Guia Prático: Lab 08 (Parte 2) — Deploy de IA com FastAPI e Docker

Inteligência Artificial II (IA II) — CEFET-MG

Prof. Aléssio Miranda Júnior

2026-01-01

Índice

1 Objetivo da Atividade

Na **Parte 1** deste laboratório, você treinou um classificador de fraudes com dados desbalanceados, avaliou-o com métricas críticas (Precision, Recall, F1-Score) e serializou os artefatos (`modelo_fraude.keras` + `scaler_fraude.joblib`).

Agora, na **Parte 2**, você transformará esses arquivos estáticos em um **microsserviço de IA operacional**. Ao final desta sessão, seu modelo estará respondendo requisições HTTP reais, validado por um gateway Pydantic e encapsulado em um container Docker portátil.

O código inicial que você recebeu no arquivo `src/api_starter.py` contém um esqueleto funcional com TODOs guiados. Seu objetivo é completar as seções marcadas e chegar ao deploy completo.

---

2 Dinâmica de Sala de Aula (60 minutos)

Etapa	Duração	Atividade Principal
Etapa 1	**05 min**	Verificar os artefatos da Parte 1 (`modelo_fraude.keras` + `scaler_fraude.joblib`).
Etapa 2	**15 min**	Completar o `src/api.py` com os schemas Pydantic e endpoints FastAPI.
Etapa 3	**10 min**	Executar localmente com Uvicorn e testar no Swagger UI (`/docs`).
Etapa 4	**10 min**	Criar o `Dockerfile`, `.dockerignore` e `requirements.txt`.
Etapa 5	**10 min**	Build da imagem Docker, execução do container e auditoria com `curl`.
Etapa 6	**10 min**	Teste de robustez (HTTP 422 com payload inválido) e submissão no GitLab.

---

3 Passo a Passo do Laboratório

3.1 Etapa 1: Verificação dos Artefatos (Pré-requisito)

Antes de construir a API, confirme que os artefatos da Parte 1 existem na raiz do projeto:

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

Ambos devem existir e ter tamanho > 0 bytes. Se não existirem, **volte à Parte 1** e execute o treinamento completo primeiro.

> **Por que o Scaler é obrigatório?** O modelo foi treinado com dados normalizados (z-score: $0,\ 1$). Se você enviar o valor bruto de uma transação (ex: R$ 1.500,00), a rede neural receberá uma magnitude centenas de vezes maior que a esperada, resultando em previsões completamente erráticas. O escalonador **faz parte indissociável da inferência**.

3.2 Etapa 2: Construindo a API REST com FastAPI

Abra o arquivo `src/api_starter.py`. Ele contém a estrutura base com TODOs marcados. Complete as seções abaixo:

3.2.1 2.1 Schemas Pydantic (Contratos de Dados)

Localize o TODO `# TODO 1: Completar o schema de saída` e adicione os campos:

```
class PredicaoOutput(BaseModel):
    probabilidade: float
    classe: int
    alerta_fraude: bool
    status: str
```

3.2.2 2.2 Endpoint de Predição

Localize o TODO `# TODO 2: Implementar a lógica de inferência` e complete:

```
@app.post("/predict", response_model=PredicaoOutput, tags=["Inferência"])
def predict(transacao: TransacaoInput):
    # Converter entrada para array 2D (1 amostra × 15 features)
    X_raw = np.array(transacao.features).reshape(1, -1)

    #  NUNCA pule esta etapa! O scaler normaliza os dados
    # com os mesmos  e  usados no treinamento
    X_norm = scaler.transform(X_raw)

    # Inferência da rede neural
    score = float(modelo.predict(X_norm, verbose=0)[0][0])
    classe = 1 if score > 0.5 else 0

    return PredicaoOutput(
        probabilidade=round(score, 4),
        classe=classe,
        alerta_fraude=bool(classe == 1),
        status="BLOQUEADA" if classe == 1 else "APROVADA"
    )
```

3.2.3 2.3 (Opcional) Tratamento de Erro Robusto

Adicione tratamento de exceção para proteger o servidor contra falhas internas:

```
from fastapi import HTTPException

# Dentro da função predict(), envolva a lógica com try/except:
try:
    # ... lógica de inferência ...
except Exception as e:
    raise HTTPException(
        status_code=500,
        detail=f"Falha interna de inferência: {str(e)}"
    )
```

3.3 Etapa 3: Execução Local e Validação com Swagger UI

Inicie o servidor de desenvolvimento com recarga automática:

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

Abra o navegador em **http://localhost:8000/docs** — o Swagger UI é gerado automaticamente!

3.3.1 Checklist de Validação:

- ☐ **GET /health** → Retorna `{"status": "healthy", "modelo_carregado": true}`
- ☐ **POST /predict** com 15 números → Retorna HTTP 200 com `probabilidade`, `classe`, `alerta_fraude`, `status`
- ☐ **POST /predict** com 14 números (payload inválido) → Retorna **HTTP 422** (Pydantic rejeita na fronteira!)
- ☐ **POST /predict** com `"features": "banana"` → Retorna **HTTP 422** (tipo errado)

> **O HTTP 422 é seu amigo!** Significa que o Pydantic está protegendo sua IA contra dados malformados sem derrubar o servidor.

3.4 Etapa 4: Conteinerização com Docker

3.4.1 4.1 Criar o `Dockerfile` na raiz do projeto

```
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"]
```

3.4.2 4.2 Criar o `.dockerignore`

```
__pycache__/
*.pyc
.git
.venv
tests/
.pytest_cache
*.png
```

3.4.3 4.3 Criar/Verificar `requirements.txt`

```
fastapi>=0.110.0
uvicorn>=0.28.0
pydantic>=2.6.0
tensorflow-cpu>=2.16.0
scikit-learn>=1.4.0
joblib>=1.3.0
numpy>=1.26.0
```

3.5 Etapa 5: Build, Run e Auditoria

3.5.1 Build da imagem

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

> **Dica de performance:** O Docker faz cache das camadas. Se você só alterou `src/`, apenas a camada `COPY src/` será reconstruída — o `pip install` não será re-executado.

3.5.2 Executar o container

```
docker run -d -p 8000:8000 --name sentinela neobank-fraude:v1
```

3.5.3 Verificar status e logs

```
docker ps                    # Lista containers em execução
docker logs sentinela        # Logs do Uvicorn dentro do container
```

Se a saída mostrar `Uvicorn running on http://0.0.0.0:8000`, o container está operacional.

3.5.4 Auditoria com cURL

```
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 esperada:

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

3.5.5 Parar e limpar (quando terminar)

```
docker stop sentinela
docker rm sentinela
```

3.6 Etapa 6: Teste de Robustez e Submissão

3.6.1 Teste de Resiliência

O que acontece se um cliente enviar 14 características em vez de 15? Execute o teste destrutivo:

```
curl -X POST "http://localhost:8000/predict" \
     -H "Content-Type: application/json" \
     -d '{"features": [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0]}'
```

Resposta esperada: **HTTP 422** com mensagem clara: `"ensure this value has at least 15 items"`.
O servidor permanece no ar — sem crash, sem stack trace.

3.6.2 Submissão no GitLab

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

---

4 O Ciclo Completo

Ao final deste laboratório, você terá executado a trilha completa da Engenharia de IA:

```
1. Modelagem (Keras) → 2. Avaliação (F1, Precision) → 3. Serialização (.keras + .joblib)
      ↓
4. API REST (FastAPI + Pydantic) → 5. Swagger UI (/docs) → 6. Container (Docker) → 7. Produção (c
```

Parabéns! Você acaba de fechar o Módulo 1 dominando desde o Perceptron solitário até o deploy conteinerizado de microsserviços de IA.

---

5 Conexão com o TP1

O fluxo que você executou hoje (Parte 1 + Parte 2) serve como **template arquitetural obrigatório** para o Trabalho Prático 1:

- Coleta e exploração de dataset real (Kaggle ou WOKDEX)
- Treinamento com prevenção de sobreajuste (Dropout + EarlyStopping)
- Avaliação crítica com métricas adequadas ao desbalanceamento
- Exposição via API FastAPI documentada + container Docker