Lab. 05: Classificação Multiclasse e Overfitting

Aléssio Miranda Júnior

Agosto - 2026/2

Lab. 05: Classificação Multiclasse e Overfitting

⋆ Objetivo do Laboratório

No Lab 04, você construiu um pipeline industrial de classificação binária com Keras. Hoje você enfrenta dois desafios simultâneos: primeiro, vai **provocar deliberadamente** o Overfitting para entender visualmente a diferença entre "aprender" e "decorar"; depois, vai migrar da Sigmoid para a **Softmax**, expandindo sua rede para classificar **múltiplas classes** ao mesmo tempo. Ao final, você saberá responder: *"meu modelo tira 99% no treino e 55% no teste — o que aconteceu?"*

1 Parte 1: O Problema E — Estilo Maratona

Problema E: O Detector de Anomalias da Indústria 4.0

Contexto: A fábrica *SmartFactory* instalou 10 sensores IoT em sua linha de produção. Cada sensor registra leituras contínuas (vibrações, temperaturas, pressões). Os engenheiros precisam de um classificador que identifique se uma peça saiu com **defeito** ou está **aprovada**. Porém, os dados dos sensores são ruidosos — cerca de 10% dos rótulos históricos estão errados (peças boas marcadas como defeituosas e vice-versa). Sua missão tem **duas fases**:

Fase A — Diagnóstico: Construir uma rede **propositalmente superdimensionada** (256-128-64 neurônios para apenas 500 amostras), treiná-la por 300 épocas e **provocar Overfitting**. O objetivo é produzir o gráfico de diagnóstico que comprova visualmente o sobreajuste.

Fase B — Multiclasse: Expandir o classificador de 2 classes (binário: defeito/ok) para **3 classes** usando Softmax — agora a peça pode ser `Aprovada`, `Defeito Menor` ou `Defeito Crítico`.

Modelo de Entrada (X): Vetor com 10 leituras de sensores (valores contínuos normalizados):

- $x_1 \dots x_5$: Sensores informativos (correlação real com o defeito)
- $x_6 \dots x_8$: Sensores redundantes (combinações lineares dos anteriores)
- x_9, x_{10}: Sensores ruidosos (dados espúrios sem utilidade)

Modelo de Saída:

- **Fase A** (Binário): $\hat{y} \in \{0, 1\}$ via Sigmoid — Aprovada ou Defeito
- **Fase B** (Multiclasse): $\hat{y} \in \{0, 1, 2\}$ via Softmax — 3 categorias de qualidade

Critérios de Aprovação:

1. **Fase A:** Gap (acurácia treino $-$ acurácia validação) $> 5\%$ (Overfitting confirmado)
2. **Fase B:** Acurácia multiclasse no teste $> 50\%$ (acima do baseline aleatório de 33%)

2 Parte 2: Gerando o Dataset com Ruído (8 min)

Para provocar Overfitting genuíno, precisamos de um dataset com ruído controlado. O parâmetro `flip_y=0.1` troca 10% dos rótulos aleatoriamente — simulando o caos de dados industriais reais.

Crie o arquivo `src/overfitting_lab.py`:

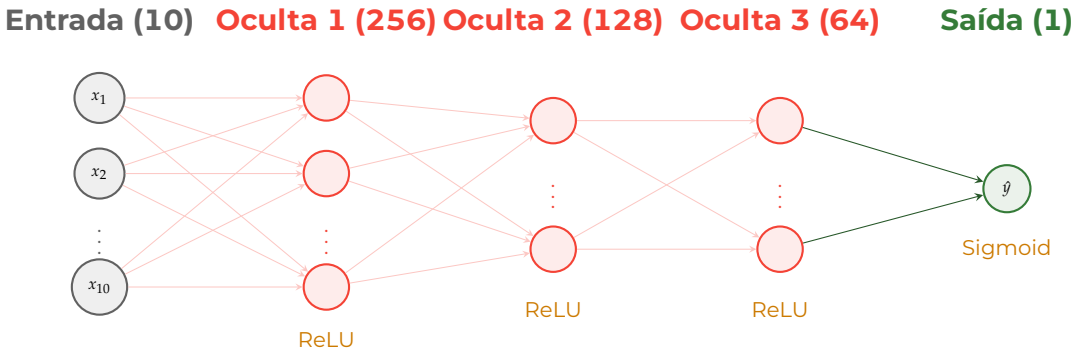
```
import numpy as np
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
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
from tensorflow.keras.utils import to_categorical

# Dataset: 500 amostras, 10 features, 2 classes (com ruído!)
X, y = make_classification(
    n_samples=500, n_features=10,
    n_informative=5, n_redundant=3,
    random_state=42, flip_y=0.1  # 10% de rótulos trocados
)

# Divisão treino/teste
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

# Normalização
scaler = StandardScaler()
X_train_norm = scaler.fit_transform(X_train)
X_test_norm = scaler.transform(X_test)
```

• Teoria: Por que flip_y=0.1?

O parâmetro `flip_y` troca aleatoriamente 10% dos rótulos. Na indústria, isso é equivalente a erros humanos de classificação no banco de dados. Uma rede pequena "ignora" essas anomalias e aprende o padrão geral; uma rede superdimensionada tenta "explicar" cada uma delas, criando contorções absurdas — o clássico Overfitting.

3 Parte 3: A Rede Superdimensionada (10 min)

Construa uma rede **propositalmente grande demais** para o problema. Com 10 features e 400 amostras de treino, uma rede com 256 neurônios na primeira camada é colossal — perfeita para decorar.



Total: $(10 \times 256 + 256) + (256 \times 128 + 128) + (128 \times 64 + 64) + (64 \times 1 + 1)$

$= 2.816 + 32.896 + 8.256 + 65 =$ **44.033 parâmetros!**

△ Importante: A Armadilha Proposital

Esta rede tem **44.033 parâmetros treináveis** para apenas **400 amostras** de treino. É como contratar 44 mil funcionários para processar 400 pedidos — a maioria ficará ociosa e começará a "inventar trabalho" (decorar ruído). A razão parâmetros/amostras (110 : 1) é absurda e garantirá o Overfitting.

Adicione ao seu arquivo:

```
def criar_rede_gigante():
    """Rede absurdamente grande para 10 features."""
    model = Sequential()
    model.add(Dense(256, activation='relu', input_shape=(10,)))
    model.add(Dense(128, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

    model.compile(
        optimizer='adam',
        loss='binary_crossentropy',
        metrics=['accuracy']
    )
    return model
```

4 Parte 4: Treinamento com Monitoramento (10 min)

Treine por 300 épocas e capture o objeto `history` — a "caixa preta" do Keras que registra loss e acurácia a cada época, tanto no treino quanto na validação:

```
modelo = criar_rede_gigante()

# Treinar com validation_split para monitorar overfitting
historico = modelo.fit(
    X_train_norm, y_train,
    epochs=300,
    batch_size=16,
    validation_split=0.2,  # 20% do treino vira validação
    verbose=0
)

# Extrair as curvas de aprendizado
loss = historico.history['loss']
val_loss = historico.history['val_loss']
acc = historico.history['accuracy']
val_acc = historico.history['val_accuracy']
```

A. O que Esperar: Assinatura do Overfitting

Antes de plotar, entenda o padrão que você deve observar. A tabela abaixo mostra os marcos típicos durante o treino da rede superdimensionada:

Época	Loss Treino	Loss Valid.	Acc Treino	Acc Valid.	Diagnóstico
1–30	↓↓	↓↓	↑↑	↑↑	Aprendizado real
30–80	↓	≈ estável	↑	≈ estável	Ponto de inflexão
80–300	↓↓	↑↑	→ 99%	↓	**Overfitting!**

A "assinatura" do Overfitting é inconfundível: a loss de treino continua caindo (a rede memoriza cada exemplo), mas a loss de validação **sobe** (ela não generaliza para dados novos). O gap cresce monotonicamente.

5 Parte 5: Plotando as Curvas de Diagnóstico (10 min)

Crie o gráfico que é a "radiografia" do Overfitting:

```
def plotar_diagnostico(loss, val_loss, acc, val_acc):
    """Gera o gráfico de diagnóstico de Overfitting."""
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))
    epocas = range(1, len(loss) + 1)
```

```
    # Gráfico 1: Loss
    ax1.plot(epocas, loss, 'b-', label='Loss (Treino)',
             linewidth=1.5)
    ax1.plot(epocas, val_loss, 'r-',
             label='Val Loss (Validação)', linewidth=1.5)
    ax1.set_title('Curvas de Perda')
    ax1.set_xlabel('Épocas')
    ax1.set_ylabel('Loss')
    ax1.legend()
    ax1.grid(True, alpha=0.3)

    # Gráfico 2: Accuracy
    ax2.plot(epocas, acc, 'b-', label='Accuracy (Treino)',
             linewidth=1.5)
    ax2.plot(epocas, val_acc, 'r-',
             label='Val Accuracy (Validação)', linewidth=1.5)
    ax2.set_title('Curvas de Acurácia')
    ax2.set_xlabel('Épocas')
    ax2.set_ylabel('Accuracy')
    ax2.legend()
    ax2.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('overfitting.png', dpi=150)
    plt.close()
    print("Gráfico salvo: overfitting.png")
```

• Teoria: O que Observar no Gráfico

- No gráfico de **Loss**: a linha azul (treino) cai até quase zero, mas a vermelha (validação) atinge um mínimo e depois **sobe** — é a "boca de jacaré" do Overfitting.
- No gráfico de **Accuracy**: a acurácia de treino chega a 99%+, mas a de validação estagna ou cai.
- O **gap** entre treino e validação é o indicador direto da gravidade do sobreajuste.

6 Parte 6: Migrando para Multiclasse com Softmax (7 min)

Agora que você diagnosticou o Overfitting no problema binário, é hora de expandir para **3 classes**. A tabela abaixo mostra exatamente o que muda:

Componente	Binário (Fase A)	Multiclasse (Fase B)
Saída da rede	1 neurônio (Sigmoid)	3 neurônios (Softmax)
Função de ativação	`sigmoid`	`softmax`
Função de perda	`binary_crossentropy`	`categorical_crossentropy`
Formato do target	$y \in \{0,1\}$	One-Hot: $[1,0,0]$, $[0,1,0]$, $[0,0,1]$

Adicione ao seu arquivo a versão multiclasse:

```
# === FASE B: MULTICLASSE COM SOFTMAX ===
X_multi, y_multi = make_classification(
    n_samples=500, n_features=10,
    n_informative=5, n_redundant=3,
    n_classes=3, n_clusters_per_class=1,
    random_state=42, flip_y=0.05
)

X_train_m, X_test_m, y_train_m, y_test_m = train_test_split(
    X_multi, y_multi, test_size=0.2, random_state=42)

scaler_m = StandardScaler()
X_train_m_norm = scaler_m.fit_transform(X_train_m)
X_test_m_norm = scaler_m.transform(X_test_m)

# Converter target para One-Hot Encoding
y_train_cat = to_categorical(y_train_m, num_classes=3)
y_test_cat = to_categorical(y_test_m, num_classes=3)

def criar_rede_multiclasse():
    """Rede para 3 classes com Softmax."""
    model = Sequential()
    model.add(Dense(32, activation='relu', input_shape=(10,)))
    model.add(Dense(16, activation='relu'))
    model.add(Dense(3, activation='softmax'))  # 3 classes!

    model.compile(
        optimizer='adam',
        loss='categorical_crossentropy',  # Multiclasse!
        metrics=['accuracy']
    )
    return model
```

• Teoria: Softmax: O que Muda na Prática

A Softmax converte K valores brutos em uma **distribuição de probabilidades** que soma exatamente 1. Exemplo para $z = [2.0, 1.0, 0.1]$:

Classe	Cálculo	Probabilidade
Aprovada	$\frac{e^{2.0}}{e^{2.0}+e^{1.0}+e^{0.1}} = \frac{7.39}{11.22}$	**65.9%**
Defeito Menor	$\frac{e^{1.0}}{11.22}$	24.2%
Defeito Crítico	$\frac{e^{0.1}}{11.22}$	9.9%
	Soma	**100%**

A rede prediz "Aprovada" com 65.9% de confiança. Compare com a Sigmoid, que só produzia um único número $\in (0, 1)$.

7 Parte 7: Bloco Principal e Execução (5 min)

Monte o bloco principal que executa ambas as fases:

```
if __name__ == "__main__":
    # === FASE A: Overfitting ===
    plotar_diagnostico(loss, val_loss, acc, val_acc)

    gap = acc[-1] - val_acc[-1]
    print(f"\n{'='*50}")
    print(f"  FASE A - Diagnóstico de Overfitting")
    print(f"  Loss Treino Final:     {loss[-1]:.4f}")
    print(f"  Loss Validação Final:  {val_loss[-1]:.4f}")
    print(f"  Acc Treino Final:      {acc[-1]:.2%}")
    print(f"  Acc Validação Final:   {val_acc[-1]:.2%}")
    print(f"  Gap (Treino - Valid):   {gap:.2%}")
    print(f"{'='*50}")

    # === FASE B: Multiclasse ===
    modelo_multi = criar_rede_multiclasse()
    modelo_multi.fit(X_train_m_norm, y_train_cat,
                     epochs=100, verbose=0)

    _, acc_multi = modelo_multi.evaluate(
        X_test_m_norm, y_test_cat, verbose=0)

    print(f"\n{'='*50}")
    print(f"  FASE B - Classificação Multiclasse (Softmax)")
    print(f"  Acurácia no Teste:     {acc_multi:.2%}")
    print(f"  Baseline (aleatório):  33.3%")
    print(f"{'='*50}")
```

Execute `python src/overfitting_lab.py` e verifique que:

- **Fase A:** O gap treino−validação é $> 5\%$ (Overfitting confirmado).
- **Fase B:** A acurácia multiclasse supera 50% (acima do baseline aleatório de 33%).

8 Parte 8: Garantia de Qualidade — Testes CI/CD (5 min)

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

```
import numpy as np
from src.overfitting_lab import (
    criar_rede_gigante, criar_rede_multiclasse,
    historico, modelo,
    X_test_norm, y_test,
    X_train_m_norm, y_train_cat,
    X_test_m_norm, y_test_cat
)

def test_overfitting_detectado():
    """Confirma que o modelo sofre de overfitting."""
    loss = historico.history['loss']
    val_loss = historico.history['val_loss']
    assert val_loss[-1] > loss[-1], \
        "Overfitting não detectado: val_loss > loss"

def test_modelo_funcional():
    """Garante que o modelo gera previsões válidas."""
    preds = modelo.predict(X_test_norm, verbose=0)
    assert preds.shape == (len(X_test_norm), 1)
    assert np.all(preds >= 0) and np.all(preds <= 1), \
        "Sigmoid deveria produzir valores entre 0 e 1"

def test_historico_completo():
    """Garante que o histórico registrou 300 épocas."""
    assert len(historico.history['loss']) == 300
    assert 'val_loss' in historico.history

def test_multiclasse_supera_baseline():
    """Acurácia multiclasse deve superar 50%."""
    m = criar_rede_multiclasse()
    m.fit(X_train_m_norm, y_train_cat,
          epochs=100, verbose=0)
    _, acc = m.evaluate(X_test_m_norm, y_test_cat,
                        verbose=0)
    assert acc > 0.50, f"Acc={acc:.2%} < 50%"
```

1 Rode os testes no terminal: `pytest tests/test_overfitting.py -v`.

2 Se a tela exibir **4 passed** em verde, sua implementação está correta.

9 Parte 9: Commit e Push (5 min)

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

```
git add src/overfitting_lab.py overfitting.png
git commit -m "Lab 05: Overfitting + Softmax Multiclasse"
git push origin main
```

O GitLab CI interceptará seu push e executará o `pytest` automaticamente. O **Selo Verde** confirma sua pontuação.

△ Importante: Se o Pipeline Falhar

Os erros mais comuns são: (1) esquecer o `to_categorical()` na conversão do target multiclasse, (2) usar `binary_crossentropy` com Softmax (a loss correta é `categorical_crossentropy`), (3) não salvar o gráfico `overfitting.png` antes do push. O CI roda os **mesmos testes** que você rodou localmente.

Conclusão: O que Você Construiu Hoje

✔ Takeaways

- Você provocou **Overfitting deliberadamente** usando uma rede superdimensionada (256+128+64 neurônios, 44.033 parâmetros) treinada por 300 épocas em apenas 400 amostras.
- Plotou as **curvas de diagnóstico** (Loss e Accuracy) e identificou a "boca de jacaré": loss de treino ↓ enquanto loss de validação ↑.
- Migrou de classificação **binária** (Sigmoid) para **multiclasse** (Softmax), trocando apenas a ativação final, o formato do target (`to_categorical`) e a loss (`categorical_crossentropy`).
- Verificou que a Softmax produz uma distribuição de probabilidades que soma exatamente 1 para K classes.
- Os 4 testes CI/CD confirmam: Overfitting detectado, modelo funcional, histórico completo e multiclasse acima do baseline.

◇ **Informação**

Gancho para a Próxima Aula: Agora que você sabe **diagnosticar** o Overfitting, a pergunta natural é: *"como impedi-lo?"* Na próxima aula, aprenderemos a "podar" os neurônios com **Dropout** (desligar aleatoriamente neurônios durante o treino), a parar o treinamento automaticamente com **Early Stopping** quando o `val_loss` começa a subir, e a aplicar a **Regularização L2** que penaliza pesos grandes. A receita completa transformará aquela rede "decoradora" em um modelo robusto e generalizável.