misturar fontes melhora cobertura, mas aumenta o risco de ruído e desbalanceamento.
- O mix final deve refletir o uso real do modelo.

Problemas comuns nos dados

- Ruído, spam e páginas duplicadas.
- Conteúdo tóxico ou ofensivo.
- Texto mal formatado ou quebrado.
- Idioma incorreto ou misturado.
- Vazamento de benchmark.
- Conteúdo desatualizado ou contraditório.
- Texto curto demais para contexto útil.
- Distribuição desbalanceada entre domínios.

Impacto

Dados ruins degradam tanto a qualidade da resposta quanto a estabilidade do treinamento.

Limpeza e filtragem

Operações básicas

- Remover HTML, menus e boilerplate.
- Normalizar Unicode, pontuação e espaços.
- Padronizar quebras de linha e codificação.
- Remover conteúdo extremamente curto ou repetitivo.

Filtros mais fortes

- Classificação de idioma.
- Scoring de qualidade.
- Regras de segurança e privacidade.
- Deduplicação por hash e similaridade.

- O objetivo não é apenas limpar texto, mas preservar sinal útil com o menor ruído possível.

Checklist de qualidade do dataset

- O texto representa o domínio-alvo.
- Há diversidade suficiente de fontes.
- Existe rastreabilidade de origem.
- Há separação clara entre treino, validação e teste.
- Há regras explícitas de exclusão.
- O volume por idioma está equilibrado.
- O conteúdo sensível foi removido.
- O dataset é reproduzível e versionado.

Deduplicação e leakage

Por que importa

Se o benchmark aparece no treino, a métrica final fica artificialmente inflada.

- A deduplicação reduz memorization.
- O leakage distorce avaliação e seleção de modelos.

Estratégias práticas

- Hash exato para duplicatas óbvias.
- MinHash ou simhash para near-duplicates.
- Split por documento, não por linha.
- Remoção de exemplos quase idênticos aos benchmarks.

Glossário de siglas I: Arquitetura e treinamento

Sigla	Significado
LLM	Large Language Model
NLP	Natural Language Processing
BPE	Byte Pair Encoding
FFN	Feed Forward Network
MLP	Multi-Layer Perceptron
RoPE	Rotary Positional Embedding
ALiBi	Attention with Linear Biases
MoE	Mixture of Experts
KV	Key-Value Cache

Arquitetura

Essas siglas aparecem principalmente na tokenização, embeddings e Transformer.

Glossário de siglas II: Alinhamento

Sigla	Significado
SFT	Supervised Fine-Tuning
RLHF	Reinforcement Learning from Human Feedback
DPO	Direct Preference Optimization
PPO	Proximal Policy Optimization
RM	Reward Model
CoT	Chain of Thought
RLAIF	Reinforcement Learning from AI Feedback

Alinhamento

Essas técnicas transformam um modelo pré-treinado em um assistente útil e seguro.

Glossário de siglas III: Inferência

Sigla	Significado
RAG	Retrieval-Augmented Generation
API	Application Programming Interface
EOS	End Of Sequence
TPS	Tokens Per Second
TTFT	Time To First Token
QPS	Queries Per Second
GPU	Graphics Processing Unit
CPU	Central Processing Unit

Produção

São métricas e componentes frequentemente usados em sistemas reais.

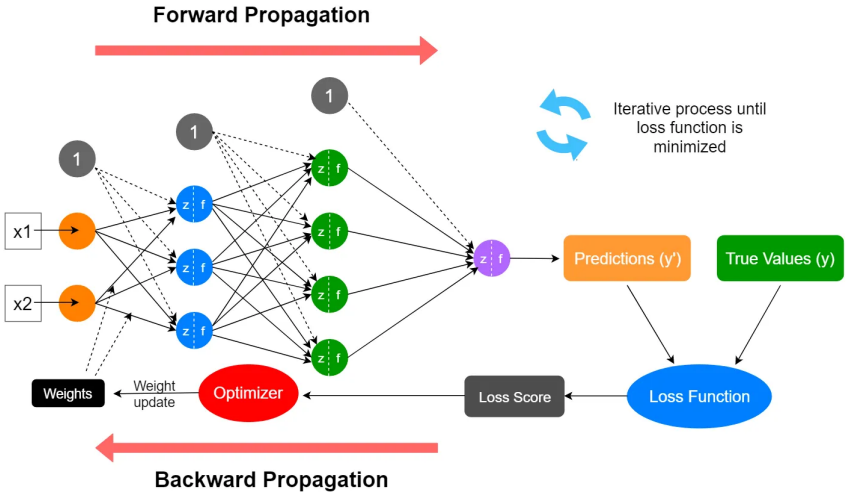
Glossário de siglas IV: Infraestrutura

Sigla	Significado
CUDA	Compute Unified Device Architecture
DDP	Distributed Data Parallel
FSDP	Fully Sharded Data Parallel
GEMM	General Matrix Multiplication
SIMD	Single Instruction Multiple Data
ONNX	Open Neural Network Exchange
GGUF	GPT-Generated Unified Format
RPC	Remote Procedure Call

Engenharia

Essas siglas aparecem em treinamento distribuído, otimização e serving.

Tokenização



Por que tokenizar

- Redes neurais operam sobre números inteiros, não sobre texto bruto.
- Tokenização converte linguagem natural em uma sequência discreta.
- Cada token será posteriormente convertido em um embedding.
- O custo computacional do modelo depende diretamente do número de tokens.

Pipeline

Texto → Tokens → IDs → Embeddings

Exemplo

“Transformers são úteis”
↓
[Transform, ers, são, úteis]

Observação

O mesmo texto pode gerar quantidades muito diferentes de tokens dependendo do tokenizador utilizado.

Por que a tokenização é tão importante?

- Determina o tamanho efetivo do contexto.
- Afeta diretamente memória, latência e custo de treinamento.
- Influencia desempenho em linguagens diferentes.
- Impacta tarefas envolvendo código, matemática e símbolos.
- Define o tamanho da matriz de embeddings e da camada de saída.

Exemplo

Uma janela de 8.000 tokens não significa 8.000 palavras.
Em inglês, frequentemente corresponde a 5.000–6.000 palavras. Em código-fonte ou português, a relação pode ser diferente.

Tipos de tokenização

Caractere

- Menor unidade possível.
- Vocabulário pequeno.
- Sequências muito longas.

Palavra

- Um token por palavra.
- Fácil de interpretar.
- Problemas com palavras desconhecidas.

Subword

- BPE.
- WordPiece.
- Unigram.
- Dominante em LLMs modernos.

Byte-level

- Opera diretamente sobre bytes.
- Cobertura universal.
- Muito usado em modelos GPT.

Comparação entre estratégias

Método	Vocabulário	Comprimento	Robustez
Caractere	Muito pequeno	Muito alto	Alta
Palavra	Muito grande	Baixo	Baixa
Subword	Médio	Médio	Alta
Byte-level	Médio	Médio/alto	Muito alta

- Tokenização por palavra praticamente desapareceu dos LLMs modernos.
- Subword tornou-se o padrão por equilibrar eficiência e generalização.
- Byte-level simplifica o tratamento de idiomas raros e código.

O problema das palavras desconhecidas

Tokenização por palavra

- computador
- programação
- inteligência

Nova palavra:

bioinformaticamente

Pode não existir no vocabulário.

Tokenização subword

bio + informa + tica + mente

- Reaproveita partes já aprendidas.
- Generaliza para palavras nunca vistas.
- Reduz problema de OOV (Out Of Vocabulary).

BPE: Byte Pair Encoding

Ideia principal

Construir o vocabulário unindo iterativamente pares de símbolos que aparecem com maior frequência.

1. Começar com caracteres individuais.
2. Contar pares adjacentes.
3. Encontrar o par mais frequente.
4. Mesclar esse par.
5. Repetir até atingir o tamanho desejado do vocabulário.

- Simples.
- Escalável.
- Base de diversos tokenizadores modernos.

Exemplo conceitual de BPE

Considere as palavras:

low, lower, lowest, slow

Inicialmente:

l o w l o w e r s l o w

Pares frequentes:

- (l,o)
- (o,w)

Após sucessivas fusões:

low, er, est, slow

Resultado

Partes frequentes tornam-se tokens reutilizáveis em milhares de palavras.

WordPiece e Unigram

WordPiece

Usado originalmente pelo BERT.
Seleciona novos tokens com base em ganho estatístico na probabilidade dos dados.

Unigram

Usado pelo SentencePiece.
Parte de um vocabulário grande e remove tokens pouco úteis.

- Produzem resultados semelhantes ao BPE.
- Diferenças aparecem principalmente durante o treinamento do tokenizador.

Byte-level tokenization

- Trabalha diretamente sobre bytes.
- Qualquer texto pode ser representado.
- Não depende de alfabeto específico.
- Muito útil para código, emojis e caracteres raros.

Exemplos

Português
中文
Русский
Python

Vantagem

Não existem tokens desconhecidos.

Trade-offs de projeto

Vocabulário grande

- Menos tokens por frase.
- Embeddings maiores.
- Mais memória.
- Softmax mais caro.

Vocabulário pequeno

- Mais tokens por frase.
- Menor memória.
- Melhor cobertura.
- Contexto efetivo reduzido.

Tokenização em LLMs modernos

Modelo	Tokenizador	Vocabulário
GPT-2	Byte-level BPE	50k
Llama 2	SentencePiece	32k
Mistral	SentencePiece	32k
Gemma	SentencePiece	256k
BERT	WordPiece	30k

- Não existe um tokenizador universalmente melhor.
- A escolha depende de idiomas, domínio e restrições computacionais.

Mensagem principal

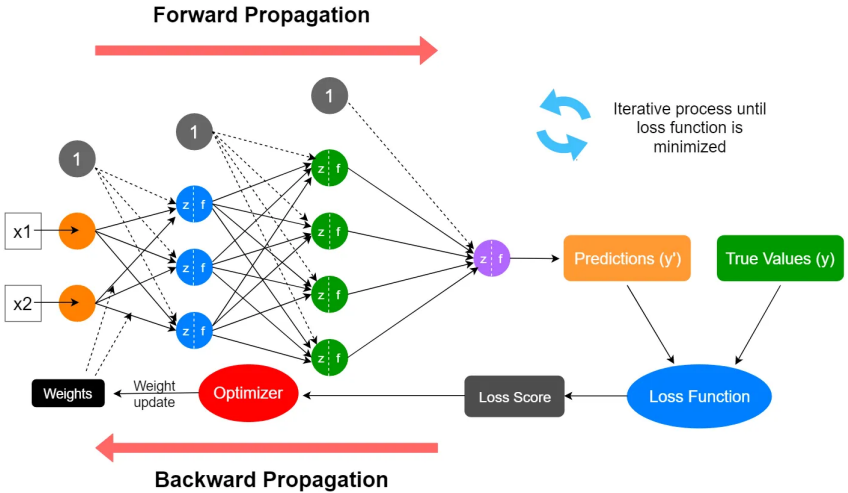
Uma visão simplificada

A tokenização é a interface entre linguagem humana e computação numérica.

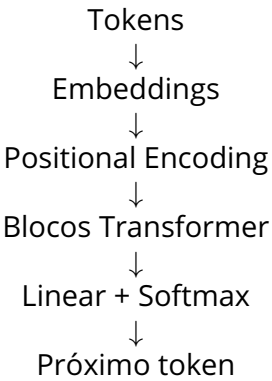
- Define o tamanho do contexto.
- Afeta custo, memória e velocidade.
- Influencia a capacidade multilíngue.
- Impacta diretamente a qualidade final do modelo.

$$\text{Dados} \rightarrow \text{Tokenização} \rightarrow \text{Embeddings} \rightarrow \text{Transformer}$$

Arquitetura



Visão geral da arquitetura



Objetivo

Transformar uma sequência discreta de tokens em uma distribuição de probabilidade para o próximo token.

Do token ao embedding

Texto:

"O céu é azul"

Tokenização:

[15, 381, 42, 901]

Embedding:

$$15 \rightarrow \begin{bmatrix} 0.2 \\ -1.1 \\ 0.7 \\ \vdots \end{bmatrix}$$

Cada token é mapeado para um vetor contínuo.

Embeddings

- Cada token recebe um vetor de dimensão $\mathrm{d}_{\mathrm{model}}$.
- Os vetores são aprendidos durante o treinamento.
- Tokens semanticamente relacionados tendem a ocupar regiões próximas do espaço vetorial.
- O embedding é frequentemente compartilhado com a camada de saída.

Forma

$$\mathrm{E} \in \mathbb{R}^{|\mathrm{V}| \times \mathrm{d}_{\mathrm{model}}}$$

Exemplo

$$50.000 \times 4096$$

Importante

O embedding não possui significado explícito por dimensão. O significado emerge da combinação de milhares de dimensões.

Propriedades do espaço vetorial

- Similaridade semântica aparece como proximidade geométrica.
- Conceitos relacionados tendem a formar clusters.
- Operações lineares podem capturar relações abstratas.

Exemplo clássico

$$\mathrm{v}(\text{rei}) - \mathrm{v}(\text{homem}) + \mathrm{v}(\text{mulher}) \approx \mathrm{v}(\text{rainha})$$

Embora menos evidente em LLMs modernos, esse comportamento ilustra o poder das representações distribuídas.

O problema da ordem

Considere as frases:

“O cachorro mordeu o homem”

“O homem mordeu o cachorro”

Possuem praticamente os mesmos tokens.

Problema

Sem informação posicional, o Transformer não consegue distinguir as duas sequências.

Atenção pura é invariante à permutação dos elementos.

Positional encoding

Objetivo

Adicionar informação sobre posição dos tokens.

- Ordem das palavras.
- Distância relativa.
- Estrutura da sequência.

Métodos modernos

- Sinusoidal.
- Learned Position Embeddings.
- RoPE.
- ALiBi.

RoPE: Rotary Positional Embeddings

- Método usado em Llama, Mistral, Gemma e diversos modelos modernos.
- Aplica rotações matemáticas aos vetores de atenção.
- Preserva relações relativas entre posições.
- Escala melhor para contextos longos.

Motivação

Posições relativas geralmente importam mais do que posições absolutas.

A ideia central da atenção

Quando lemos uma frase, nem todas as palavras são igualmente importantes.

"O gato que estava em cima da mesa caiu."

Ao processar "caiu", queremos olhar principalmente para:

gato

e não para todas as palavras com o mesmo peso.

Ideia

A atenção aprende automaticamente quais partes do contexto são relevantes.

Self-attention

Para cada posição da sequência:

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

- Q: Query
- K: Key
- V: Value

Cada token consulta os demais tokens do contexto e constrói uma representação contextualizada.

Intuição para Q, K e V

Query	Key	Value
O que estou procurando?	O que eu ofereço?	Qual informação carrego?

A compatibilidade entre Queries e Keys gera os pesos de atenção.
Esses pesos determinam quanto de cada Value será agregado.

Matriz de atenção

	T1	T2	T3	T4
T1	1.0	0	0	0
T2	0.6	0.4	0	0
T3	0.2	0.5	0.3	0
T4	0.1	0.3	0.4	0.2

Cada linha representa quanto um token observa os demais.
Os pesos são aprendidos dinamicamente para cada entrada.

Complexidade da atenção

Para uma sequência de comprimento n:

$$\mathrm{QK}^{\mathrm{T}}$$

gera uma matriz:

$$\mathrm{n} \times \mathrm{n}$$

Consequência

Custo computacional e memória crescem aproximadamente com:

$$\mathrm{O}(\mathrm{n}^2)$$

Esse é um dos principais gargalos para contextos longos.

Multi-head attention

- Uma única atenção é limitada.
- O modelo aprende múltiplas atenções em paralelo.
- Cada cabeça pode focar em padrões diferentes.

Head 1 $\rightarrow$ Sintaxe
Head 2 $\rightarrow$ Dependências longas
Head 3 $\rightarrow$ Co-referência
Head 4 $\rightarrow$ Estrutura de código

Por que múltiplas cabeças ajudam?

Uma única cabeça

Representação limitada.

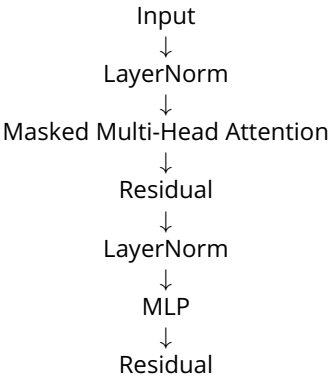
Múltiplas cabeças

Diversos relacionamentos podem ser aprendidos simultaneamente.

$$\text{MultiHead}(\mathrm{Q}, \mathrm{K}, \mathrm{V}) = \text{Concat}(\mathrm{h}_1, \ldots, \mathrm{h}_\mathrm{H})\mathrm{W}_\mathrm{O}$$

onde cada h_i é uma atenção independente.

Estrutura de um bloco Transformer



Residual connections

Ideia

Permitir que a informação original atravesse diversas camadas sem degradação.

$$y = x + F(x)$$

- Facilita treinamento profundo.
- Melhora fluxo de gradientes.
- Reduz problemas de desaparecimento de gradiente.

Máscara causal

- Impede acesso a tokens futuros.
- Garante consistência entre treino e inferência.
- Fundamental para geração autoregressiva.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

MLP dentro do Transformer

$$\mathrm{d_{model}} \rightarrow \mathrm{d_{ff}} \rightarrow \mathrm{d_{model}}$$

- Expansão de dimensionalidade.
- Aplicação de não linearidade.
- Compressão de volta ao espaço original.

Ativações comuns

- GELU
- SiLU
- ReLU
- SwiGLU
- GeGLU

SwiGLU e os LLMs modernos

Modelos recentes frequentemente substituem MLPs clássicos por variantes gated.

$$\text{SwiGLU}(\mathrm{x}) = (\mathrm{x}\mathrm{W}_1) \otimes \text{SiLU}(\mathrm{x}\mathrm{W}_2)$$

- Melhor eficiência de parâmetros.
- Maior capacidade representacional.
- Utilizado em Llama, PaLM e Mistral.

Decoder-only Transformer

- GPT
- Llama
- Mistral
- Gemma
- Qwen

- Máscara causal.
- Predição do próximo token.
- Escalabilidade elevada.
- Excelente para geração.

Escala de modelos modernos

Modelo	Parâmetros	Contexto	Heads
GPT-2	1.5B	1024	25
Llama 2 7B	7B	4096	32
Llama 3 70B	70B	8192+	64
Mistral 7B	7B	8192	32

Escalar parâmetros é apenas uma parte da história. Dados, contexto e otimizações arquiteturais são igualmente importantes.

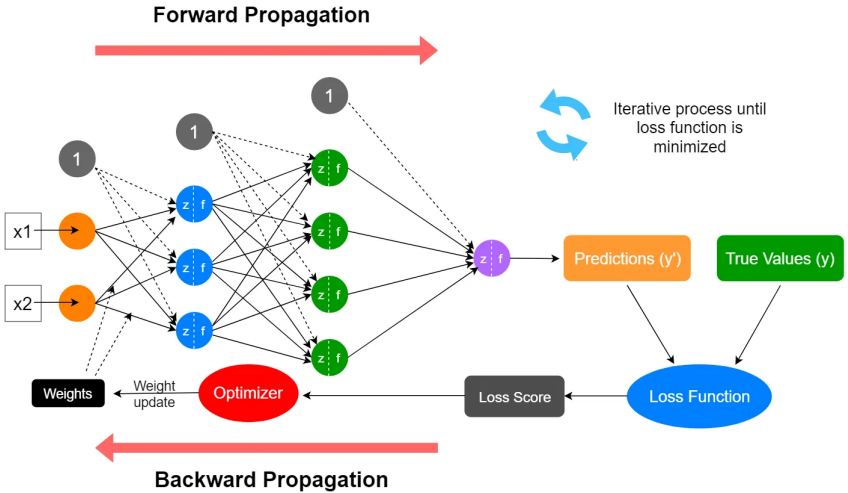
Resumo da arquitetura

1. Converter tokens em embeddings.
2. Adicionar informação posicional.
3. Aplicar múltiplos blocos Transformer.
4. Produzir logits para cada token do vocabulário.
5. Escolher o próximo token.
6. Repetir o processo autoregressivamente.

Essência

LLMs modernos são grandes pilhas de mecanismos de atenção treinadas para prever o próximo token.

Treinamento



Visão geral do treinamento

Corpus
↓
Tokenização
↓
Transformer
↓
Predição do próximo token
↓
Cálculo da perda
↓
Backpropagation
↓
Atualização dos pesos

Objetivo

Ajustar bilhões de parâmetros para minimizar o erro de previsão do próximo token.

Objetivo de pré-treinamento

Um LLM é treinado para prever o próximo token da sequência.

"O céu é"
$\rightarrow$
"azul"

A função objetivo é:

$$\mathcal{L} = -\sum_{t=1}^{T} \log P(x_t | x_{<t})$$

- Conhecida como Negative Log-Likelihood (NLL).
- Equivalente à Cross Entropy para classificação de tokens.

Exemplo de predição

Considere a frase:

“Paris é a capital da”

O modelo produz:

Token	Probabilidade
França	0.83
Alemanha	0.08
Europa	0.04
Itália	0.02

A perda mede o quão distante a distribuição prevista está da resposta correta.

Cross Entropy

Intuição

A perda penaliza previsões erradas e recompensa previsões corretas com alta confiança.

- Se a resposta correta recebe alta probabilidade, a perda é pequena.
- Se recebe baixa probabilidade, a perda cresce rapidamente.
- O treinamento tenta reduzir essa perda ao longo de bilhões de exemplos.

$$\text{Menor perda} \Rightarrow \text{melhor capacidade preditiva}$$

Forward pass

Fluxo de cálculo:

1. Converter tokens em embeddings.
2. Executar todos os blocos Transformer.
3. Produzir logits.
4. Aplicar softmax.
5. Calcular probabilidades.

Saída

$$z \to \text{softmax}(z)$$

Distribuição sobre todo o vocabulário.

Backpropagation

- A perda é calculada na saída.
- Gradientes percorrem a rede em sentido inverso.
- Cada parâmetro recebe uma atualização proporcional à sua contribuição para o erro.

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$$

Essência

Treinamento é um processo repetitivo de correção incremental dos pesos.

Loop de treinamento

1. Amostrar batch de sequências.
2. Executar forward pass.
3. Calcular cross entropy.
4. Executar backpropagation.
5. Atualizar pesos.
6. Registrar métricas.
7. Repetir milhões de vezes.

Escala

Grandes LLMs podem executar trilhões de passos de multiplicação de matrizes durante o treinamento.

Otimizador AdamW

- AdamW é o padrão dominante.
- Mantém estimativas do primeiro e segundo momentos dos gradientes.
- Adiciona weight decay desacoplado.
- Converge mais rapidamente que SGD em modelos grandes.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Learning rate schedule

Warmup

Aumenta gradualmente a taxa de aprendizado no início do treino.

Decay

Reduz a taxa de aprendizado conforme o treinamento progride.

Estratégias comuns:

- Linear decay.
- Cosine decay.
- Cosine with restarts.

Sem warmup

Treinos grandes frequentemente divergem logo nas primeiras iterações.

Gradient clipping

- Evita explosão de gradientes.
- Limita a magnitude máxima da atualização.
- Melhora estabilidade em modelos profundos.

$$\mathrm{g} \leftarrow \mathrm{g} \cdot \min\left(1, \frac{\tau}{||\mathrm{g}||}\right)$$

onde τ é o limite escolhido.

Batching e throughput

Objetivo

Maximizar utilização de GPU.

- Batches maiores.
- Menos tempo ocioso.
- Melhor throughput.

Problema

Padding excessivo desperdiça memória e FLOPs.

- Packing.
- Bucketing.
- Dynamic batching.

Gradient accumulation

Problema

Nem sempre a GPU comporta batches muito grandes.

- Acumular gradientes por vários mini-batches.
- Atualizar pesos apenas após várias iterações.
- Simular batches efetivamente maiores.

$$\mathrm{Batch}_{\mathrm{efetivo}} = \mathrm{Batch}_{\mathrm{GPU}} \times \mathrm{N}_{\mathrm{acum}}$$

Mixed Precision

FP32

Maior precisão.
Maior consumo de memória.

FP16/BF16

Menor memória.
Maior throughput.

- FP16 exige loss scaling.
- BF16 é mais estável.
- BF16 tornou-se padrão em GPUs modernas.

Consumo de memória

A memória total é composta por:

- Pesos do modelo.
- Gradientes.
- Estados do otimizador.
- Ativações intermediárias.

Surpresa comum

Em muitos casos, ativações consomem mais memória do que os próprios pesos.

Checkpointing

Salvar periodicamente:

- Pesos.
- Estado do AdamW.
- Scheduler.
- Métricas.

Benefícios

- Recuperação após falhas.
- Comparação de experimentos.
- Reprodutibilidade.

Leis de escala

- Mais parâmetros ajudam.
- Mais dados ajudam.
- Mais compute ajuda.

- Nenhum deles sozinho resolve.
- É necessário equilíbrio entre os três fatores.
- Subtreino desperdiça capacidade do modelo.

Leis de escala de Chinchilla

Resultado importante

Muitos modelos históricos eram grandes demais para a quantidade de dados utilizada.

O trabalho Chinchilla mostrou que:

- Mais parâmetros não são sempre a melhor escolha.
- Frequentemente vale mais aumentar os dados.
- Existe uma relação aproximadamente ótima entre parâmetros e tokens de treino.

Paralelismo em larga escala

Data Parallel	Tensor Parallel	Pipeline Parallel
Replica o modelo.	Divide matrizes.	Divide camadas.

Modelos modernos normalmente utilizam combinações dessas estratégias.

FlashAttention

- Reorganiza cálculos de atenção.
- Minimiza movimentação de memória.
- Mantém resultado matematicamente equivalente.
- Pode acelerar significativamente treino e inferência.

Ideia

Mover menos dados é frequentemente mais importante do que realizar menos operações.

Gradient checkpointing

- Não armazena todas as ativações.
- Recalcula parte delas durante o backward.
- Reduz memória.
- Aumenta tempo de computação.

Trade-off

Menos memória em troca de mais FLOPs.

Mixture of Experts (MoE)

- Apenas parte da rede é ativada por token.
- Capacidade total cresce muito.
- Custo computacional cresce menos.

Exemplo

Um modelo pode possuir centenas de bilhões de parâmetros, mas utilizar apenas uma fração deles em cada inferência.

Atenção e contexto longo

- Contexto maior permite incorporar mais informação.
- O custo da atenção cresce aproximadamente com n^2.
- Contextos longos exigem otimizações especializadas.

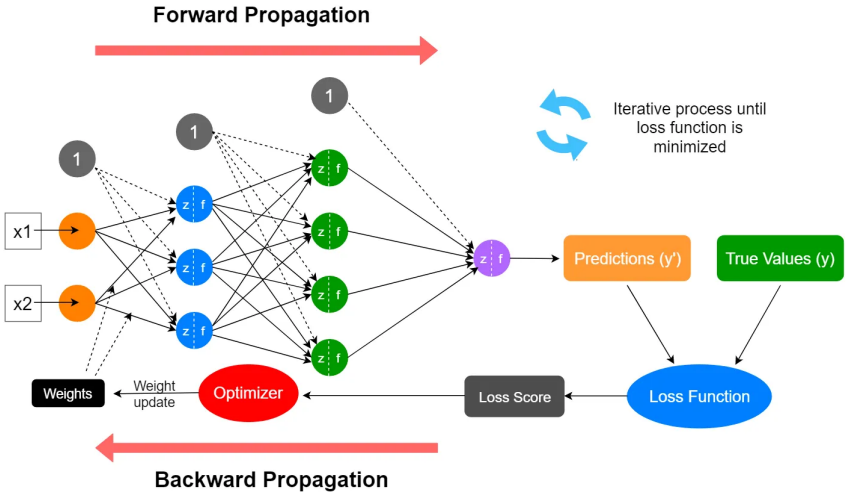
$$4\text{k} \rightarrow 8\text{k} \rightarrow 32\text{k} \rightarrow 128\text{k tokens}$$

Nem toda aplicação se beneficia igualmente de contextos muito longos.

Resumo

- Treinar um LLM significa minimizar erro de previsão de tokens.
- AdamW, warmup e mixed precision são componentes fundamentais.
- Escala depende de dados, parâmetros e compute.
- Memória é frequentemente o principal gargalo.
- Técnicas como FlashAttention, checkpointing e MoE tornam modelos modernos viáveis.

Pós-treinamento



Do base model ao assistente

Pré-treinamento
↓
Base Model
↓
Instruction Tuning
↓
Preference Alignment
↓
Assistant Model

Ideia principal

O pré-treinamento ensina linguagem.
O pós-treinamento ensina comportamento.

O que falta após o pré-treinamento?

Um modelo pré-treinado geralmente sabe completar texto, mas não necessariamente seguir instruções.

Prompt:

"Explique o que é gravidade em duas frases."

Possível resposta de um modelo base:

"Gravidade é uma força física. Einstein desenvolveu..."

e continua indefinidamente.

O modelo ainda não aprendeu:

- Obediência a instruções.
- Formatação desejada.
- Estilo conversacional.
- Restrições de segurança.

Supervised Fine-Tuning (SFT)

Entrada

Instrução do usuário.

Saída

Resposta ideal escrita por humanos.

“Resuma este texto”

Resumo correto.

- O modelo aprende exemplos de comportamento desejado.
- É a primeira etapa do alinhamento moderno.
- Frequentemente utiliza centenas de milhares ou milhões de exemplos.

Instruction tuning

- Dados no formato instrução-resposta.
- Aprendizado de padrões conversacionais.
- Formatação consistente.
- Obediência a comandos.
- Melhor generalização para tarefas novas.

Exemplos

- Tradução
- Resumo
- Classificação
- Perguntas e respostas
- Geração de código

Origem dos dados de instrução

- Anotadores humanos.
- Logs de uso previamente autorizados.
- Conjuntos acadêmicos.
- Dados sintéticos gerados por modelos maiores.

Tendência atual

Grande parte dos datasets modernos de instrução é gerada sinteticamente e posteriormente filtrada.

Alinhamento por preferências

Após o SFT, ainda existem múltiplas respostas corretas.

Pergunta:

“Explique redes neurais.”

Resposta A:

- Correta.

Resposta B:

- Correta.
- Mais clara.
- Mais útil.

O objetivo passa a ser aprender preferências humanas.

RLHF

RLHF significa:

Reinforcement Learning from Human Feedback

Pipeline clássico:

1. Treinar modelo SFT.
2. Coletar preferências humanas.
3. Treinar reward model.
4. Aplicar reinforcement learning.

Como funciona o RLHF

Prompt:

“Explique LLMs”

- Resposta A
- Resposta B

Humano escolhe a melhor.

Milhares de comparações geram um conjunto de preferências que alimenta o reward model.

Reward Model

Objetivo

Prever qual resposta um humano preferiria.

Entrada:

- Prompt.
- Resposta candidata.

Saída:

$$\text{Reward} = 8.7$$

Quanto maior o score, mais desejável a resposta.

PPO no RLHF

Historicamente o algoritmo mais utilizado foi PPO.

- Atualiza a política gradualmente.
- Evita mudanças muito agressivas.
- Mantém estabilidade.

Desvantagem

Pipeline complexo e operacionalmente caro.

DPO: Direct Preference Optimization

RLHF

SFT
→
Reward Model
→
PPO

DPO

SFT
→
Preferências
→
Treino direto

- Elimina reward model explícito.
- Mais simples de implementar.
- Tornou-se extremamente popular em modelos open-source.

Constitutional AI

Outra abordagem moderna consiste em substituir parte da supervisão humana por princípios explícitos.

- Regras de comportamento.
- Critérios de segurança.
- Autoavaliação por modelos.

Exemplo

“Prefira respostas úteis, honestas e seguras.”

Popularizada pela Anthropic.

O problema da avaliação

Modelos generativos possuem infinitas respostas possíveis.

Duas respostas podem ser:

- Diferentes.
- Corretas.
- Úteis.

Avaliar LLMs é mais difícil do que avaliar classificadores tradicionais.

Métricas automáticas

- Perplexity.
- Accuracy.
- F1-score.
- Exact Match.

- BLEU.
- ROUGE.
- BERTScore.
- Pass@k para código.

Perplexity

Interpretação

Número médio de alternativas que o modelo considera plausíveis para o próximo token.

$$\mathrm{PPL} = \mathrm{e}^{\mathrm{Loss}}$$

- Menor é melhor.
- Excelente para acompanhar treinamento.
- Pouco correlacionada com utilidade final.

Benchmarks modernos

Benchmark	Objetivo
MMLU	Conhecimento geral
GSM8K	Matemática
HumanEval	Código
MBPP	Programação
ARC	Raciocínio científico
HellaSwag	Commonsense

Avaliação humana

- Clareza.
- Utilidade.
- Precisão factual.
- Segurança.
- Robustez.

Ainda é o padrão ouro

Nenhuma métrica automática substitui completamente avaliação humana.

LLM-as-a-Judge

Modelos modernos também são utilizados para avaliar outros modelos.

- Comparação automática.
- Escalabilidade.
- Menor custo operacional.

Limitação

Juízes automáticos podem herdar vieses dos modelos que os geraram.

Avaliação de segurança

- Toxicidade.
- Discurso de ódio.
- Assédio.
- Conteúdo ilegal.

- Vazamento de dados.
- Prompt injection.
- Jailbreaks.
- Ataques adversariais.

O que acontece na inferência?

Prompt
↓
Tokenização
↓
Forward Pass
↓
Próximo Token
↓
Novo Contexto
↓
Repetir

Geração autoregressiva

1. Receber prompt.
2. Converter para tokens.
3. Produzir logits.
4. Aplicar estratégia de decoding.
5. Gerar próximo token.
6. Atualizar contexto.
7. Repetir até EOS.

Decoding strategies

Método	Diversidade	Determinismo
Greedy	Baixa	Alta
Beam Search	Baixa	Alta
Top-k	Média	Média
Top-p	Alta	Média
Temperature	Controla	Controla

Temperature

A temperatura modifica a distribuição dos logits.

$$\mathrm{P_i} = \frac{\mathrm{e}^{\mathrm{z_i}/\mathrm{T}}}{\sum_{\mathrm{j}} \mathrm{e}^{\mathrm{z_j}/\mathrm{T}}}$$

- $\mathrm{T} < 1$: respostas mais conservadoras.
- $\mathrm{T} = 1$: distribuição original.
- $\mathrm{T} > 1$: maior criatividade.

Top-k e Top-p

Top-k

Seleciona os k tokens mais prováveis.

Top-p

Seleciona o menor conjunto cuja probabilidade acumulada atinge p.

Top-p costuma adaptar-se melhor a diferentes contextos.

KV Cache

- Armazena Keys e Values previamente calculados.
- Evita recomputação da atenção.
- Reduz drasticamente latência.

Sem cache

Reprocessar toda a sequência.

Com cache

Processar apenas o novo token.

Speculative Decoding

- Modelo pequeno propõe vários tokens.
- Modelo grande valida as propostas.
- Acelera inferência sem alterar significativamente a qualidade.

Tendência recente

Amplamente adotado em sistemas de produção de larga escala.

Quantização

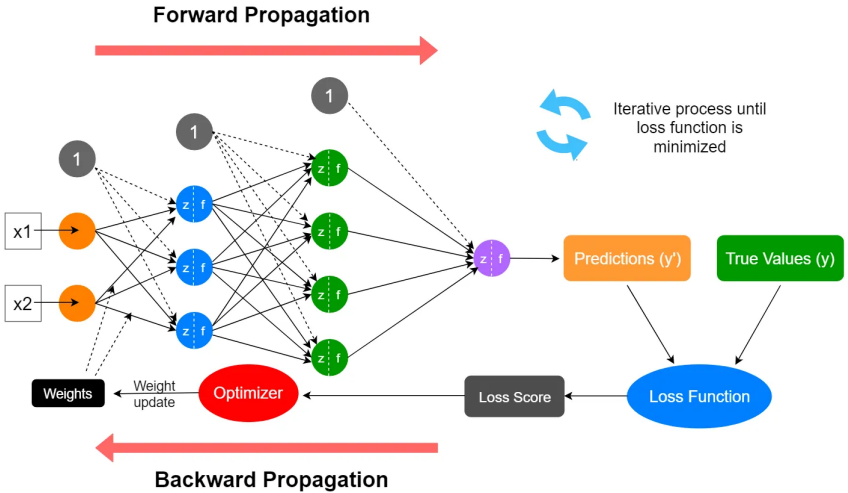
- FP16
- BF16
- INT8
- INT4

- Menos memória.
- Maior throughput.
- Menor custo.
- Possível perda de qualidade.

Resumo

- Pós-treinamento transforma modelos em assistentes.
- SFT, RLHF e DPO são as principais estratégias de alinhamento.
- Avaliação exige métricas automáticas e humanas.
- Inferência é um processo autoregressivo iterativo.
- KV cache, speculative decoding e quantização são fundamentais para produção.

Python e C++


Forward Propagation
Iterative process until loss function is minimized
x1
x2
Predictions (y')
True Values (y)
Weights
Weight update
Optimizer
Loss Score
Loss Function
Backward Propagation

Por que Python domina IA?

- Sintaxe simples.
- Desenvolvimento rápido.
- Ecossistema científico maduro.
- Grande comunidade.

- NumPy.
- PyTorch.
- TensorFlow.
- Hugging Face.

Importante

Python não é necessariamente rápido.
Grande parte do desempenho vem de bibliotecas implementadas em C++, CUDA e kernels especializados.

Stack típica de desenvolvimento

Datasets
↓
Tokenização
↓
PyTorch
↓
Treinamento
↓
Checkpoint
↓
Exportação

Ferramentas comuns:

- Hugging Face Transformers
- Datasets
- Accelerate
- DeepSpeed
- FSDP

Componentes de um projeto LLM

Componente	Ferramenta comum
Dataset	datasets
Tokenização	tokenizers
Treinamento	PyTorch
Distribuído	DeepSpeed/FSDP
Logs	TensorBoard/W&B
Avaliação	lm-eval-harness
Deploy	vLLM/TGI

Estrutura mínima em PyTorch

```
dataset -> dataloader
          -> model
          -> optimizer
          -> scheduler

for batch in loader:
    loss = train_step(batch)
    loss.backward()
    optimizer.step()
```

Abstração

Todo treinamento de LLM é uma versão sofisticada desse loop.

Esqueleto de treino em Python

```python
for step, batch in enumerate(loader):

    input_ids = batch["input_ids"].to(device)
    labels = batch["labels"].to(device)

    optimizer.zero_grad()

    logits = model(input_ids)

    loss = criterion(
        logits.view(-1, vocab_size),
        labels.view(-1)
    )

    loss.backward()

    torch.nn.utils.clip_grad_norm_(
        model.parameters(),
        1.0
    )

    optimizer.step()
    scheduler.step()
```

O que acontece em cada iteração?

1. Ler batch da GPU.
2. Executar forward pass.
3. Calcular cross entropy.
4. Executar backward pass.
5. Atualizar parâmetros.
6. Registrar métricas.

Escala

Treinar um LLM moderno significa repetir esse ciclo bilhões de vezes.

Exemplo com Hugging Face

```python
from transformers import AutoModel

model = AutoModel.from_pretrained(
    "meta-llama/Llama-3-8B"
)

tokenizer = AutoTokenizer
    .from_pretrained(...)
```

Poucas linhas permitem carregar modelos com bilhões de parâmetros.

Treinamento distribuído

DDP
Replica o modelo.

FSDP
Fragmenta pesos.

DeepSpeed
Otimizações avançadas.

Essas ferramentas permitem treinar modelos maiores do que a memória de uma única GPU.

Observabilidade durante o treino

Monitorar:

- Loss.
- Learning rate.
- Throughput.
- Uso de GPU.
- Grad norm.
- Consumo de memória.

Ferramentas

TensorBoard, Weights & Biases, MLflow.

Boas práticas em Python

- Seeds fixas.
- Checkpoints frequentes.
- Configuração versionada.
- Experimentos reproduzíveis.
- Logs estruturados.
- Separação treino/validação.
- Monitoramento contínuo.
- Testes automatizados.

Por que C++ continua importante?

- Controle fino de memória.
- Baixa latência.
- Alto throughput.
- Vetorização.
- Integração com CUDA.
- Runtime de produção.

Grande parte dos kernels usados por PyTorch e TensorRT é implementada em C++.

Onde C++ aparece em LLMs?

- Runtime de inferência.
- TensorRT.
- ONNX Runtime.
- llama.cpp.
- Kernels CUDA.
- Quantização.
- Serving de alta performance.

Arquitetura típica de produção

Python
↓
Treinamento
↓
Checkpoint
↓
ONNX / TensorRT / GGUF
↓
Runtime C++
↓
API REST / gRPC

Formatos de exportação

Formato	Uso principal	Ecossistema
ONNX	Interoperabilidade	Geral
TensorRT	NVIDIA	GPU
GGUF	llama.cpp	CPU/GPU
TorchScript	PyTorch	Produção

llama.cpp

Projeto open-source extremamente popular.

- Implementação em C/C++.
- Suporte a quantização agressiva.
- Execução local.
- Funciona em CPU, GPU e dispositivos embarcados.

Importância

Demonstrou que LLMs podem rodar eficientemente fora de datacenters.

Exemplo conceitual de inferência

```cpp
auto tokens =
    tokenizer.encode(prompt);

auto kv_cache =
    init_kv_cache();

for (int t = 0;
     t < max_new_tokens;
     ++t)
{
    auto logits =
        model.forward(
            tokens,
            kv_cache
        );

    int next =
        sampler.sample(logits);

    tokens.push_back(next);

    if(next == eos_id)
        break;
}
```

O que domina o tempo de inferência?

- Multiplicações de matrizes.
- Atenção.
- Transferência de memória.
- KV cache.

Na prática

Movimentar dados costuma ser mais caro do que realizar operações aritméticas.

Otimizações em C++

- SIMD.
- AVX2.
- AVX512.
- Kernel fusion.
- Prefetching.
- Thread pools.
- Cache locality.
- Quantização.

CUDA e kernels customizados

Em GPUs modernas, boa parte do desempenho vem de kernels especializados.

- FlashAttention.
- GEMM otimizado.
- Quantização.
- Fusão de operações.

Bibliotecas como cuBLAS e CUTLASS são amplamente utilizadas.

Python versus C++

Aspecto	Python	C++
Produtividade	Alta	Média
Pesquisa	Excelente	Limitada
Treinamento	Excelente	Raro
Inferência	Boa	Excelente
Latência	Média	Baixa
Controle de memória	Baixo	Alto

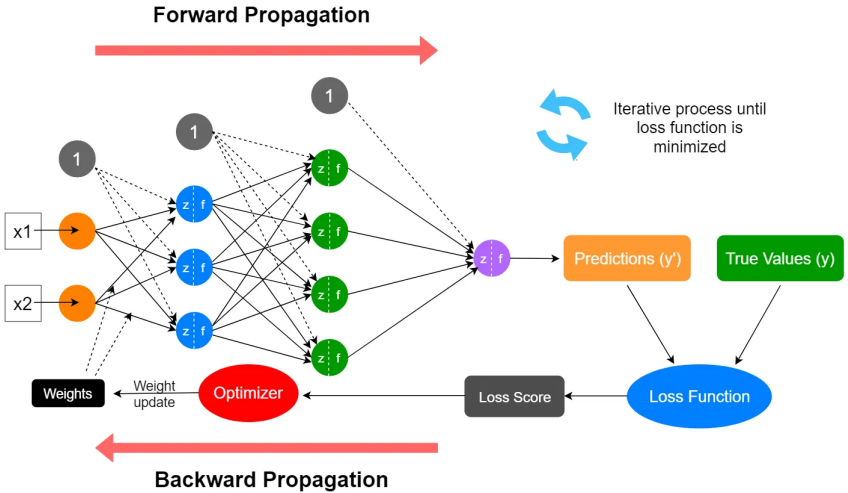
Mensagem principal

Regra prática

Treine em Python.
Otimize e sirva em C++.

- Python domina pesquisa e treinamento.
- C++ domina desempenho e produção.
- Sistemas modernos combinam os dois ecossistemas.

Sistema completo



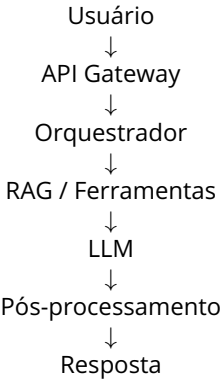
Visão sistêmica de um produto com LLM

Dados
↓
Treinamento
↓
Pós-treinamento
↓
Inferência
↓
Aplicação
↓
Feedback
↓
Nova versão

Ideia principal

Um LLM é apenas um componente de um sistema maior que envolve dados, infraestrutura, produto e operação contínua.

Arquitetura moderna de produto



A maioria dos produtos modernos utiliza vários componentes além do modelo principal.

Anatomia de uma requisição

1. Usuário envia prompt.
2. Sistema recupera contexto adicional.
3. Prompt final é montado.
4. Modelo gera resposta.
5. Filtros verificam segurança.
6. Resposta é entregue ao usuário.
7. Logs alimentam monitoramento.

Arquitetura de produto

1. Coletar dados.
2. Limpar e versionar.
3. Treinar modelo base.
4. Validar desempenho.
5. Alinhar comportamento.
6. Implantar em produção.
7. Monitorar uso real.
8. Iterar continuamente.

O ciclo de melhoria contínua

Produção
↓
Feedback
↓
Análise de falhas
↓
Novos dados
↓
Nova versão

Observação

Modelos de sucesso evoluem continuamente após o lançamento inicial.

Monitoramento em produção

Infraestrutura

- Latência.
- Throughput.
- Uso de GPU.
- Taxa de erro.

Qualidade

- Satisfação do usuário.
- Taxa de rejeição.
- Hallucinations.
- Segurança.

Drift de distribuição

Problema

Usuários reais frequentemente fazem perguntas diferentes das observadas durante treinamento.

- Mudança de domínio.
- Mudança de linguagem.
- Novos comportamentos.
- Eventos recentes.

Isso pode degradar a qualidade do sistema ao longo do tempo.

Observabilidade para LLMs

Registrar:

- Prompt.
- Contexto recuperado.
- Resposta.
- Tempo de inferência.
- Uso de tokens.
- Feedback do usuário.

Objetivo

Entender por que uma resposta foi produzida.

RAG versus Fine-Tuning

RAG	Fine-Tuning
Atualiza conhecimento	Atualiza comportamento
Não re-treina modelo	Requer treinamento
Fácil atualização	Atualização mais lenta
Bom para fatos	Bom para estilo e tarefas

Na prática, muitos sistemas utilizam ambos simultaneamente.

Quando usar RAG?

- Base documental muda frequentemente.
- Conhecimento precisa estar atualizado.
- Existe grande volume de documentos.
- É necessário citar fontes.

Exemplos

Suporte técnico, bases corporativas, documentação interna e sistemas de busca.

Quando usar Fine-Tuning?

- Ajustar estilo.
- Ajustar formato.
- Especializar comportamento.
- Aprender tarefas repetitivas.

Exemplos

Assistentes jurídicos, médicos, financeiros ou copilotos especializados.

Quando não treinar do zero

- Orçamento limitado.
- Poucos dados.
- Equipe pequena.
- Prazo curto.
- Modelo aberto suficiente.
- API já atende o caso.

Treinar do zero raramente é a melhor decisão para um primeiro produto.

Estimativa de custo de decisão

Estratégia	Custo	Complexidade
API comercial	Baixo	Baixa
RAG	Médio	Média
Fine-tuning	Médio/alto	Média
Treino completo	Muito alto	Muito alta

Plano de laboratório

1. Treinar mini-transformer.
2. Avaliar perplexity.
3. Aplicar SFT.
4. Adicionar RAG.
5. Quantizar modelo.
6. Servir via API local.

Mini projeto sugerido

Objetivo

Construir um assistente especializado em um domínio específico.

- Dataset próprio.
- Modelo pequeno (100M–1B parâmetros).
- Fine-tuning instrucional.
- Base vetorial para RAG.
- Interface web simples.

Evolução sugerida do projeto

1. Prompt engineering.
2. RAG.
3. Fine-tuning.
4. Quantização.
5. Deploy.
6. Monitoramento.

Mensagem

A ordem importa. Não comece pelo treinamento mais caro.

Erros comuns de iniciantes

- Ignorar qualidade dos dados.
- Medir apenas accuracy.
- Não controlar leakage.
- Overfitting em benchmarks.
- Ignorar latência.
- Ignorar custo.
- Não monitorar produção.
- Treinar sem hipótese clara.

Mapa mental

Dados
$\rightarrow$
Tokenização
$\rightarrow$
Embeddings
$\rightarrow$
Transformer
$\rightarrow$
Treinamento
$\rightarrow$
Alinhamento
$\rightarrow$
Inferência
$\rightarrow$
Produto

Resumo final

- LLMs são modelos probabilísticos treinados para prever tokens.
- Dados frequentemente importam mais do que arquitetura.
- Transformers são a base dos modelos modernos.
- Pós-treinamento transforma modelos em assistentes úteis.
- Produção exige monitoramento, otimização e melhoria contínua.
- Python domina pesquisa e treinamento.
- C++ domina inferência e desempenho.