Inteligência Artificial 2

O Ecossistema Keras e Dados Reais

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

18 de agosto de 2026

Onde Paramos?

Na aula anterior, você dominou a **mecânica do aprendizado**:

- A **MSE** mede o erro: $\mathcal{L} = \frac{1}{N}\sum(y - \hat{y})^2$.
- O **Gradient Descent** caminha contra o gradiente: $W \leftarrow W - \alpha \nabla W$.
- O **Backpropagation** calcula gradientes eficientemente via Regra da Cadeia.

A Pergunta de Hoje

Tudo foi manual e artesanal (NumPy puro).
Como escalar para **milhões de parâmetros**?

Objetivos da Sessão

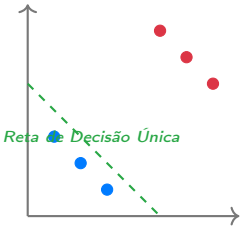
- **A Ponte Perdida:** Por que precisamos de múltiplas camadas? O Problema do XOR.
- **Topologia Empírica:** A intuição visual na escolha de neurônios.
- **O Teto do NumPy:** Por que a CPU não basta para Deep Learning.
- **TensorFlow & Keras:** Do código artesanal às abstrações industriais.
- **Dados Reais & Normalização:** Ingestão com pandas e `StandardScaler`.
- **Compile & Fit:** O loop de aprendizado em 2 linhas.

A Limitação do Neurônio Solitário

Até agora, vocês entenderam a fundo um **Perceptron Simples**. Mas qual é o limite prático dele?

O que o Perceptron faz:

- Ele traça **uma única reta** (hiperplano) no espaço de características.
- É perfeito para problemas **linearmente separáveis**.
- Exemplo: Separar maçãs de laranjas pelo peso e cor.



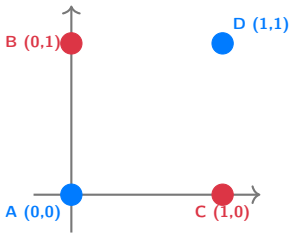
O Pesadelo Não-Linear: O Problema do XOR

E quando o mundo real não puder ser cortado por uma única reta?

A operação matemática do **Ou Exclusivo (XOR)**:

- Posições A e D (0,0 e 1,1) são da Classe 0.
- Posições B e C (0,1 e 1,0) são da Classe 1.

Desafio mental: Tente separar as bolinhas vermelhas das azuis desenhando APENAS UMA linha reta. É matematicamente impossível!

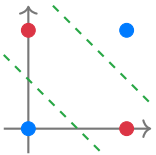


A Mágica das Camadas Ocultas (*Hidden Layers*)

Como resolvemos o XOR? Com as **Camadas Ocultas (MLP)!**

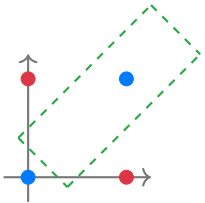
- Cada neurônio em uma camada oculta traça **uma reta própria**.
- O conjunto de neurônios **dobra e estica o espaço** para que a camada de saída possa aplicar o corte final.

Tentativa 1: Apenas 2 Neurônios



Traçam um "corredor"(2 retas).
Teoricamente possível, mas o gradiente se perde e a rede falha em quase todas as tentativas (mínimos locais).

Tentativa 2: Usando 4 Neurônios



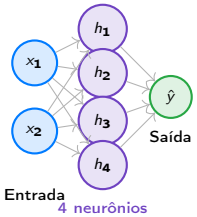
Traçam uma "caixa"fechada (4 retas).
Espaço fica flexível. O aprendizado converge muito mais rápido e sem falhas.

Largura vs. Profundidade: O Mesmo XOR, Duas Filosofias

E se, ao invés de *alargar* a camada, nós a **aprofundássemos**?

Solução por Largura

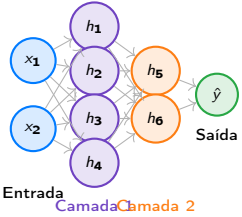
$2 \to 4 \to 1$



Uma camada larga "estica" o espaço de uma vez só.

Solução por Profundidade

$2 \to 4 \to 2 \to 1$



Duas camadas compõem representações **hierárquicas**.

Evidência Empírica: TensorFlow Playground (XOR, Tanh, 537 épocas)

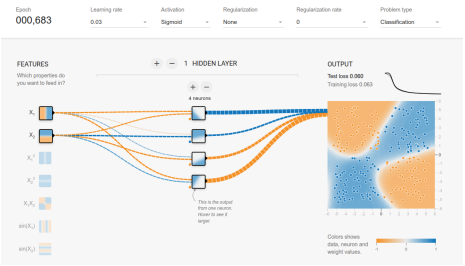
Arquitetura	Test Loss	Fronteira
1 camada oculta [4]	0,063	Irregular, ruidosa
2 camadas ocultas [4,2]	**0,001**	**XOR perfeito (4 quadrantes)**

A profundidade compõe representações **hierárquicas**: a 1ª camada aprende fronteiras locais; a 2ª as combina em

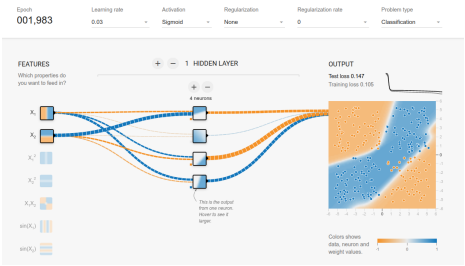
Estudo de Caso: A Instabilidade em Redes Rasas (Sigmoid)

Configuração: 1 Camada Oculta, 4 neurônios, ativação **Sigmoid**.
Dois treinamentos idênticos. Por que o resultado final foi diferente?

Treinamento A



Treinamento B

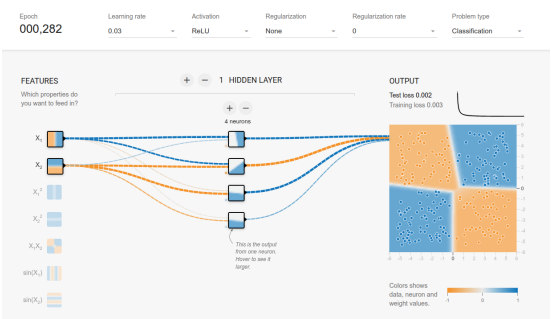


O Problema dos Mínimos Locais

A *Sigmoid* cria fronteiras curvas e suaves, mas é muito sensível à **inicialização aleatória dos pesos**. A rede pode ficar presa em diferentes **mínimos locais**, tornando o aprendizado em redes rasas instável e dependente da "sorte".

Estudo de Caso: A Geometria da ReLU

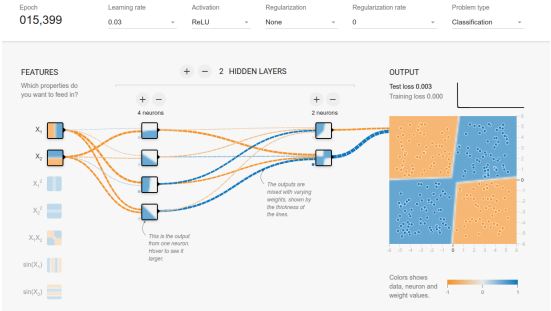
Configuração: 1 Camada Oculta, 4 neurônios, ativação **ReLU**.
Ao trocar a função de ativação, a geometria da fronteira muda drasticamente.



- **Fronteiras Retas:** A ReLU ($f(x) = \max(0, x)$) é *piecewise linear*. Ela traça polígonos com quinas secas em vez de curvas.
- **Vantagem:** O gradiente flui melhor (evita os mínimos locais e o gradiente evanescente da Sigmoid).
- **Limitação Raso:** Com apenas 1 camada, a rede até tenta isolar o centro, mas as áreas vizinhas não ficam divididas em "quadrantes puros".

Estudo de Caso: O Poder da Profundidade (Deep Learning)

Configuração: 2 Camadas Ocultas (4 e 3 neurônios).
A verdadeira mágica surge na composição de **representações hierárquicas**.

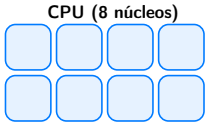


- **Camada 1 (4 neurônios):** Aprende "conceitos primitivos" (linhas retas individuais que cortam o plano).
- **Camada 2 (2 neurônios):** Combina essas linhas soltas de forma lógica (como portas AND/OR), montando a estrutura completa.
- **Resultado:** Generalização limpa e perfeita. Os 4 quadrantes são isolados com total confiança pela rede.

O Teto Computacional do NumPy

CPU (NumPy)

- 8–32 núcleos potentes.
- Ótima para lógica complexa (`if/else`).
- Péssima para trilhões de multiplicações simples.



GPU (TensorFlow/CUDA)

- 10.000+ micro-núcleos paralelos.
- Projetada para álgebra linear massiva.
- Treino que levaria **semanas** na CPU → **horas** na GPU.

CPU vs GPU: A Diferença em Números

	CPU (NumPy)	GPU (TensorFlow)
Núcleos	8–32 (potentes)	10.000+ (simples)
Operação	Sequencial	Massivamente paralela
Ideal para	Lógica, I/O, strings	Álgebra linear
ResNet-50 treino	$\sim$2 semanas	$\sim$4 horas
Custo	CPU do notebook	GPU dedicada ($$$)

A Fórmula de Amdahl

Deep Learning = 99% multiplicação de matrizes. A GPU é projetada exatamente para isso — o ganho chega a **100x**.

A Analogia: Professor vs Alunos

Para entender por que a álgebra linear massiva prefere a GPU, pense em matemática simples:

O Professor (CPU)

1 gênio capaz de resolver derivadas parciais avançadas em segundos. Mas se você pedir para ele somar 10.000 números simples, ele fará **um por vez**.

O Ginásio (GPU)

5.000 alunos do ensino fundamental. Eles não sabem fazer derivadas. Mas se você der 2 números para cada um, eles somarão os 10.000 números **instantaneamente e ao mesmo tempo**.

A Inteligência Artificial baseia-se em quatrilhões de somas e multiplicações infantis. O ginásio sempre vence o professor.

PyTorch vs TensorFlow: Filosofia e Ecossistema

A principal diferença histórica está na **Filosofia de Design**.

PyTorch (Pesquisa & Acadêmico)

Código nativo Python, bastante intuitivo e transparente. Dominante absoluta em **Visão, NLP e LLMs** na academia.

```
import torch
x = torch.tensor([1.0, 2.0])
y = x * 2

model = MyModel()
output = model(x)
loss.backward()
```

TensorFlow (Produção & Edge)

Foco em *Deploy* corporativo (TF Serving, TF Lite). Graças ao **Keras**, hoje a curva de aprendizado caiu muito:

```
import tensorflow as tf
model = tf.keras.Sequential([
    tf.keras.layers.Dense(128,
        activation="relu")
])
```

GPU e TPU

Ambos trabalham perfeitamente em **GPUs NVIDIA** (`x.to("cuda")` vs `with tf.device("/GPU:0")`). O TensorFlow sempre dominou **TPUs** pelo vínculo com Google, mas o PyTorch já possui ótimo suporte hoje.

Qual Escolher? (A Analogia Automotiva)

Pense neles como carros:

- **PyTorch:** Direção manual e direta. Você sente a pista. Ótimo para *modificar o motor* e testar experimentos rapidamente.
- **TensorFlow:** Plataforma empresarial. É uma infraestrutura de larga escala, muito poderosa, mas exige interagir com mais componentes.

Vá de PyTorch se...

- Está começando do zero.
- O objetivo é **aprender** Deep Learning.
- Trabalha com pesquisa acadêmica.

Vá de TensorFlow se...

- A empresa já tem legado na stack.
- Escala corporativa massiva.
- Trabalha com embarcados (*Edge/Mobile*).

Atenção: Não é uma escolha para a vida toda!
Os conceitos (Tensores, Backpropagation, Loss, CNNs) são **os mesmos**.
Migrar de um framework para o outro será perfeitamente natural no futuro.

A Guerra do Hardware: GPU vs TPU

GPU + CUDA (ex: NVIDIA H100)

- Processador **altamente paralelo e flexível**.
- Criado para gráficos, perfeito para IA.
- Milhares de núcleos genéricos que rodam qualquer lógica matricial.
- Domínio imenso (**Ecossistema gigante**).

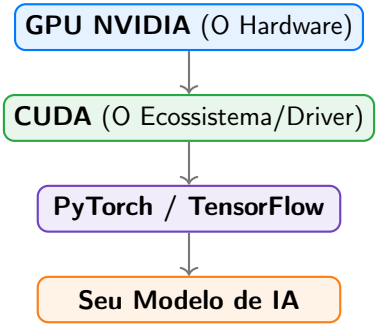
TPU (Google)

- Hardware **especializado** (*ASIC*).
- Feito do zero só para Machine Learning.
- Usa arquitetura *Systolic Array* (linhas de montagem de matrizes).
- Menos flexível, porém hiper eficiente.

Ambos treinam LLMs gigantescos de forma brilhante.
A escolha real depende de custo, disponibilidade na Nuvem vs Local e do Framework.

Desmistificando o CUDA

Atenção: CUDA não é uma placa de vídeo! É a plataforma de **Software** da NVIDIA que permite utilizar a GPU para computação.



Quando fazemos `model.to("cuda")`, estamos dizendo:
"Coloque o modelo no hardware GPU utilizando a linguagem CUDA".

O que usar no seu PC de casa?

Se você vai estudar IA em casa e possui uma placa de vídeo NVIDIA (ex: RTX 3060, 4070):

O Caminho Padrão

1. GPU NVIDIA → **2.** CUDA → **3.** PyTorch → **4.** Treino Local!

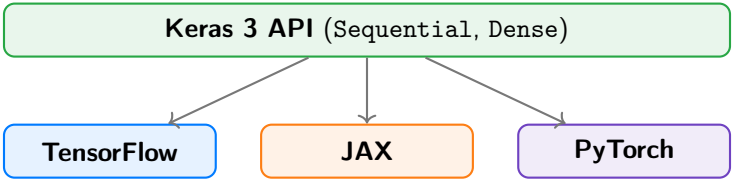
```
import torch
device = "cuda" if torch.cuda.is_available() else "cpu"
model = model.to(device)
```

E se eu não tiver TPU em casa?

Sem problemas. O PyTorch roda direto na sua NVIDIA. E se amanhã você precisar usar TPUs na nuvem, o projeto **PyTorch/XLA** permite que você rode o exato mesmo código PyTorch direto nos aceleradores do Google.

Keras 3.0: A Nova Era Multi-Backend

Atenção: Keras **não** é mais sinônimo de TensorFlow!
Hoje, o Keras 3 atua como uma API de alto nível capaz de rodar sobre os três maiores motores de IA do mundo:



Para a nossa disciplina

Usaremos **Keras + TensorFlow**. Você aprenderá os fundamentos de forma independente de framework. A mesma lógica servirá caso migre amanhã.

Trocando o motor do Keras (Ex: PyTorch)

Quer rodar o Keras sobre o PyTorch na sua GPU NVIDIA? Basta alterar a variável de ambiente (backend) com 1 linha de código:

```
import os
os.environ["KERAS_BACKEND"] = "torch" # ou "tensorflow" ou "jax"

import keras
model = keras.Sequential([
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Fluxo de execução (Por baixo dos panos):
Keras $\rightarrow$ PyTorch $\rightarrow$ CUDA $\rightarrow$ GPU NVIDIA.

Dica Prática: Na disciplina, use *TensorFlow* como planejado. Nos estudos extracurriculares em casa, experimente com

Keras: Orientação Humana à IA

Nos primórdios, multiplicar matrizes na GPU exigia +50 linhas de código árido e gestão manual de ponteiros de memória.

Para democratizar o uso, François Chollet criou o **Keras** (2015).

- Não é um motor de GPU, mas uma **API de alto nível** (um *wrapper*).
- Roda "por cima"do TensorFlow (e agora PyTorch e JAX).
- **Filosofia:** "Redes são pilhas de camadas. A programação deve refletir apenas as camadas, ocultando a matemática de baixo nível".

Sintaxe Keras: Uma Rede em 5 Linhas

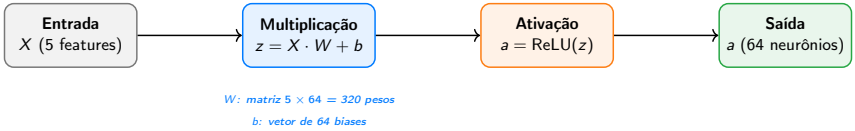
O equivalente a dezenas de matrizes e ReLUs manuais do NumPy:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()  # Container de rede em cascata
# 64 neurônios ocultos + ReLU (5 features de entrada)
model.add(Dense(64, activation='relu', input_shape=(5,)))
# Saída com Sigmoid (probabilidade 0 a 1)
model.add(Dense(1, activation='sigmoid'))
```

O que o Keras faz automaticamente

Inicializa pesos, aloca GPU, conecta camadas, prepara Backpropagation.

O que Acontece Dentro do `Dense`?



Cada `Dense(64)` é **exatamente** o que vocês fizeram na Aula 2:

- Multiplicação matricial $X \cdot W + b$ (Álgebra Linear).
- Ativação ReLU: $\max(0, z)$.
- O Keras faz isso em uma linha; vocês fizeram em 15.

Mapeamento NumPy Manual → Keras

NumPy (Aulas 1–3)	Keras (Aula 4)
`W1 = np.random.randn(3,4)`	`Dense(4, input_shape=(3,))`
`A1 = relu(np.dot(X,W1)+b1)`	`activation='relu'`
`for epoch in range(100):`	`model.fit(X, y, epochs=100)`
`gradiente = 2 * peso`	`Autograd (automático)`
`W -= lr * grad`	`optimizer='adam'`

Agora vocês entendem o que o Keras faz "por baixo dos panos" — esse é o diferencial de quem cursou as Aulas 1–3.

Dados do Mundo Real: Ingestão com Pandas

Na indústria, os dados vêm em **CSVs**, bancos SQL ou planilhas Excel. O `pandas` traduz tudo em *DataFrames*:

- Carregar: `df = pd.read_csv('dados.csv')`
- Separar Features: `X = df[['idade','salario']]`
- Separar Target: `y = df['churn']`

DataFrame `pandas`

idade	salário	churn
25	3.500	0
52	120.000	1
31	45.000	0

X **(Features)** y **(Target)**

Explorando o DataFrame (EDA Básica)

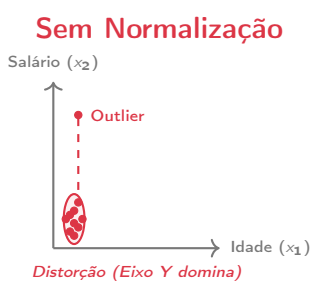
Antes de jogar os dados na rede neural, um engenheiro sempre **olha** para eles. O pandas fornece métodos essenciais para *Exploratory Data Analysis* (EDA):

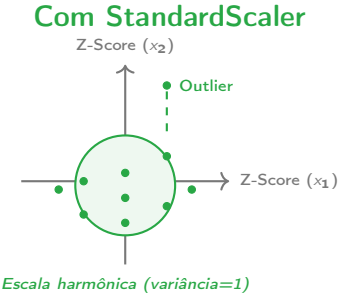
```
# Exibe as 5 primeiras linhas (garante que carregou certo)
print(df.head())

# Resumo estatístico (media, min, max, quartis)
print(df.describe())

# Verifica se há valores nulos faltando na tabela
print(df.isnull().sum())

# Conta quantos registros há de cada classe no Target
print(df['churn'].value_counts())
```

A Tirania da Escala (Normalização)

O Erro Fatal do Iniciante

Entregar à rede uma tabela onde Idade = 30 e Salário = 85.000.
A rede acha que Salário é "mais importante" — o gradiente explode!





Lei de Ouro: **Sempre normalize os dados antes de treinar.**

StandardScaler: A Matemática

$$z_i = \frac{x_i - \mu}{\sigma}$$

Idade	→	Z-Score
25		−1.22
31		−0.29
52		2.00

Salário	→	Z-Score
3.500		−1.05
45.000		−0.30
120.000		1.35

Após normalização, ambas as features ficam com $\mu \approx 0$ e $\sigma \approx 1$. A rede trata **todas** igualmente.

Outliers não são removidos!

O `StandardScaler` re-escala os dados, mas preserva suas distâncias relativas. Um outlier continuará sendo um outlier (valor de Z-Score isolado), mas sem explodir os pesos da rede matematicamente.

O Vocabulário do Scikit-Learn (A Sequência)

Para usar o `StandardScaler` sem cometer erros graves, você precisa entender 3 verbos fundamentais:

- `fit()`: O verbo **aprender**. O algoritmo olha os dados, tira uma "fotografia" e descobre quem é a média (μ) e o desvio padrão (σ). Ele *não modifica* os dados, apenas anota a fórmula.
- `transform()`: O verbo **aplicar**. Ele pega a fórmula que foi gravada no `fit()` e efetivamente substitui os números velhos pela nova escala.
- `fit_transform()`: Faz as duas coisas juntas num único passo. Aprende a média/desvio dos dados que você forneceu e já os altera em seguida.

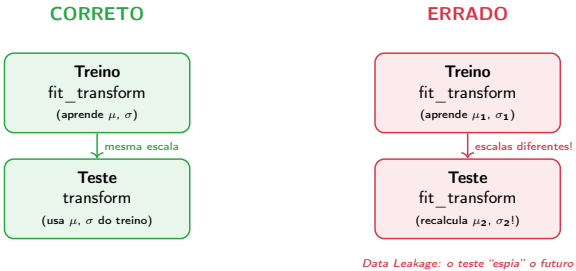
A Sequência Correta no Código

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

# 1. No TREINO: Aprende a escala do treino e ja aplica.
X_train_norm = scaler.fit_transform(X_train)

# 2. No TESTE: Aplica a MESMA formula gravada no passo 1.
X_test_norm = scaler.transform(X_test)
```

O Erro Fatal de Prova

Nunca use `fit_transform` nos dados de teste! Se você fizer isso, o scaler vai esquecer a escala do treino e inventar uma nova escala baseada nos dados de teste. A rede neural não vai entender nada.

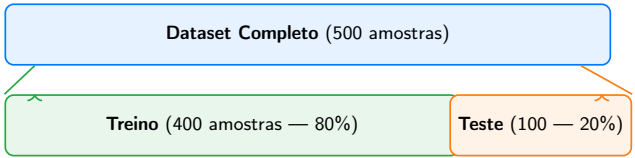
O Erro do `fit_transform` no Teste



Train/Test Split: Dividindo os Dados

Antes de normalizar, separamos treino e teste para avaliar a rede em dados "nunca vistos":

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)
# 80% treino, 20% teste
```



Compile & Fit: O Loop Completo em 2 Linhas

O que você construiu manualmente nas Aulas 2 e 3 vira:

```
# Compilar: define otimizador, loss e métricas
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Treinar: Forward + Loss + Backprop + Update (automatico!)
model.fit(X_train_norm, y_train, epochs=100)
```

Parâmetro	**O que faz**
`optimizer='adam'`	Gradient Descent inteligente (ajusta α automaticamente)
`loss='binary_crossentropy'`	Loss otimizada para classificação binária
`metrics=['accuracy']`	Log legível para humanos
`epochs=100`	Número de voltas no loop de treino

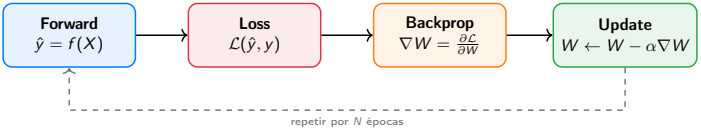
Lendo a Saída do Keras no Terminal

Ao rodar o `model.fit()`, o Keras exibe um relatório em tempo real do treinamento:

```
Epoch 1/100
13/13 [==============================] - 1s 5ms/step
- loss: 0.7021 - accuracy: 0.5120
Epoch 2/100
13/13 [==============================] - 0s 2ms/step
- loss: 0.6105 - accuracy: 0.6550
...
Epoch 100/100
13/13 [==============================] - 0s 2ms/step
- loss: 0.3214 - accuracy: 0.8950
```

- **13/13:** Número de lotes (*batches*) processados por época.
- **loss caindo e accuracy subindo:** O Gradient Descent está funcionando!

O que Acontece Dentro do `fit()`?



- Isso é **exatamente** o que vocês fizeram manualmente nas Aulas 2 e 3!
- O Keras apenas empacota tudo em uma chamada `model.fit()`.
- O Autograd calcula os gradientes automaticamente (sem derivadas manuais).

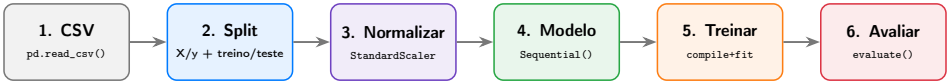
Avaliação: `evaluate()` e `predict()`

```
# Avaliar no conjunto de TESTE (dados nunca vistos)
loss, accuracy = model.evaluate(X_test_norm, y_test)
print(f"Acurácia no teste: {accuracy:.2%}")

# Prever probabilidades para novos dados
probabilidades = model.predict(X_test_norm)
# Ex: [[0.82], [0.15], [0.93], ...]
```

A Diferença

- `evaluate()` → calcula loss e acurácia (avaliação quantitativa).
- `predict()` → gera probabilidades para cada exemplo (inferência).

O Pipeline Completo: Do CSV ao Modelo Treinado



Este pipeline é o **esqueleto padrão** de qualquer projeto de IA com dados tabulares. Vocês o usarão em **todos os labs** desta disciplina.

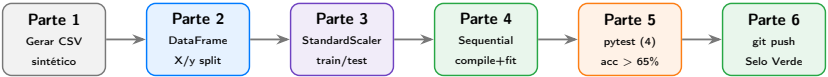
Recapitulação — O que Levar Desta Aula

1. A **GPU** acelera Deep Learning de semanas para horas (paralelismo massivo).
2. **Keras** é a API de alto nível do TensorFlow — define arquiteturas em 5 linhas.
3. **Pandas** ingere CSVs e organiza dados em DataFrames (X features, y target).
4. **StandardScaler** normaliza: $z = \frac{x-\mu}{\sigma}$, sempre `fit` no treino, `transform` no teste.
5. **compile() + fit()** = Forward + Loss + Backprop + Update automáticos.
6. **evaluate()** mede a qualidade real no conjunto de teste.

Próxima Aula

Até agora, nosso y é binário (0 ou 1). E se o problema tiver **3, 10 ou 100 classes**?
Classificação **Multiclasse** com Softmax e o temido **Overfitting**.

Missão: Treinar uma rede Keras em dados tabulares e atingir acurácia $> 65\%$.



- Dataset gerado com `make_classification` (sem dependências externas).
- Alvo: acurácia $> 65\%$ no teste (modelo precisa generalizar, não decorar).

Lab Passo 1: Gerando o Dataset

Para que todos tenham os **mesmos dados** no laboratório, não baixaremos um CSV. Vamos *gerá-lo matematicamente* usando o Scikit-Learn:

- Usaremos a função `make_classification`.
- Ela criará uma matriz de dados fictícios (ex: 500 linhas e 5 colunas).
- E um vetor alvo y (0 ou 1) que depende ocultamente das colunas, contendo um padrão a ser descoberto.

O Desafio

Você não sabe qual é a regra matemática que conecta o X ao y. Sua rede neural terá que descobrir sozinha durante o treinamento.

Lab Passo 2: O DataFrame e a Divisão

Com os arrays Numpy em mãos (gerados no Passo 1), vocês organizarão a "casa":

1. **Criar o DataFrame:** Injetar o array X no `pd.DataFrame` e dar nomes arbitrários às colunas (ex: F1, F2, F3...).
2. **Anexar o Target:** Criar uma nova coluna no DataFrame chamada `target` contendo o vetor y.
3. **Inspecionar:** Rodar um `df.head()` para garantir que a tabela está perfeita.
4. **Divisão (Split):** Usar o `train_test_split` para arrancar 20% da tabela e esconder. **A rede só pode treinar nos 80% restantes!**

Lab Passo 3: Normalização Implacável

Vocês aplicarão a **Lei de Ouro** que discutimos: nivelar as colunas numéricas.

- Instanciar o `StandardScaler()`.
- No conjunto de Treino: usar `fit_transform()` (ele aprende a média e já normaliza).
- No conjunto de Teste: usar `transform()` (ele usa a média memorizada do treino, para não trapacear vazando dados do futuro).

Se você esquecer dessa etapa, os gradientes explodem, a rede retorna `NaN` ou trava em 50% de acurácia.

Lab Passo 4: O Arquiteto Keras

Aqui começa a engenharia de IA moderna propriamente dita:

1. Iniciar um `model = Sequential()`.
2. Adicionar uma `Dense` inicial (ex: 32 neurônios) conectada às suas features. Não esqueça do `activation='relu'`.
3. Adicionar a `Dense` de saída: 1 único neurônio, pois é classificação binária. O ativador **obrigatório** é a `sigmoid`.
4. Chamar `compile()` dizendo que a loss é a `binary_crossentropy`.
5. Dar o comando de partida: `model.fit()` por 50 ou 100 épocas.

Lab Passo 5: Avaliação e Testes Unitários

No final do laboratório, você executará a célula do `pytest`. Ela validará seu código em frações de segundo:

- **Teste 1:** Verifica se o DataFrame tem o número correto de colunas e linhas.
- **Teste 2:** Verifica se você realmente dividiu 80%/20%.
- **Teste 3:** Inspeciona a normalização (se a média de X agora é perto de zero).
- **Teste 4 (O Chefe):** Checa se sua rede treinou e atingiu **mais de 65% de acurácia no Teste**. Se ficar abaixo, você reprova.

Lab Passo 6: CI/CD Pipeline (Selando a Aula)

Depois de passar localmente no Jupyter, você deve eternizar sua conquista:

```
git add .
git commit -m "Laboratorio 04 concluido - rede Keras"
git push origin master
```

No GitLab da disciplina, robôs baixarão seu código, criarão um servidor Linux temporário do zero, instalarão o TensorFlow, rodarão seu modelo e, se ele acertar $> 65\%$, você ganha um **Selo Verde** (Pipeline Passed). Essa é a sua nota!

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: Multiclasse & Overfitting

100% da Aula 4 Concluída!