Inteligência Artificial 2

Deploy de IA: FastAPI e Docker

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

29 de setembro de 2026

Onde Paramos?

Na sessão anterior, vocês enfrentaram o
textbfmundo real dos dados desbalanceados:

- **Acurácia mente:** 95% de acurácia com 0 fraudes detectadas.
- **Precision, Recall e F1-Score:** as métricas que revelam a verdade.
- **Matriz de Confusão:** TP, TN, FP e FN decompostos visualmente.
- **Serialização:** salvaram o modelo (`.keras`) e o scaler (`.joblib`).

A Pergunta de Hoje

Seu modelo está salvo em disco.
E agora? **Como alguém usa ele?**

Objetivos da Sessão

- **O Gap entre Notebook e Produção:** Por que um `.keras` no disco não é um produto.
- **FastAPI:** Criar uma API REST que recebe JSON e retorna previsões.
- **Pydantic:** Validar dados na fronteira da rede antes de tocar no modelo.
- **Swagger UI:** Testar a API interativamente no navegador.
- **Docker:** Empacotar tudo em um container portátil e reprodutível.
- **Ciclo completo:** Do treinamento ao `curl` em produção.

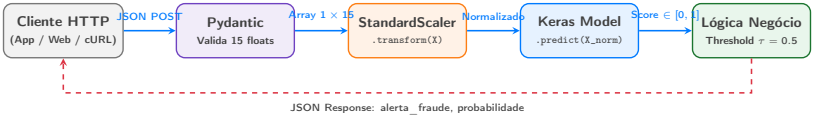
O Problema: O Modelo Preso no Notebook

Sem API: ninguém fora do notebook consegue usar o modelo.

Regra de Ouro da Engenharia de IA

Um modelo que não é **acessível via rede** tem **impacto operacional nulo**.
Precisamos transformá-lo em um **microsserviço**.

A Solução: Arquitetura de Inferência



- **5 estágios** entre o clique do usuário e a resposta da IA.
- Cada estágio tem uma responsabilidade única (princípio da *Separation of Concerns*).

A Armadilha do Escalonador Esquecido

SEM Scaler
Valor bruto: R$ 1.500,00
Rede recebe magnitude 10^3

$\rightarrow$ **Previsão ERRADA**

COM Scaler
Valor normalizado: $z = 0.42$
Mesma escala do treino

$\rightarrow$ **Previsão CORRETA**

Atenção de Engenharia

O modelo foi treinado com dados z-score ($\mu \approx 0, \sigma \approx 1$).
Enviar dados brutos gera previsões **erráticas e sem sentido**.
O escalonador é parte indissociável da inferência.

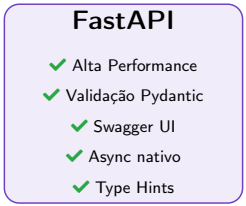
O que é uma API REST?

Verbo HTTP	Ação	Exemplo
`GET`	Ler/consultar	`GET /health`
`POST`	Enviar dados	`POST /predict`
`PUT`	Atualizar	`PUT /model/v2`
`DELETE`	Remover	`DELETE /cache`

FastAPI: O Framework de IA em Python

Por que FastAPI?

- **Performance:** Baseado em Starlette (ASGI); a latência e a vazão dependem da aplicação, infraestrutura e carga.
- **Validação automática:** Pydantic valida tipos e formatos sem código extra.
- **Documentação grátis:** Swagger UI gerada automaticamente em `/docs`.
- **Python puro:** Mesma linguagem do TensorFlow/Keras.



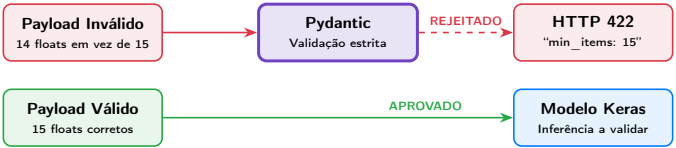
Pydantic: O Guarda-Costas dos Dados

Sem Pydantic:

- Cliente envia `"features": "banana"`.
- A rede neural recebe lixo.
- `ValueError` derruba o servidor.
- **Inaceitável em produção.**

Com Pydantic:

- Cliente envia `"features": "banana"`.
- Pydantic rejeita **na fronteira**.
- Retorna `HTTP 422` com erro claro.
- **Servidor intacto.**



Código: Contratos de Dados (Pydantic)

Os *schemas* definem o contrato estrito entre cliente e servidor:

```python
from pydantic import BaseModel, Field

class TransacaoInput(BaseModel):
    features: list[float] = Field(
        ...,                # obrigatorio
        min_items=15,        # minimo 15 numeros
        max_items=15,        # maximo 15 numeros
        description="Vetor com 15 caracteristicas"
    )

class PredicaoOutput(BaseModel):
    probabilidade: float    # score entre 0 e 1
    classe: int             # 0 ou 1
    alerta_fraude: bool     # True/False
    status: str             # "APROVADA" / "BLOQUEADA"
```

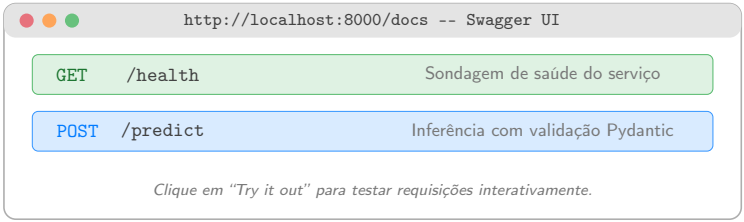
Código: Criando a API com FastAPI

```
from fastapi import FastAPI
import joblib, numpy as np
from tensorflow.keras.models import load_model

app = FastAPI(title="NeoBank Sentinela")

@app.on_event("startup")
def carregar_artefatos():
    global modelo, scaler
    modelo = load_model("modelo_fraude.keras")
    scaler = joblib.load("scaler_fraude.joblib")

@app.get("/health")
def health_check():
    return {"status": "healthy",
            "modelo_carregado": modelo is not None}

@app.post("/predict",
          response_model=PredicaoOutput)
```

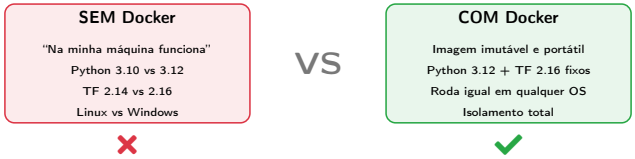
Executando Localmente com Uvicorn

O servidor ASGI de alta performance:

```
uvicorn src.api:app --host 0.0.0.0 --port 8000 --reload
```



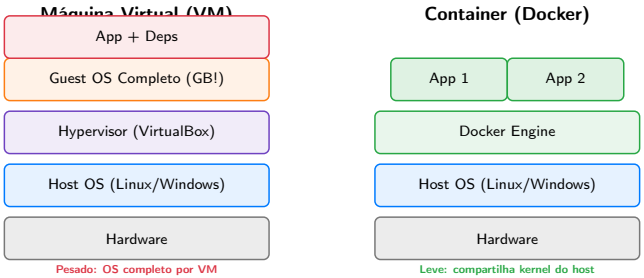
O que é Docker? E por que precisamos dele?



Analogia

Docker = **caixa de transporte marítimo** (container).
Independente do navio (OS), a carga (sua app) chega intacta e idêntica.

Container vs Máquina Virtual



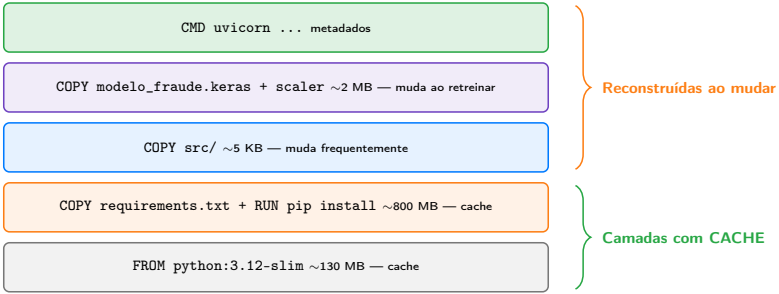
- **VM:** carrega um OS inteiro (GBs) — lenta para iniciar.
- **Container:** compartilha o kernel do host — inicia em **segundos** e pesa MBs.

Anatomia de um Dockerfile

Cada linha é uma *instrução* que cria uma **camada** (layer) da imagem:

```
FROM python:3.12-slim
ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY src/ ./src/
COPY modelo_fraude.keras .
COPY scaler_fraude.joblib .
EXPOSE 8000
CMD ["uvicorn", "src.api:app", "--host", "0.0.0.0", "--port", "8000"]
```

Como as Camadas (Layers) Funcionam



- Docker faz **cache** das camadas que não mudaram.
- Por isso copiamos `requirements.txt` **antes** do código-fonte.
- Rebuilds ficam **muito mais rápidos** (só recompila o que mudou).

Comandos Docker: Build, Run, Inspect

```
# 1. Build da imagem
docker build -t neobank-fraude:v1 .

# 2. Rodar em background
docker run -d -p 8000:8000 --name sentinela neobank-fraude:v1

# 3. Verificar status e logs
docker ps
docker logs sentinela

# 4. Parar e remover
docker stop sentinela && docker rm sentinela
```

- `-p 8000:8000`: mapeia porta do host → container.
- `-d`: background (daemon mode).
- Acesse `http://localhost:8000/docs`!

Testando em Produção com cURL

Requisição direta ao microsserviço via terminal:

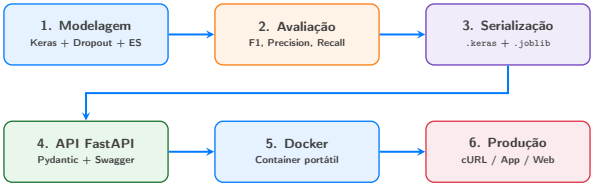
```
curl -X POST "http://localhost:8000/predict" \
     -H "Content-Type: application/json" \
     -d '{"features": [1250.0, 2.5, 0.8, -1.2, 0.05,
          3.1, -0.4, 1.8, 0.9, -0.2,
          0.1, 0.4, -0.05, 0.12, -0.3]}'
```

Resposta:

```
{"probabilidade": 0.0412, "classe": 0,
 "alerta_fraude": false, "status": "APROVADA"}
```

Sua IA está operando como um microsserviço real!

O Ciclo Completo: De Notebook a Produção



A Grande Lição

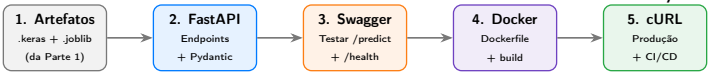
Vocês dominaram a trilha completa: do cálculo manual do Perceptron até o **deploy conteinerizado de microsserviços de IA**.

Recapitulação — O que Levar Desta Sessão

1. **Modelo no disco $\neq$ modelo em produção.** É preciso uma **API** para torná-lo acessível.
2. **FastAPI** cria endpoints REST de alta performance com validação automática via **Pydantic**.
3. O **Scaler é inseparável** do modelo — sem ele, dados brutos geram previsões erráticas.
4. **Swagger UI** (`/docs`) permite testar a API interativamente sem escrever código.
5. **Docker** encapsula código, runtime e dependências em uma imagem imutável e portátil.
6. `docker build` $\rightarrow$ `docker run` $\rightarrow$ `curl /predict` = **IA em produção**.

Roteiro do Laboratório 08 (Parte 2)

Missão: Transformar o modelo serializado em um microsserviço Docker.



Lab Passo 1: Verificar os Artefatos da Parte 1

Antes de construir a API, confirme que os artefatos da Sessão 1 existem:

- `modelo_fraude.keras` — topologia + pesos da rede neural.
- `scaler_fraude.joblib` — média (μ) e desvio padrão (σ) do treino.
- Se não existirem: **volte à Parte 1** e execute o treinamento primeiro.

Verificação Rápida no Terminal

```
ls -lh modelo_fraude.keras scaler_fraude.joblib
```

Ambos devem existir e ter tamanho > 0 bytes.

Lab Passo 2: Criar o src/api.py

```
from fastapi import FastAPI
from pydantic import BaseModel, Field
import numpy as np, joblib
from tensorflow.keras.models import load_model

app = FastAPI(title="NeoBank Sentinela")

@app.on_event("startup")
def carregar():
    global modelo, scaler
    modelo = load_model("modelo_fraude.keras")
    scaler = joblib.load("scaler_fraude.joblib")

@app.post("/predict")
def predict(t: TransacaoInput):
    X = scaler.transform(
        np.array(t.features).reshape(1, -1))
    score = float(modelo.predict(X, verbose=0)[0][0])
    return {"probabilidade": round(score, 4),
```

Lab Passo 3: Testar com Swagger UI

```
uvicorn src.api:app --host 0.0.0.0 --port 8000 --reload
```

1. Acessem `http://localhost:8000/docs`.
2. Cliquem em **POST /predict** → **Try it out**.
3. Enviem 15 números e verifiquem `HTTP 200`.
4. **Teste:** removam 1 número (14 features) → esperem `HTTP 422`.

Se o 422 apareceu

O Pydantic está **protegendo** sua IA contra dados malformados. Missão cumprida!

Lab Passo 4: Criar o Dockerfile e Rodar

Criem o `Dockerfile` na raiz do projeto e construam a imagem:

```
# Construir
docker build -t neobank-fraude:v1 .

# Rodar em background
docker run -d -p 8000:8000 --name sentinela \
    neobank-fraude:v1

# Verificar
docker ps
docker logs sentinela
```

Se a saída mostrar `Uvicorn running on 0.0.0.0:8000`, o container está operacional. Acessem `http://localhost:8000/docs` — funciona **idêntico**, mas agora dentro do container.

Lab Passo 5: Auditoria via cURL e Entrega

Testem a inferência via terminal e enviem ao GitLab:

```
curl -X POST "http://localhost:8000/predict" \
     -H "Content-Type: application/json" \
     -d '{"features": [1250.0, 2.5, 0.8, -1.2,
          0.05, 3.1, -0.4, 1.8, 0.9, -0.2,
          0.1, 0.4, -0.05, 0.12, -0.3]}'
```

Entrega:

```
git add src/api.py Dockerfile .dockerignore requirements.txt
git commit -m "Lab 08 (Parte 2): API FastAPI + Docker"
git push origin main
```

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: Módulo 2: NLP e Transformers

100% da Aula 8 Concluída!