Inteligência Artificial 2

Aula 15: APIs e Prompts Básicos

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Inteligência Artificial 2

Sumário

1. O Paradigma da IA em Nuvem
2. A Teoria da Tokenização
3. Comunicação REST e Segurança
4. Engenharia de Prompts Básica
5. Desenvolvimento Resiliente e Desafios de API

Onde Paramos?

Nos Módulos I e II, construímos redes neurais **locais** — do Perceptron ao Transformer:

- Treino, validação, regularização, CNNs, RNNs, embeddings.
- Tudo rodando na sua máquina com dados tabulares ou textos curtos.

A Virada do Módulo III

Agora os modelos têm **bilhões de parâmetros** e vivem na **nuvem**.
Como consumir essa inteligência via API?

Objetivos da Sessão

Ao final desta aula, você será capaz de:

1. **Explicar** por que LLMs exigem infraestrutura de nuvem (custo de VRAM).
2. **Descrever** o processo de tokenização (BPE) e sua relação com custo.
3. **Configurar** credenciais seguras com `.env` e `.gitignore`.
4. **Ajustar** hiperparâmetros de inferência: Temperature, Top-p, Top-k.
5. **Consumir** uma API generativa (Gemini/OpenAI) via SDK Python.
6. **Implementar** tolerância a falhas com Exponential Backoff.

O Gargalo do Hardware na Era dos LLMs

- Modelos modernos de linguagem (LLMs) como GPT-4, Llama 3 (70B) e Gemini 1.5 possuem de dezenas de bilhões a mais de um trilhão de parâmetros.
- **Custo de Memória (VRAM):** Cada parâmetro em precisão FP16 (16 bits) consome 2 bytes. Um modelo de 70 bilhões de parâmetros necessita, estritamente para armazenamento dos pesos na GPU, de 140 GB de VRAM.
- **Custo Adicional de Inferência:** Além dos pesos, a alocação dinâmica para o *KV-Cache* (Key-Value Cache) durante a geração de sequências longas eleva a exigência de memória substancialmente.
- Hardware de consumo convencional não suporta tamanhas demandas, restringindo a pesquisa e aplicação de fronteira a servidores dedicados de altíssimo desempenho (HPC).

Clusterização e Economia de IA

- Uma única GPU NVIDIA H100 (80GB VRAM), padrão ouro atual para treinamento e inferência, possui um custo na casa das dezenas de milhares de dólares.
- Para rodar inferência de um modelo massivo (e.g., Mixtral 8x22B), exige-se comumente arranjos multi-GPU distribuídos (Tensor Parallelism).
- **CAPEX vs. OPEX:** Investimento de capital inicial (*CAPEX*) em infraestrutura de IA on-premise envolve rápida obsolescência e altos custos energéticos e de refrigeração.
- A transição para o modelo **AI-as-a-Service (AIaaS)** converteu esse custo de *hardware* contínuo em despesas operacionais (*OPEX*), democratizando o acesso ao estado-da-arte através de *cloud computing*.

CAPEX vs OPEX: O Modelo de Negócio da IA

A Conclusão para Engenheiros

Vocês não precisam comprar GPUs — basta consumir APIs. O custo saiu do **hardware** para o **software** (tokens consumidos).

Além da Temperatura: Top-p e Top-k

Top-k (Corte por Ranking)

Considerar apenas os **k tokens** mais prováveis e ignorar o restante.

Ex: $k = 5 \rightarrow$ só os 5 melhores candidatos competem.

Top-p (Nucleus Sampling)

Considerar tokens até que a **probabilidade acumulada** atinja p.

Ex: $p = 0.9 \rightarrow$ pega tokens até somar 90% de probabilidade.

Na Prática

Combine: $T = 0.7$, top_p $= 0.9$, top_k $= 40$ é um bom ponto de partida para chatbots. Para extração de dados, use $T = 0.1$, top_p $= 0.95$, top_k $= 1$.

Compreendendo Tokens na Camada Algorítmica

- Redes neurais não processam strings e caracteres brutos; processam tensores matemáticos.
- A etapa inicial de uma LLM é a conversão de texto não-estruturado para um vetor discreto de índices. A essa menor unidade de informação processável chamamos **Token**.
- Abordagens puramente baseadas em caracteres requerem um contexto demasiadamente profundo e têm dificuldade em capturar semântica morfológica.
- Abordagens puramente baseadas em palavras falham frente a variações morfológicas (conjugações, plurais) e palavras fora do vocabulário (*Out-Of-Vocabulary* - OOV).
- A solução padrão contemporânea é a **Tokenização de Sub-palavras** (Subword Tokenization).

Byte-Pair Encoding (BPE)

O BPE é um algoritmo de compressão de dados adaptado para NLP (usado pela OpenAI, LLaMA).

1. Inicializa o vocabulário baseando-se em bytes individuais ou caracteres.
2. Em cada iteração, identifica o par de tokens adjacentes mais frequente no *corpus* de treinamento e os funde em um novo token.
3. Repete o processo até atingir um tamanho de vocabulário alvo (frequentemente $V = 32.000$ a 128.000).

Outros algoritmos comuns incluem **WordPiece** (usado no BERT, do Google) e **Unigram Language Model**. Todos buscam maximizar o significado por token, fracionando termos raros e unificando termos comuns.

Precificação: Tokens como Moeda

- Em provedores de Nuvem, a cobrança é granular, efetuada tipicamente a cada 1.000 ($1K$) ou 1.000.000 ($1M$) de tokens processados.
- **Regra de Bolso (Inglês):** 1 token $\approx$ 4 caracteres $\approx$ 0.75 palavra. Em português, a eficiência da tokenização cai, resultando em mais tokens para a mesma quantidade de texto.
- **Input Tokens (Prompt):** Todo o contexto que você envia para a IA. O processamento (*Prefill*) de tokens de entrada é massivamente paralelizável nas GPUs e, portanto, mais barato.
- **Output Tokens (Generation):** Os tokens gerados pela IA de volta para você. O processamento é puramente sequencial (um token de cada vez) devido à natureza autorregressiva do modelo, tornando o custo e a latência de *Output Tokens* significativamente maiores.

Arquitetura RESTful e Payload JSON

- A interação com LLMs em nuvem ocorre majoritariamente através do protocolo HTTP, seguindo arquiteturas *REST* (Representational State Transfer).
- A aplicação cliente atua estritamente consumindo o serviço via internet.
- As requisições de geração de texto são mapeadas no método HTTP `POST`, uma vez que criam/processam recursos substanciais no servidor e transmitem informações extensas no corpo da requisição.
- O corpo (*Body*) da requisição é codificado em **JSON** (JavaScript Object Notation), encapsulando o *prompt*, metadados, e hiperparâmetros de inferência.

Estrutura de Requisição (Exemplo Agnóstico)

```
POST /v1/chat/completions HTTP/1.1
Host: api.provedor-llm.com
Authorization: Bearer sk-SUA_API_KEY_AQUI
Content-Type: application/json

{
    "model": "nome-do-modelo-v2",
    "messages": [
        {"role": "system", "content": "Voce e um assistente
            matematico."},
        {"role": "user", "content": "Explique o Teorema de Bayes."}
    ],
    "temperature": 0.7,
    "max_tokens": 512
}
```

Credenciais de Acesso: API Keys

- O *endpoint* na nuvem precisa autorizar e faturar a requisição correta. Isso é alcançado pela transmissão de um *Token* de autorização criptográfico no cabeçalho HTTP: a **API Key**.
- A API Key é o equivalente computacional da sua assinatura e cartão de crédito. É estritamente confidencial.
- A falha fatal de desenvolvedores inexperientes é realizar o *hardcoding* da API Key (escrever diretamente no arquivo de código fonte `.py` ou `.js`).
- Quando código *hardcoded* é versionado no GitHub, bots raspadores varrem repositórios públicos em milissegundos, furtando as credenciais para cometer fraudes computacionais (minerando criptomoedas ou abusando de APIs), gerando faturas massivas para a vítima.

Isolamento de Credenciais com `.env`

- A melhor prática da indústria é extrair configurações e credenciais sensíveis do código fonte.
- Utiliza-se um arquivo de texto simples na raiz do projeto, rotineiramente chamado de `.env`.

Exemplo de Arquivo `.env`

```
OPENAI_API_KEY=sk-abc123def456ghi789jkl
GOOGLE_API_KEY=AIzaSyB123CdeFghIjkLMnOpQrsTuvWx
ENVIRONMENT=development
```

Crucial: Este arquivo DEVE ser incluído no arquivo `.gitignore` para que o sistema de controle de versão (Git) o ignore completamente e jamais o faça *upload* para a nuvem.

Implementação Prática de `.env` em Python

No ecossistema Python, usamos a biblioteca `python-dotenv` para injetar o conteúdo do arquivo local no escopo de variáveis de ambiente do SO (`os.environ`), tornando-as acessíveis dinamicamente.

```python
import os
from dotenv import load_dotenv

# Carrega variaveis do arquivo .env para o sistema
load_dotenv()

# Recupera de forma segura da memoria
minha_chave = os.getenv("GOOGLE_API_KEY")

if minha_chave is None:
    raise ValueError("A chave da API nao foi encontrada no ambiente!
        ")

# Proceder com a invocacao da API usando a variavel 'minha_chave'
```

A Anatomia de um Prompt de Chat

Em APIs modernas, os prompts não são apenas *strings* solitárias, mas um array estruturado de mensagens, cada qual com um papel (Role) distinto:

- **System Prompt (Instrução de Sistema):** Define o comportamento macro, a personalidade e as restrições globais do modelo antes da interação. Funciona como a "metainstrução" de contexto.
- **User Prompt (Usuário):** O comando, pergunta ou instrução direta oriunda do operador humano ou da aplicação consumidora.
- **Assistant Prompt (Assistente):** As respostas prévias geradas pela própria IA no histórico conversacional, injetadas para preservar a memória e coesão das mensagens, já que a API é *Stateless* (não guarda estado entre as requisições).

Hiperparâmetros de Inferência: Temperatura (T)

O parâmetro "Temperatura" governa a aleatoriedade estocástica da geração.
Matematicamente, ajusta os valores brutos de saída (*logits* z_i) da última camada da rede antes de aplicar a função *Softmax*.
A distribuição de probabilidade para o próximo token x_i é dada por:

$$P(x_i) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)} \tag{1}$$

- $T = 1.0$: A *Softmax* padrão, reflete os pesos reais aprendidos.
- $T < 1.0$ (ex: 0.1): Distribuição aguçada (*Sharpening*). Tokens de alta probabilidade se tornam dominantes, reduzindo drasticamente a criatividade (Ideal para extração de dados estritos).
- $T > 1.0$ (ex: 1.5): Distribuição achatada (*Smoothing*). Aumenta a amostragem na cauda longa do vocabulário, elevando alucinações mas gerando textos criativos.

Hiperparâmetros de Inferência: Nucleus Sampling ($Top - p$) e $Top - k$

Além da temperatura, limitamos o espaço de busca dos tokens para evitar "trilhas aleatórias":

- $Top - k$**:** Apenas os k tokens com maiores probabilidades são considerados na amostragem estocástica do próximo passo (por exemplo, os 40 mais prováveis). Corta caudas irrelevantes absolutas.
- $Top - p$ **(Nucleus Sampling):** Os tokens são ordenados por probabilidade e somados sequencialmente até que a probabilidade cumulativa alcance p (ex: 0.90). O espaço de amostragem varia dinamicamente dependendo de quão "certo" o modelo está de suas escolhas. Se 2 tokens já somam 0.9, apenas eles compõem o núcleo.

Latência e Streaming

- Por gerar texto de maneira autorregressiva ($O(N)$ inferências para produzir N tokens), requisições LLM são inerentemente lentas (geralmente levando de vários segundos a dezenas de segundos para *outputs* vastos).
- Durante requisições REST tradicionais (bloqueantes), a conexão HTTP ficaria estagnada até o modelo processar o último ponto final.
- Para sanar isso, APIs de IA suportam **Server-Sent Events (SSE)** (ou "*Streaming*").
- Com o Streaming ativado, o servidor devolve um pacote de resposta (chunck) imediato para cada token (ou grupo de tokens) à medida que são gerados na GPU, possibilitando a criação de interfaces de Chat com resposta em tempo-real.

Rate Limits (Limites de Requisição)

Sistemas de IAaaS aplicam controles rigorosos de uso para evitar o esgotamento dos datacenters:

- **RPM (Requests Per Minute):** Quantidade máxima de chamadas HTTP (POST) que podem ser feitas à API por minuto.
- **TPM (Tokens Per Minute):** Limita a soma de *Input Tokens* e *Output Tokens* por minuto, evitando que uma única requisição gigante drene os recursos.
- Ao ultrapassar essas cotas, o servidor recusa requisições futuras temporariamente retornando o *Status Code* HTTP **429 (Too Many Requests)**.

Tolerância a Falhas: Exponential Backoff

Em engenharia de dados aplicada a LLMs em massa (processando milhares de documentos via API), precisamos tratar proativamente o erro 429.

A Estratégia do Exponential Backoff

Quando uma requisição falha, o script não entra em pânico. Ele pausa a execução (*sleep*) por um intervalo que dobra sucessivamente (2^c) e tenta novamente, aliviando o servidor.

Exemplo prático de fluxo de tentativa (*Retry Policy*):

1. Falha 1 (Erro 429) $\rightarrow$ Pausa de 2s $\rightarrow$ Nova tentativa
2. Falha 2 (Erro 429) $\rightarrow$ Pausa de 4s $\rightarrow$ Nova tentativa
3. Falha 3 (Erro 429) $\rightarrow$ Pausa de 8s $\rightarrow$ Nova tentativa

Orquestração via SDK vs REST Puro

Na prática, as chamadas REST podem ser feitas nativamente via biblioteca `requests` (Python), mas o código se torna verboso. Bibliotecas de terceiros (SDKs oficiais) envelopam essa abstração.

```python
import google.generativeai as genai
import os

genai.configure(api_key=os.getenv("GEMINI_API_KEY"))

model = genai.GenerativeModel('gemini-1.5-pro-latest')

# O SDK cria o objeto JSON e efetua o POST REST internamente
resposta = model.generate_content("Qual e o cerne da IA em nuvem?",
    generation_config=genai.types.GenerationConfig(
        temperature=0.4,
        max_output_tokens=100
    )
)
```

Pipeline Completo: Do Prompt à Resposta

Cada Seta = Ponto de Falha

O SDK pode ter bug, o HTTP pode dar timeout, a GPU pode estar sobrecarregada (429), e a resposta pode ser truncada. **Resiliência** é obrigatória em produção.

Exemplo Visual: Tokenização BPE Passo a Passo

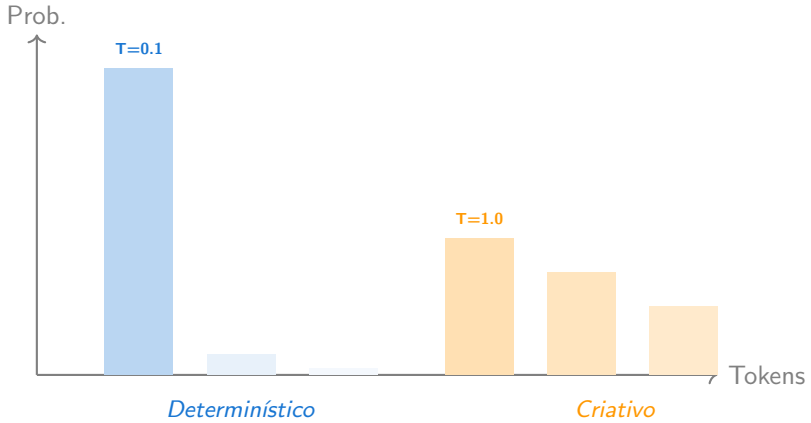
Frase: "Inteligência Artificial é fascinante"

1. **Bytes iniciais:** I | n | t | e | l | i | g | ê | n | c | i | a | ...
2. **Merge 1:** "in" (par mais frequente) → **In**t | el | ig | ên | ci | a
3. **Merge N:** "Intel" | "igência" | "Art" | "ificial" | "é" | "fasc" | "inante"

Inglês (Eficiente)	Português (Caro!)
"Artificial Intelligence" = **2 tokens**	"Inteligência Artificial" = **4-5 tokens**

Conclusão: o mesmo conceito custa 2x mais em português. Isso impacta diretamente o custo financeiro.

Visualizando a Temperatura



Regra Prática

$T \leq 0.3$: extração de dados, JSON. $T \approx 0.7$: chatbot. $T \geq 1.0$: brainstorming, poesia.

Tabela de Custos: Quanto Custa Usar IA?

Modelo	Input (/1M tok)	Output (/1M tok)	Contexto
GPT-4o	$2.50	$10.00	128K
GPT-4o mini	$0.15	$0.60	128K
Gemini 1.5 Pro	$1.25	$5.00	1M
Gemini 1.5 Flash	$0.075	$0.30	1M
Llama 3 (local)	Grátis	Grátis	8K

Output é 4x Mais Caro que Input!

Porque o input é processado em paralelo (Prefill), mas o output é gerado token a token (autorregressivo). **Controlar** `max_tokens` é essencial para o orçamento.

O Perigo do Hardcoding: Caso Real

O Erro

```
api_key = "sk-abc123..."
git add .  && git push
```

- Bots raspam o GitHub em **menos de 30 segundos**.
- A chave é usada para minerar criptomoedas.
- A vítima recebe uma **fatura de $10.000+**.

A Defesa

```
.env: API_KEY=sk-abc123
.gitignore: .env
os.getenv("API_KEY")
```

- Chave nunca entra no Git.
- Cada ambiente tem seu `.env`.
- CI/CD injeta via **Variables** do GitLab.

Streaming (SSE): Resposta em Tempo Real

Sem Streaming

1. Usuário envia pergunta
2. **Espera 15 segundos** (tela em branco)
3. Resposta completa aparece de uma vez

× UX péssima

Com Streaming (SSE)

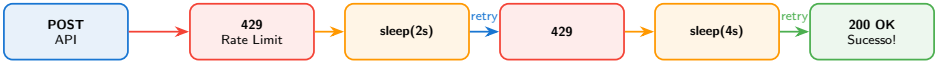
1. Usuário envia pergunta
2. Tokens aparecem **um a um** (<1s)
3. Efeito "máquina de escrever"

✓ UX do ChatGPT

Na Prática

Ative streaming com `stream=True` no SDK. O servidor envia **Server-Sent Events** (SSE) — cada chunk é um pacote HTTP parcial.

Fluxo de Retry com Exponential Backoff



Fórmula

$\text{wait} = \text{base}^{\text{tentativa}} + \text{jitter} \quad (\text{ex: } 2^1 = 2s,\ 2^2 = 4s,\ 2^3 = 8s)$

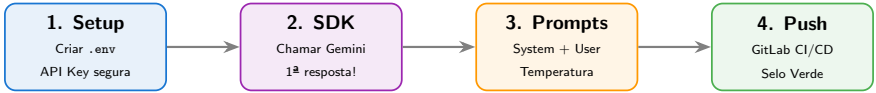
Recap — O Que Levamos da Aula 15

1. LLMs vivem na **nuvem** — VRAM local não comporta bilhões de parâmetros.
2. O **Token** é a unidade atômica de custo e processamento.
3. **API Keys** nunca entram no Git — `.env` + `.gitignore` é obrigatório.
4. **Temperatura** controla criatividade vs determinismo.
5. **Rate Limits** (429) exigem Exponential Backoff.
6. **SDKs** (genai, openai) abstraem o HTTP cru.

Conexão com Engenharia de Software

Tudo que vocês aprenderam sobre REST APIs, JSON, segurança e resiliência em disciplinas anteriores converge **exatamente aqui**.

Roteiro do Laboratório 15



Meta do Lab

Ao final, vocês terão um script Python que consome a API do Gemini com `.env` seguro, System Prompt configurado e temperatura ajustável. O **Selo Verde** no GitLab valida a entrega.

Dúvidas?

Email: alessio@cefetmg.br
Aula: Segunda e Quarta-Feira, 7h as 8:40h (Sala 39)
WhatsApp: Dúvidas rápidas

Lembrete

Confira sua Issue no GitLab!

Próxima aula: JSON Mode & Schemas

100% da Aula 15 Concluída!