Lab. 04: Ecossistema Keras

Aléssio Miranda Júnior

Agosto - 2026/2

Lab. 04: Ecossistema Keras

⋆ Objetivo do Laboratório

Nos Labs 01–03, você construiu neurônios, redes MLP e loops de treinamento com NumPy puro — traduzindo cada equação em código artesanal. Hoje você abandona a oficina e entra na **fábrica**: o ecossistema Keras/TensorFlow. A missão é construir o **mesmo pipeline** de Machine Learning, mas agora com ferramentas industriais — `pandas` para dados, `StandardScaler` para normalização e `Sequential` para a rede. Ao final, a pergunta que o Lab responde é: *"se o Keras faz tudo sozinho, por que precisei aprender na mão?"*

1 Parte 1: O Problema D — Estilo Maratona

Problema D: A Previsão de Churn Bancário

Contexto: O banco *DigitalBank* está perdendo clientes a um ritmo alarmante. A diretoria contratou sua equipe de Engenharia de IA para construir um modelo preditivo que identifique, **antes** do cancelamento, quais clientes têm maior risco de abandonar o banco (*churn*). Os dados históricos já foram coletados — sua missão é construir o **pipeline completo** de classificação binária.

Modelo de Entrada (X): Vetor com 2 features numéricas por cliente:

- x_1: Idade (inteiro entre 18 e 70)
- x_2: Salário Mensal em R$ (inteiro entre 2.000 e 350.000)

Regra de Negócio (Rótulo): Um cliente cancelou ($y = 1$) se, e somente se, ambas as condições forem satisfeitas simultaneamente:

$$y = \begin{cases} 1 & \text{se } x_1 > 40 \textbf{ E } x_2 > 50.000 \\ 0 & \text{caso contrário} \end{cases}$$

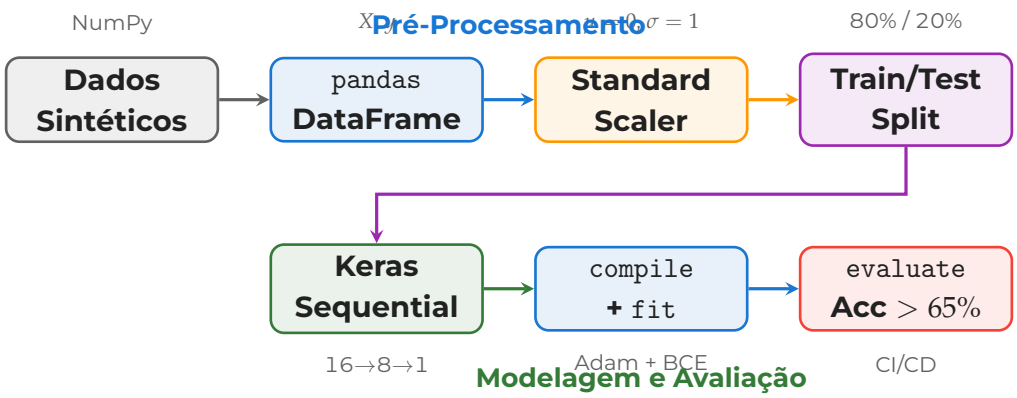
Modelo de Saída ($\hat{y}$): Probabilidade de churn via Sigmoid $\in (0, 1)$:

- $\hat{y} > 0.5 \quad \rightarrow$ Cliente em risco de churn
- $\hat{y} \leq 0.5 \quad \rightarrow$ Cliente seguro

Critério de Aprovação: Acurácia no conjunto de teste $> 65\%$.

A. O Pipeline Industrial

Diferente dos Labs anteriores onde cada componente era programado à mão, agora usaremos ferramentas especializadas para cada etapa. Observe o pipeline completo que você vai construir:



• Teoria: Do Artesanal ao Industrial

Compare mentalmente com os Labs anteriores: no Lab 01, você escreveu a função degrau; no Lab 02, a classe `CamadaDensa` com forward pass; no Lab 03, o loop de épocas com update rule. Agora, a linha `model.fit(X, y, epochs=80)` executa **tudo isso internamente** — Forward Pass, Loss, Backpropagation e Update Rule — com aceleração em GPU. O que mudou não é a matemática, é a **escala**.

2 Parte 2: Gerando e Estruturando os Dados (8 min)

Antes de codificarmos, certifique-se de que seu arquivo `requirements.txt` está atualizado. Como o ecossistema Python evolui rápido, misturar versões pode causar quebras no ambiente do GitLab (especialmente em compatibilidade com o `numpy>=2.0.0`).

△ Importante: Atualize o requirements.txt (Recomendações)

Abra o arquivo `requirements.txt` do seu projeto e adicione (ou atualize) as bibliotecas industriais usadas neste lab. Cuidado para não deixar versões antigas duplicadas:

- `numpy>=2.0.0` (Obrigatório para álgebra linear)
- `pandas>=3.0.0` (Obrigatório para manipulação de CSVs)
- `scikit-learn>=1.5.0` (Obrigatório para o StandardScaler e Split)
- `tensorflow-cpu>=2.17.0` (Obrigatório — usamos a versão CPU para evitar warnings de driver CUDA em máquinas convencionais)

Após atualizar o arquivo, não se esqueça de rodar: `pip install -r requirements.txt`.

Como o GitLab executará seu script em um container isolado sem internet, criaremos dados sintéticos em memória que simulam o problema bancário de churn. Crie o arquivo `src/churn_keras.py` e adicione a fundação:

```python
import numpy as np
import pandas as pd
import os

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'  # Silencia logs em C++ do TF
os.environ['TF_ENABLE_ONEDNN_OPTS'] = '0' # Silencia warnings do oneDNN

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input

# Fixando a semente para reprodutibilidade
np.random.seed(42)

# Gerando 500 clientes fictícios
idades = np.random.randint(18, 70, size=500)
salarios = np.random.randint(2000, 350000, size=500)

# Regra de churn: idade > 40 E salário > 50k => cancelou (1)
alvo = np.where((idades > 40) & (salarios > 50000), 1, 0)

# Organizando em DataFrame pandas
df = pd.DataFrame({
    'idade': idades,
    'salario': salarios,
    'churn': alvo
})

# Separando Features (X) e Target (y)
X = df[['idade', 'salario']].values
y = df['churn'].values
```

△ Importante: Por que Dados Sintéticos?

Em ambiente CI/CD, o script roda em um container sem acesso à internet. Gerar dados em memória garante reprodutibilidade e independência. Em projetos reais, você substituiria este bloco por `pd.read_csv('dados_reais.csv')`. A semente `np.random.seed(42)` garante que todos os alunos geram exatamente os mesmos dados.

B. Verificação: Distribuição dos Dados

Antes de continuar, verifique a distribuição de classes no seu dataset:

Estatística	Valor	Significado
Total de clientes	500	Gerados com `np.random`
Clientes sem churn ($y = 0$)	≈ 350	Idade ≤ 40 OU Salário ≤ 50k
Clientes com churn ($y = 1$)	≈ 150	Idade > 40 E Salário > 50k
Features	2	Idade, Salário

3 Parte 3: Dividindo e Normalizando (10 min)

C. Train/Test Split

Separe os dados em treino (80%) e teste (20%). Adicione ao seu arquivo:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)
```

D. StandardScaler — A Normalização Obrigatória

Agora aplique a lei áurea do pré-processamento. Lembre-se do vocabulário do *scikit-learn* que vimos nos slides: `fit` é o verbo **aprender** (μ e σ), e `transform` é o verbo **aplicar**. Observe a diferença crucial:

```
scaler = StandardScaler()

# 1. fit_transform no TREINO: Aprende a escala e ja aplica
X_train_escalonado = scaler.fit_transform(X_train)

# 2. transform no TESTE: Aplica a MESMA escala (sem aprender de novo)
X_test_escalonado = scaler.transform(X_test)
```

△ Importante: Erro Grave — Data Leakage

Se você usar `fit_transform()` no teste, o scaler vai "esquecer"a escala do treino e recalcular a média usando os dados novos. Isso “vaza” informação do futuro e destrói a matemática da rede neural. Use **sempre** apenas `transform()` no conjunto de teste. Este é o erro #1 em entrevistas de Machine Learning.

E. Tabela Numérica: Antes e Depois da Normalização

Para entender o efeito do `StandardScaler`, observe os valores reais (a tabela abaixo usa dados ilustrativos):

Feature	Antes (Bruto) Média	Escala	Depois (Normalizado) Média	Escala
Idade	≈ 44	$[18, 70]$	≈ 0.0	$[-1.8, +1.7]$
Salário	≈ 176k	$[2\text{k}, 350\text{k}]$	≈ 0.0	$[-1.7, +1.7]$

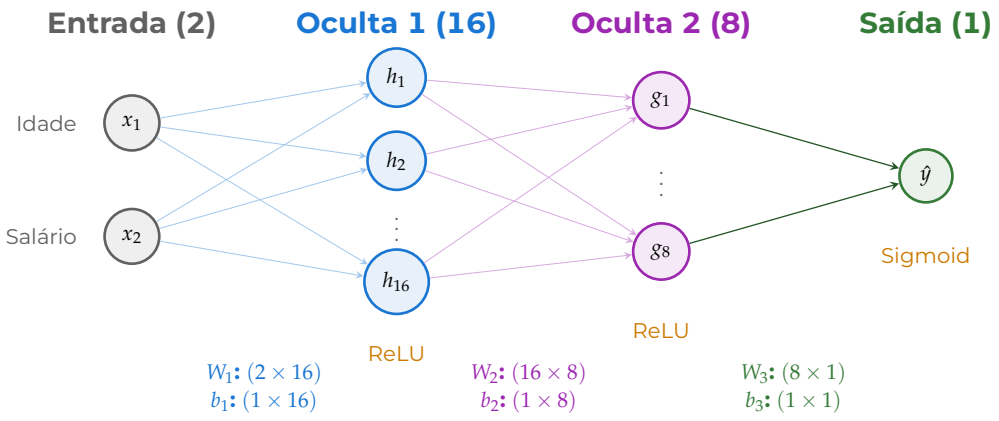
Sem normalização, o salário (escala de milhares) dominaria completamente a idade (escala de dezenas). O `StandardScaler` coloca ambas as features na mesma escala, com média ≈ 0 e desvio padrão ≈ 1.

Adicione a verificação de sanidade:

```
print(f"Antes:  Idade media={X_train[:,0].mean():.1f}, "
      f"Salario media={X_train[:,1].mean():.0f}")
print(f"Depois: Idade media={X_train_escalonado[:,0].mean():.4f}, "
      f"Salario media={X_train_escalonado[:,1].mean():.4f}")
# Depois da normalização, ambas as médias devem ser ~0.0
```

4 Parte 4: Construindo o Modelo Keras (12 min)

Agora que os tensores estão digeríveis, é hora de construir a rede. Observe o diagrama da arquitetura:



A rede possui **3 camadas**: duas ocultas com ReLU (16 e 8 neurônios) e uma de saída com Sigmoid. O total de parâmetros treináveis é $(2 \times 16 + 16) + (16 \times 8 + 8) + (8 \times 1 + 1) = 48 + 136 + 9 = \mathbf{193}$ — minúsculo para o Keras, mas suficiente para o nosso problema.

Adicione as funções ao seu arquivo:

```
def criar_modelo():
    """Cria uma MLP com Keras Sequential."""
    model = Sequential()
    model.add(Input(shape=(2,)))
    # Camada Oculta 1: 16 neurônios + ReLU
    model.add(Dense(16, activation='relu'))
    # Camada Oculta 2: 8 neurônios + ReLU
    model.add(Dense(8, activation='relu'))
```

```
    # Saída: Sigmoid para classificação binária
    model.add(Dense(1, activation='sigmoid'))
    return model

def compilar_e_treinar(model, X_t, y_t, epochs=50):
    """Compila e treina o modelo."""
    model.compile(
        optimizer='adam',              # Gradient Descent inteligente
        loss='binary_crossentropy',    # Loss para classificação binária
        metrics=['accuracy']           # Métrica legível
    )
    model.fit(X_t, y_t, epochs=epochs, verbose=0)
    return model
```

• Teoria: Do NumPy ao Keras — A Tradução

Cada linha do Keras tem um equivalente artesanal que você já programou:

Keras (Industrial)	NumPy (Artesanal — Labs 01–03)
`Dense(16, activation='relu')`	`CamadaDensa(2,16)` + `relu(Z)`
`loss='binary_crossentropy'`	`calcular_mse(y, y_pred)`
`optimizer='adam'`	`peso = peso - lr * gradiente`
`model.fit(X, y, epochs=80)`	Loop `for epoch in range(80)`

Você **sabe** o que acontece por trás de cada abstração — essa é a vantagem competitiva do engenheiro que aprendeu na mão.

5 Parte 5: Avaliando e Analisando (8 min)

Adicione o bloco principal que treina e avalia o modelo:

```
if __name__ == "__main__":
    # Criar e treinar
    modelo = criar_modelo()
    compilar_e_treinar(modelo, X_train_escalonado, y_train, epochs=80)

    # Avaliar no conjunto de TESTE (dados nunca vistos)
    perda, acuracia = modelo.evaluate(
        X_test_escalonado, y_test, verbose=0)

    print(f"\n{'='*40}")
    print(f"  Loss no Teste:     {perda:.4f}")
    print(f"  Acurácia no Teste: {acuracia:.2%}")
    print(f"{'='*40}")

    if acuracia > 0.65:
```

```
        print("O modelo SUPEROU o baseline de 65%!")
    else:
        print("ALERTA: O modelo NÃO atingiu o baseline.")
```

Execute `python src/churn_keras.py` e verifique que a acurácia supera 65%. Se não superar, tente:

- Aumentar `epochs` para 100 ou 150.
- Adicionar mais neurônios na camada oculta.
- Verificar se a normalização está aplicada corretamente.

6 Parte 6: Garantia de Qualidade — Testes Automatizados (7 min)

Os testes já estão prontos no arquivo `tests/test_churn.py` do seu repositório. Abra-o e observe a estrutura:

```
import numpy as np
from src.churn_keras import (criar_modelo, compilar_e_treinar,
                             X_train_escalonado, y_train,
                             X_test_escalonado, y_test,
                             scaler)

def test_modelo_supera_baseline():
    """A acurácia no teste DEVE superar 65%."""
    modelo = criar_modelo()
    compilar_e_treinar(modelo, X_train_escalonado, y_train)
    _, acc = modelo.evaluate(X_test_escalonado, y_test, verbose=0)
    assert acc > 0.65, f"Falha! Acurácia = {acc:.2%} (< 65%)"

def test_normalizacao_aplicada():
    """Garante que os dados foram normalizados corretamente."""
    media = np.abs(X_train_escalonado.mean(axis=0))
    # A média de cada coluna deve estar próxima de 0
    assert np.all(media < 0.1), f"Dados não normalizados! Média={
        media}"

def test_shapes_corretos():
    """Garante que os shapes de treino/teste estão consistentes."""
    assert X_train_escalonado.shape[1] == 2, "Features erradas"
    assert X_test_escalonado.shape[1] == 2, "Features erradas"
    assert len(y_train.shape) == 1, "Target deve ser 1D"
```

1. Rode os testes no terminal: `pytest tests/test_churn.py -v`.
2. Se a tela exibir **3 passed** em verde, sua implementação está correta.

7 Parte 7: Commit e Push (5 min)

Com todos os testes passando localmente, efetue o envio para o GitLab:

```
git add src/churn_keras.py
git commit -m "Lab 04: Keras Sequential + StandardScaler + CI/CD"
git push origin main
```

O GitLab CI interceptará seu push, criará um ambiente em nuvem e executará o `pytest` automaticamente. O **Selo Verde** (Pipeline Passed) confirma sua pontuação.

△ Importante: Se o Pipeline Falhar

Leia o log de erro no GitLab (aba CI/CD > Pipelines > Jobs). Os erros mais comuns são: (1) esquecer o `os.environ['TF_CPP_MIN_LOG_LEVEL']` antes dos imports do TensorFlow, (2) usar `fit_transform` no conjunto de teste (Data Leakage), (3) nome de função diferente do esperado pelo teste. O CI roda os **mesmos testes** que você rodou localmente.

Conclusão: O que Você Construiu Hoje

✔ Takeaways

- Você construiu seu primeiro **pipeline completo de Machine Learning industrial**: dados → normalização → modelo → treino → avaliação.
- Usou **pandas** para estruturar dados tabulares e separar Features (X) do Target (y).
- Aplicou o **StandardScaler** corretamente (`fit_transform` no treino, `transform` no teste) e entendeu o risco do *Data Leakage*.
- Treinou uma rede **Keras Sequential** com 2 camadas ocultas ($16 \to 8 \to 1$), validando acurácia $> 65\%$.
- Os 3 testes CI/CD garantem: baseline superado, dados normalizados e shapes consistentes.
- Cada abstração do Keras tem um equivalente artesanal que você programou nos Labs 01–03 — a diferença é **escala**, não **conceito**.

◇ **Informação**

Gancho para a Próxima Aula: Até agora, nosso target y é binário (0 ou 1: churn ou não-churn). Mas e se o problema tiver **3, 10 ou 100 classes**? E como saber se o modelo realmente aprendeu o padrão ou apenas **decorou** os dados de treino? Na próxima aula, entraremos na classificação **Multiclasse** com a função **Softmax** e enfrentaremos o vilão mais traiçoeiro do Machine Learning: o **Overfitting**. A pergunta será: *"meu modelo tira 99% no treino e 55% no teste — o que aconteceu?"*