Aula 5: O Problema do Overfitting e Diagnóstico de Modelos

Aléssio Miranda Júnior

Agosto - 2026/2

O que você verá neste capítulo

Construir um modelo de rede neural com alta acurácia nos dados de treinamento é uma conquista enganosa. Neste capítulo, desmistificaremos o fenômeno do *Overfitting* (Sobreajuste) — a armadilha mais perigosa da Inteligência Artificial —, formalizaremos a divisão rigorosa dos dados em conjuntos de Treino, Validação e Teste conforme a metodologia de Hastie, Tibshirani e Friedman (2009), e introduziremos os gráficos de monitoramento que permitem ao engenheiro diagnosticar, em tempo real, se a rede está aprendendo ou decorando.

1 O Problema da Memorização

Se você está acompanhando o **Caderno de Laboratórios** em paralelo a este livro-texto, você já deve ter construído na prática o classificador da aula passada e visto que ele alcançou uma alta taxa de acerto nos dados de treinamento. No entanto, em Machine Learning, acurácia alta no treino não significa, necessariamente, que a rede "aprendeu". Ela pode ter simplesmente **decorado** as respostas, agindo como um aluno que decora o gabarito de uma prova anterior, mas tira nota zero em questões inéditas.

Essa falha é chamada de **Overfitting** (Sobreajuste). Ela ocorre quando a rede neural é excessivamente complexa para o volume de dados disponível e treina por tempo demais. A rede começa a modelar os "ruídos" (imperfeições e variações aleatórias específicas do dataset) em vez de focar nos padrões genéricos e reproduzíveis do problema. Quando recebe dados novos no mundo real, a IA falha miseravelmente.

Figura 1: Analogia visual: o modelo *Underfit* é simples demais (uma reta); o *Ajuste Ótimo* entende a tendência real do problema; o *Overfit* cria regras hipercomplexas para decorar até o ruído.

• Teoria: O Dilema Viés-Variância (Bias-Variance Tradeoff)

O Overfitting é uma manifestação do clássico **Dilema Viés-Variância**, formalizado extensivamente na literatura estatística. Todo modelo preditivo carrega dois tipos de erro:

- **Viés (Bias):** O erro sistemático causado por um modelo simples demais que não consegue capturar a complexidade dos dados. Uma reta tentando representar uma parábola tem alto viés.
- **Variância:** O erro causado por um modelo complexo demais que é hipersensível às flutuações do dataset de treinamento. Uma curva que serpenteia por cada ponto tem alta variância.

O objetivo do engenheiro é encontrar o ponto ótimo entre viés e variância, onde o modelo é complexo o suficiente para capturar os padrões reais, mas não tão flexível a ponto de decorar o ruído.

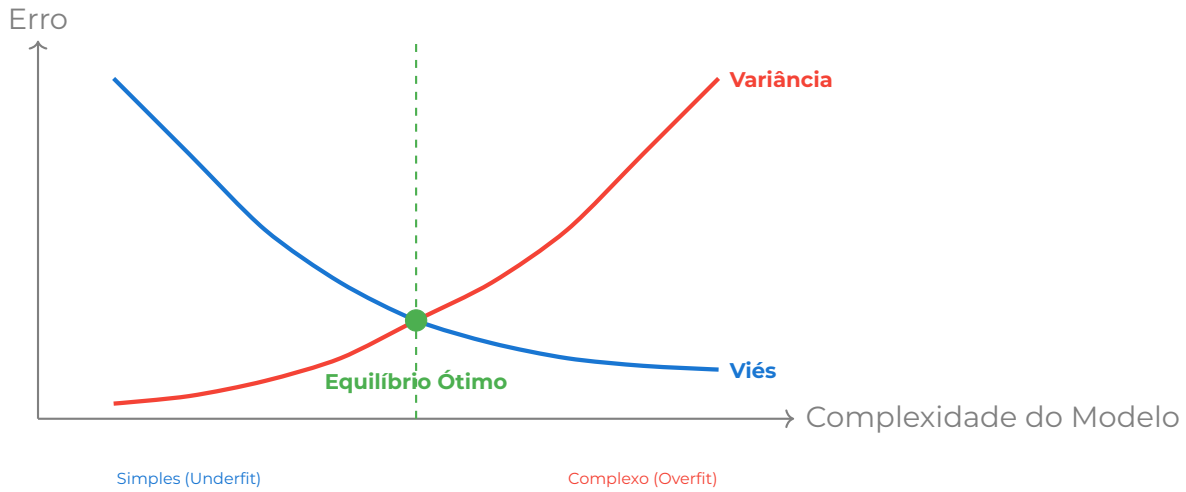


Figura 2: O Dilema Viés-Variância: modelos simples erram por viés alto (azul), modelos complexos erram por variância alta (vermelho). O ponto verde marca o equilíbrio ótimo que maximiza a generalização.

Na prática, o Overfitting se manifesta em três cenários clássicos:

1. **Rede muito grande para poucos dados:** Uma rede com milhões de parâmetros treinada com apenas 200 amostras terá capacidade computacional de sobra para memorizar cada exemplo.
2. **Treinamento excessivo (muitas épocas):** Mesmo um modelo bem dimensionado pode decorar se treinar por tempo demais.
3. **Dados não representativos:** Se o conjunto de treinamento não reflete a diversidade do mundo real, o modelo aprenderá os vícios do dataset.

O polo oposto — o **Underfitting** (Subajuste) — ocorre quando o modelo é simples demais para capturar sequer os padrões triviais. Uma reta tentando separar dados em espiral é um exemplo. O Underfitting é fácil de diagnosticar (o modelo erra em tudo), enquanto o Overfitting é traiçoeiro (parece perfeito no treino, mas falha em produção).

2 Divisão de Dados: Treino, Validação e Teste

A única forma comprovada de impedir o Overfitting é **esconder** uma parcela dos dados da rede durante o treinamento e utilizá-la exclusivamente para medir a capacidade de generalização. Essa divisão rigorosa é a espinha dorsal de qualquer projeto sério de Inteligência Artificial, estabelecida como padrão pela comunidade de Machine Learning desde os anos 1990.

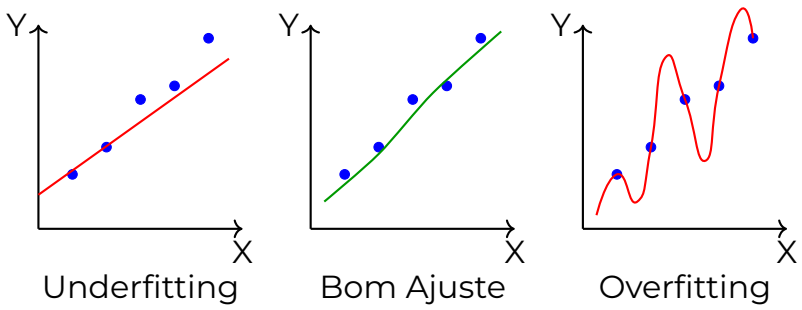


Figura 3: Comparação entre Underfitting, Bom Ajuste e Overfitting. À esquerda, o modelo é linear demais (alto viés). Ao centro, o ajuste suave captura a tendência real. À direita, a curva serpenteia por cada ponto individual, memorizando ruído.

Em geral, dividimos os dados em três fatias estratégicas:

- **Dados de Treino (Train):** A maior fatia (tipicamente 70%). É o material de estudo da rede. A IA examina as respostas, calcula a perda (*loss*) e ajusta os pesos via Gradiente. A rede enxerga e aprende diretamente com esses dados.
- **Dados de Validação (Validation):** Uma fatia menor (tipicamente 15%). Usada **ao final de cada época** de treinamento. A IA **não ajusta os pesos**

com base nesses dados; apenas calcula o *val_loss* para medir se está generalizando. É a "prova surpresa" que o professor aplica ao final de cada semana.

- **Dados de Teste (Test):** A última fatia (15%). Escondida em um "cofre digital" intocável até o fim do projeto. É a prova final usada uma única vez para atestar a qualidade real do modelo para o cliente ou para publicação científica. Jamais se treina ou se ajusta hiperparâmetros com base nos dados de teste.

► Prática: A Implementação no Scikit-Learn

A divisão dos dados é automatizada pela função `train_test_split` da biblioteca `scikit-learn`:

```
from sklearn.model_selection import train_test_split

# Primeira divisao: 85% temporario, 15% teste
X_temp, X_test, y_temp, y_test = train_test_split(
    X, y, test_size=0.15, random_state=42
)

# Segunda divisao: dos 85%, separar 15% para validacao
X_train, X_val, y_train, y_val = train_test_split(
    X_temp, y_temp, test_size=0.176, random_state=42
)
# Resultado: ~70% treino, ~15% validacao, 15% teste
```

O parâmetro `random_state=42` garante que a divisão seja **reprodutível**. Qualquer pessoa que execute o código com o mesmo seed obterá exatamente as mesmas fatias, eliminando a variabilidade experimental.

3 Acompanhando o Gráfico do Overfitting

Ao utilizarmos um conjunto de validação no Keras (via o parâmetro `validation_data`), o framework plota automaticamente duas curvas simultâneas a cada época: o `loss` (erro no conjunto de treino) e o `val_loss` (erro no conjunto de validação).

O comportamento saudável de uma rede que está **aprendendo** exibe ambas as curvas descendo em paralelo. Porém, invariavelmente, chega um momento na linha do tempo em que o `loss` continua a cair infinitamente (a rede está decorando com sucesso crescente), mas o `val_loss` **para de cair e começa a subir**.

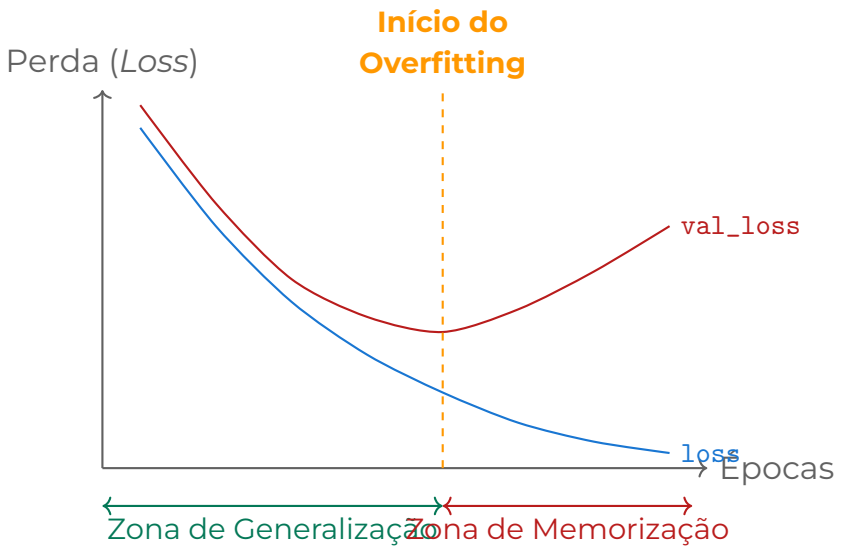


Figura 4: O gráfico canônico do Overfitting. A curva azul (treino) cai indefinidamente. A curva vermelha (validação) atinge um vale e depois inverte, sinalizando que a rede parou de generalizar e começou a memorizar.

O exato momento em que o `val_loss` começa a subir é o **marco zero do Overfitting**. O modelo parou de abstrair padrões genuínos e começou a alucinar detalhes irrelevantes, distanciando-se progressivamente do mundo real. A tabela a seguir resume os cenários possíveis:

Cenário	loss	val_loss	Diagnóstico
Underfitting	Alto e estável	Alto e estável	Rede simples demais
Bom Ajuste	Cai gradualmente	Cai junto	Continuar treinando
Overfit Leve	Continua caindo	Para de cair	Cautela
Overfit Grave	Vai a zero	**Sobe!**	Parar imediatamente

Tabela 1: Guia de diagnóstico do treinamento baseado nas curvas de loss e val_loss. O engenheiro deve monitorar continuamente o comportamento das duas curvas.

A partir do ponto de inflexão, cada época adicional de treinamento **piora** o modelo em produção, mesmo que o `loss` de treino continue caindo.

△ Importante: A Armadilha do “99% de Acurácia”

A acurácia no conjunto de treinamento é uma métrica **enganosa**. Se o seu modelo atinge 99% de acurácia no treino, mas apenas 62% na validação, ele está sofrendo de Overfitting severo. O gap entre as duas métricas (37 pontos percentuais nesse exemplo) é o indicador mais direto da gravidade do sobreajuste.

3.1 Exceções à Regra: Quando o Overfitting é o Objetivo?

O Overfitting é geralmente o maior inimigo de modelos preditivos que lidam com dados do mundo real. No entanto, existem exceções. Decorar dados pode ser um recurso intencional, dependendo da natureza do problema:

- **Sistemas de Domínio Finito e Fechado:** Se o dataset contiver *todas* as possibilidades absolutas do problema (ex: mapeamento estrito de portas lógicas complexas ou todas as regras possíveis de um jogo da velha simples), não há necessidade de "generalizar"para o desconhecido. A rede atua essencialmente como um banco de dados super-rápido.
- **Memorização Factual em LLMs:** Em Grandes Modelos de Linguagem, queremos que o modelo generalize a gramática e o raciocínio, mas faça overfitting em *fatos exatos*. Não queremos que a IA "generalize"a capital do Brasil, queremos que ela puxe o dado decorado perfeitamente.
- **Detecção de Anomalias (Autoencoders):** Em cibersegurança, podemos forçar um modelo a fazer overfitting apenas nas amostras de tráfego "normal"de uma rede corporativa. Ele vai decorar tão perfeitamente a assinatura do tráfego normal que, quando um ataque inédito ocorrer, o erro de processamento vai disparar, servindo como um alarme imediato.
- **Watermarking (Marca d'água em IA):** Empresas inserem intencionalmente padrões arbitrários secretos em poucos dados e deixam o modelo fazer overfitting neles. Se o modelo for roubado, a empresa pode provar sua autoria demonstrando que o clone decorou aquele padrão exato.

O overfitting só é um defeito quando usamos a IA como uma ferramenta de dedução estatística; ele se torna um recurso valioso quando buscamos compressão ou mapeamento determinístico.

4 Próximos Passos: O Combate ao Overfitting

Saber diagnosticar o Overfitting é apenas metade do trabalho do Engenheiro de Inteligência Artificial. Se, ao observar a curva de validação, percebemos que a rede neural tem uma forte tendência a decorar o dataset, o que devemos fazer?

A resposta reside na **Regularização**, uma família de técnicas matemáticas e algorítmicas projetadas para sabotar ativamente a capacidade da rede de memorizar os dados. Duas técnicas notáveis que implementaremos na próxima etapa do curso são:

- **Early Stopping (Parada Antecipada):** Um monitor automatizado que observa a métrica `val_loss`. No instante em que o erro de validação parar de

cair e der sinais de que vai subir (o marco zero do Overfitting), o algoritmo interrompe o treinamento *imediatamente*, salvando os melhores pesos encontrados e ignorando o restante das épocas programadas.

- **Dropout (Descarte Aleatório):** Um dos métodos mais elegantes do Deep Learning. Durante cada época de treino, o Dropout desliga (zera os sinais) uma porcentagem aleatória de neurônios da rede. Isso obriga os neurônios sobreviventes a assumirem a responsabilidade pelo aprendizado, impedindo-os de formarem "conluios"para decorar características muito específicas de uma amostra. Como a arquitetura "pisca"e muda a cada ciclo, a rede é forçada a generalizar.

✔ Takeaways

- **Overfitting** ocorre quando a rede memoriza o treino em vez de aprender padrões generalizáveis — acurácia alta no treino mas falha catastrófica em dados novos.
- O **Dilema Viés-Variância** é o equilíbrio fundamental: modelo simples demais (alto viés, underfitting) vs complexo demais (alta variância, overfitting).
- Dividir os dados em **Treino (70%) / Validação (15%) / Teste (15%)** é obrigatório para detectar e prevenir overfitting.
- A **assinatura do Overfitting** é visual: a curva `loss` desce, mas a curva `val_loss` para e começa a subir.
- Para combater o Overfitting, técnicas de **Regularização** como Early Stopping e Dropout são inseridas na arquitetura da rede neural para sabotar ativamente a memorização.

5 Conclusão

Monitorar a métrica de validação a cada época (*epoch*) é uma obrigação inegociável. Sem validação, o engenheiro pode estar entregando um modelo viciado e perigoso para produção — uma IA médica que “acerta” 99% no treino mas falha em 40% dos pacientes reais é uma bomba ética e jurídica.

Na próxima aula, veremos como parar automaticamente o treinamento antes que o Overfitting se instale, e como “podar” os neurônios que estão decorando, utilizando as técnicas de **Regularização**: *Dropout* e *Early Stopping*.

✎ Questões: Autoavaliação

1 **[Direta]**Um modelo Keras atinge 98% de acurácia no treino, mas apenas 61% na validação. Diagnostique o problema e explique o que o gap de 37 pontos indica.

2 **[Direta]**Explique a diferença funcional entre o conjunto de **Validação** e o conjunto de **Teste**. Por que não basta dividir os dados em apenas dois grupos?

3 **[Direta]**Dado o vetor de saídas brutas $z = [3.0, 1.0, 0.5]$, calcule as probabilidades Softmax para cada classe e identifique a classe predita.

4 **[Reflexiva]**Uma empresa de saúde treinou uma IA para diagnosticar doenças e obteve 99% de acurácia no treino. O gerente quer colocar o modelo em produção imediatamente. Que argumentos você usaria para convencê-lo a esperar?

5 **[Reflexiva]**O Overfitting é sempre negativo? Existe algum cenário onde "memorizar" dados pode ser aceitável ou até desejável? Justifique.

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?”*

6 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):

- $x_1 \ldots x_5$: Sensores informativos (correlação real com o defeito)
- $x_6 \ldots x_8$: 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%)

7 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`:

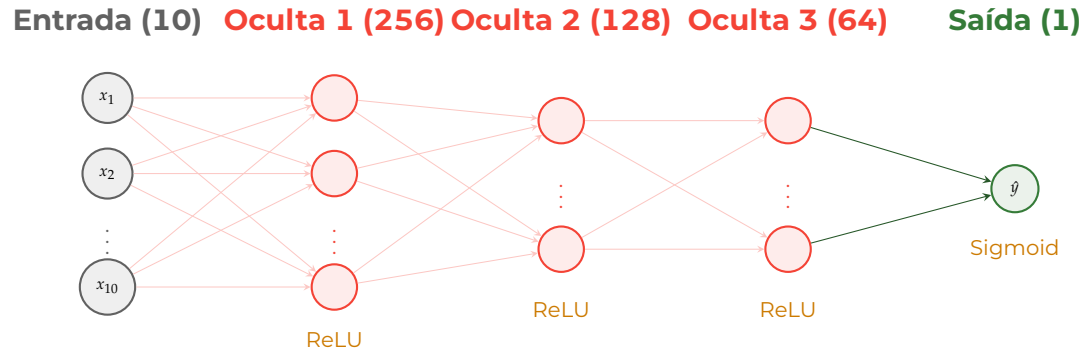
```python
import numpy as np
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

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.

8 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:

```python
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
```

9 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.

10 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)
```

```
5
6       # Gráfico 1: Loss
7       ax1.plot(epocas, loss, 'b-', label='Loss (Treino)',
8                linewidth=1.5)
9       ax1.plot(epocas, val_loss, 'r-',
10               label='Val Loss (Validação)', linewidth=1.5)
11      ax1.set_title('Curvas de Perda')
12      ax1.set_xlabel('Épocas')
13      ax1.set_ylabel('Loss')
14      ax1.legend()
15      ax1.grid(True, alpha=0.3)
16
17      # Gráfico 2: Accuracy
18      ax2.plot(epocas, acc, 'b-', label='Accuracy (Treino)',
19               linewidth=1.5)
20      ax2.plot(epocas, val_acc, 'r-',
21               label='Val Accuracy (Validação)', linewidth=1.5)
22      ax2.set_title('Curvas de Acurácia')
23      ax2.set_xlabel('Épocas')
24      ax2.set_ylabel('Accuracy')
25      ax2.legend()
26      ax2.grid(True, alpha=0.3)
27
28      plt.tight_layout()
29      plt.savefig('overfitting.png', dpi=150)
30      plt.close()
31      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 sobre-ajuste.

11 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 $\in (0, 1)$.

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

Monte o bloco principal que executa ambas as fases:

```python
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%).

13 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:

```python
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.

14 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.