Fundamentos de LLMs – Large Language Model

Transformers

Os transformers foram criados em 2017 pela Google no artigo "Attention Is All You Need".

Transformers são uma arquitetura de redes neurais desenvolvida para processar e entender sequências de texto de forma eficiente. Diferente dos modelos tradicionais, como RNNs e LSTMs, os transformers utilizam mecanismos de atenção que permitem analisar todas as partes de um texto simultaneamente, identificando relações contextuais entre palavras, mesmo que estejam distantes umas das outras.

O desenvolvimento dos transformers marcou um grande avanço no campo de processamento de linguagem natural, pois permitiu modelos significativamente mais eficientes e escaláveis em relação às arquiteturas anteriores, como RNNs e LSTMs.

Desde então, essa arquitetura revolucionou diversas aplicações de inteligência artificial, tornando-se a base para os principais modelos de linguagem atuais, como GPT, BERT, T5 e outros.

De maneira didática, imagine que o transformer funciona como alguém lendo um texto e, a cada palavra, consegue olhar para todo o restante do texto para decidir qual é o significado mais apropriado naquele contexto. Esse mecanismo de atenção faz com que o modelo seja capaz de compreender nuances e dependências complexas, tornando-o muito poderoso para tarefas de linguagem natural, como tradução, resumo e geração de texto.

Token

A máquina não lê palavras, lê números. O processo de Tokenization quebra o texto em pedaços.

Na prática 1000 tokens ≈ 750 palavras (Inglês/Português).

A cobrança de APIs e o limite de processamento são baseados em tokens (Input + Output).

Exemplo:

Paralelepípedo = 3 tokens (Para, lele, pípedo).

LLM

Modelos de Linguagem Grande baseados na arquitetura Transformer.

Previsão estatística da próxima palavra (token) baseada em um contexto anterior.

Modelos:

- Base Models: Completam texto (ex: GPT-3 davinci antigo).
- Chat/Instruct Models: Treinados para seguir instruções e manter diálogo (ex: Gemini 1.5 Pro, GPT-4o, Claude 3.5).

Alucinação: O modelo tenta preencher lacunas estatísticas com dados verossímeis, mas não necessariamente verdadeiros.

Vetores e Embeddings

Vetores

Um texto é passado para a máquina como um único vetor.

Vetores Esparsos (Sparse Vectors)

Possui um dicionário com todas as palavras do vocabulário. Cada palavra possui duas coordenadas TF e IDF.

Esse vetor nesse dicionário terá uma dimensão para cada palavra. Como um documento não tem todas as palavras do dicionário, o vetor é composto por muitos zeros.

Exemplo do problema de sinonímia: o vetor de um documento contendo "carro" e "banana" não reconheceria “automóvel” no contexto.

Embeddings Densos (Dense Vectors)

Não existe Dicionário. O vetor não tem uma dimensão para palavra.

A dimensão desse vetor é fixa e compacta (768, 1536 ou 3072), formado por pesos aprendidos por uma rede neural durante o treinamento.

Cada dimensão representa uma característica latente (abstrata) do significado. Palavras com significados parecidos ficam "perto" uma da outra nesse espaço matemático.

Bibliotecas: GoogleGenerativeAIEmbeddings, OpenAI Embeddings.

Exemplo:

Imagine um gráfico 2D simples (Eixo X = Realeza, Eixo Y = Gênero).

Vetores:

Rei: [0.9, 0.9]

Homem: [0.1, 0.9]

Mulher: [0.1, 0.1]

Qual é o vetor da Rainha?

Cálculo: Rei - Homem + Mulher = Rainha

[0.9, 0.9] - [0.1, 0.9] = [0.8, 0.0] ("Realeza neutra")

[0.8, 0.0] + [0.1, 0.1] = [0.9, 0.1] (Vetor da Rainha)

Característica	Vetor TF-IDF (Clássico)	Embedding (Moderno/LLM)
Coordenadas	Frequência da palavra (Estatística)	Peso semântico (Rede Neural)
Tamanho	Gigante (Tamanho do Vocabulário)	Fixo (ex: 768 ou 1536)
Zeros?	Muitos (Esparso)	Quase nenhum (Denso)
Busca por	Palavras-chave exatas	Significado / Intenção
Exemplo	"Advogado" ≠ "Jurista"	"Advogado" ≈ "Jurista"

Hybrid Search

Os melhores sistemas (como o que empresas usam em produção) utilizam Busca Híbrida.

Vetor Denso (Embeddings) para achar o contexto e significado.

Vetor Esparso (BM25/TF-IDF) para garantir que palavras-chave específicas estejam presentes.

Estratégias de Memória: Contexto vs. Custo

Chat IA Comum (Sem Contexto Externo)

O modelo "cru", como o ChatGPT ou Gemini web gratuito.

Você envia apenas a pergunta (prompt). O modelo responde com base no treino dele.

Sem implementação.

Uso: Brainstorming, correção de código, conhecimentos gerais.

Custo: só os tokens da pergunta e da resposta. Se tiver histórico ele também é incluído da pergunta.

Limitação: qualquer coisa que não saiba, alucina.

Envio de Todos os PDFs (Janela de Contexto Longa)

A abordagem "Força Bruta".

Além do prompt insere PDFs. Se tiver caching os PDFs ficam na memoria por um determinado tempo.

Sem implementação, mas precisa inserir os PDFs.

Uso: análise profunda.

Custo e limitação: custo muito alto sem caching.

RAG Padrão (Retrieval-Augmented Generation)

A abordagem "Bibliotecário Eficiente".

Implementação: código para ler PDF, transformar em banco vetorial, fazer pesquisa do prompt neste banco e enviar o prompt mais os vetores mais similares.

Uso: Chatbots corporativos, "Converse com seus dados", Sistemas de busca em manuais técnicos.

Custo: baixo e estável.

RAG com Ranqueamento (Rerank)

A abordagem "Bibliotecário Especialista".

Implementação: Parecido com o anterior, mas recupera mais vetores. Então ranqueia estes vetores e envia junto com o prompt os melhores.

Uso: Sistemas jurídicos ou médicos onde a precisão é crítica e "quase certo" não é suficiente.

Custo: baixo mas demora um pouco mais.

Tabela Comparativa de Custos e Precisão

Método	**O que o LLM lê por vez?**	**Custo Estimado (100 perguntas)**	**Precisão**	**Latência (Velocidade)**
Chat Comum	Só a pergunta	< US$ 0,01	Nula (Não lê o livro)	Instantânea
Envio Direto (Full)	O Livro todo (200k)	~US$ 70,00	Altíssima (Contexto Global)	Lenta (processa muito)
RAG Padrão	~3 parágrafos (1k)	~US$ 0,35	Boa (Fatos pontuais)	Rápida
RAG + Rerank	~5 parágrafos filtrados	~US$ 0,50	Muito Boa	Média

Custo simulado para **100 perguntas** em uma base de 1 PDF (500 páginas ~ 200 mil tokens).

Preços estimados baseados no Gemini 1.5 Pro (US$ 3.50 / 1M input).

Construindo Sistemas Inteligentes com LLMs
(minhas anotações melhoradas com o raciocínio Gemini 3 Pro)

1. O Conceito Geral: A Arquitetura do Chatbot

Antes de codificar, vamos visualizar o fluxo. Não estamos apenas "falando com um robô", estamos construindo um **pipeline de dados**.

O Algoritmo Mestre (O Loop Infinito):

1. **Configuração Inicial:** O usuário apresenta sua credencial (API Key) e escolhe o "Cérebro" (Google, OpenAI, etc.).
2. **Seleção de Modo:** O usuário decide a estratégia (Chat simples? Consulta a documentos?).
3. **Loop de Conversa:**
   - Entrada: Usuário faz a pergunta.
   - Processamento: O sistema busca informações extras (se necessário).
   - Saída: A IA gera a resposta.

1.1. Níveis de Complexidade (Modos de Operação)

Para facilitar o entendimento, vamos imaginar uma escada de inteligência:

- **Nível 1 - Chat Simples:** "Conversa de Bar". A IA usa apenas o que já sabe (treinamento prévio).
- **Nível 2 - Contexto Full:** "Ler o Livro". Você entrega o PDF inteiro no momento da pergunta. *Problema: Caro e limitado pelo tamanho da janela de contexto.*
- **Nível 3 - RAG (Retrieval-Augmented Generation):** "O Bibliotecário". O sistema busca apenas os parágrafos relevantes no PDF e entrega para a IA.
- **Nível 4 - RAG + Rerank:** "O Especialista". O Bibliotecário traz 20 livros, e um Especialista seleciona os 5 melhores trechos antes de entregar para a IA.
- **Nível 5 - Smart Context (Rerank + Docs):** "Leitura Profunda". O sistema identifica qual é o documento mais relevante e, em vez de ler só um trecho, lê o documento inteiro daquele tópico específico.

2. A "Matéria-Prima": Dados e Objetos

No mundo do LangChain (a biblioteca que usaremos), tudo gira em torno de um objeto fundamental.

O Objeto Document

Imagine isso como uma folha de papel dentro de uma pasta:

Python

```
class Document:
    page_content: str  # O texto escrito na folha (O conteúdo)
```

metadata: dict # A etiqueta da pasta (Fonte, página, autor, data)

3. O Passo-a-Passo da Construção (Pipeline)

Vamos dividir a construção em etapas lógicas, como uma linha de montagem.

Passo 1: Ingestão (Trazendo os PDFs)

O computador não lê PDF nativamente como nós. Precisamos extrair o texto.

- **Ferramenta:** PyPDFLoader
- **Ação:** Lê o arquivo binário e converte em texto puro.
- **Código:**

Python

```
loader = PyPDFLoader("manual.pdf")
documentos_brutos = loader.load() # Retorna uma lista de páginas
```

Passo 2: Chunking (Fatiando o Conteúdo)

As IAs têm memória curta. Não podemos enviar um livro inteiro de uma vez (ou seria muito caro). Precisamos quebrar em pedaços menores (**Chunks**).

- **Conceito Importante (Overlap):** Ao cortar o texto, deixamos uma "borda" repetida.
  - *Sem Overlap:* Corte seco. Pode quebrar uma frase no meio.
  - *Com Overlap:* Como telhas num telhado, um pedaço cobre o início do outro para não perder o contexto da frase cortada.
- **Ferramenta:** RecursiveCharacterTextSplitter
- **Código:**

Python

```
splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,  # Tamanho do pedaço
    chunk_overlap=200 # A "borda" de segurança
)
chunks = splitter.split_documents(documentos_brutos)
```

Passo 3: O Cérebro e o Tradutor (Configuração Modular)

Precisamos de dois componentes:

1. **Embeddings (O Tradutor):** Transforma texto em números (vetores).
2. **LLM (O Cérebro):** Gera a resposta textual. *Esta etapa é modular. Deve-se escolher um fornecedor.*

Fornecedor	LLM (Cérebro)	Embeddings (Tradutor)	Observação
Google	ChatGoogleGenerativeAI Modelo: *gemini-1.5-pro*	GoogleGenerativeAIEmbeddings Modelo: *text-embedding-004*	Ótimo custo-benefício (Tier gratuito).
OpenAI	ChatOpenAI Modelo: *gpt-4o*	OpenAIEmbeddings Modelo: *text-embedding-3-small*	Padrão da indústria. Pago.
Ollama	ChatOllama Modelo: *llama3*	OllamaEmbeddings Modelo: *llama3*	Roda local (Offline). Exige RAM.
Anthropic	ChatAnthropic Modelo: *claude-3-5*	*Não possui* (Use OpenAI/Google)	Raciocínio lógico superior.
DeepSeek	ChatOpenAI (Adaptado) URL: *https://www.google.com/search?q=api.deepseek.com*	*Não possui* (Use OpenAI/Google)	Muito barato e potente.

Passo 4: O Banco de Dados Vetorial (Vector Store)
Base para os vetores (embeddings) dos chunks.

- **Ação:** Salva o texto + o vetor num banco de dados local.
- **Ferramenta:** Chroma
- **Código:**

Python
```
vector_db = Chroma(
    persist_directory="./banco_de_dados",
    embedding_function=embeddings_escolhidos # O tradutor do Passo 3
)
vector_db.add_documents(chunks) # Salva tudo
```

Passo 5: Recuperação (Retrieval & Rerank)
Como encontramos a informação certa?

- **Método A: Busca Simples (as_retriever)**
- Busca por similaridade matemática. "Quem está perto deste vetor?".
- *Parâmetro k=5:* Traz os 5 vizinhos mais próximos.

- **Método B: Busca com Rerank (Refinamento)**
- Primeiro trazemos muitos candidatos (ex: 20).
- Depois, usamos um modelo mais inteligente (FlashrankRerank) para reordenar e escolher o "Top 5" real.
- *Analogia:* O estagiário traz 20 livros da estante (Busca vetorial), e o professor escolhe os 5 melhores para a aula (Rerank).

Passo 6: A Geração (A Resposta Final)
Agora unimos tudo numa **Chain** (Corrente).
Ferramenta: RetrievalQA
O que ela faz:
- Recebe a pergunta do usuário.
- Vai no Banco Vetorial (ou Reranker) buscar os textos de apoio.
- Monta um Prompt: *"Com base nestes textos [X, Y, Z], responda à pergunta [P]"*.
- Envia para a LLM.
- Devolve a resposta.

Código:
Python

```
qa_chain = RetrievalQA.from_chain_type(
    llm=llm_escolhida,
    retriever=meu_retriever, # Pode ser o simples ou o rerank
    return_source_documents=True # Importante: Para saber de onde veio a info!
)
resposta = qa_chain.invoke({"query": "Qual a conclusão?"})
```

Profissionalizando o sistema:
- **Memória:** Transformar o RetrievalQA em ConversationalRetrievalChain para o chat lembrar o que foi dito antes.
- **Contador de Tokens:** Para monitorar custos.
- **Logs:** Salvar as perguntas e respostas em um arquivo para auditoria.