Aula 4: Entrando no Mundo Real com o Ecossistema Keras

Aléssio Miranda Júnior

Agosto - 2026/2

O que você verá neste capítulo

Nas primeiras sessões deste curso, cimentamos as fundações da Inteligência Artificial em seu estado mais puro: as matrizes encadeadas via NumPy, as funções de ativação e a regra de atualização artesanal da descida do gradiente. Contudo, escalar esse código purista para arquiteturas industriais de bilhões de parâmetros é impraticável. Este capítulo oficializa a transição metodológica da Engenharia de IA: vamos aposentar a gestão direta de memória e abraçar os potentes *Frameworks* do mercado — em especial, o ecossistema **TensorFlow/Keras**. Mapearemos a hierarquia de abstrações (Hardware → TensorFlow → Keras), ingeriremos dados tabulares reais com `pandas`, aplicaremos a normalização obrigatória com `StandardScaler`, e treinaremos nosso primeiro modelo industrial com `compile()` e `fit()`.

1 A Ponte Perdida: A Topologia das Redes Neurais

Antes de mergulharmos em frameworks industriais de computação massiva, precisamos resolver uma lacuna conceitual profunda. Nas aulas anteriores, construímos o perceptron e compreendemos como a descida do gradiente atualiza um único peso. Contudo, surge a inevitável pergunta do arquiteto de software: *"Por que precisamos de tantas camadas ocultas? Por que usar quatro neurônios e não apenas um gigante?"*

A resposta reside na limitação matemática fundamental de um perceptron solitário: a **linearidade**.

1.1 A Limitação do Perceptron e o Problema do XOR

Um perceptron isolado é, em essência, um traçador de hiperplanos. No espaço 2D bidimensional, isso significa que a equação $w_1 x_1 + w_2 x_2 + b = 0$ consegue

apenas desenhar uma **única reta perfeita**. Se os dados forem "linearmente separáveis" (como um grupo de maçãs à esquerda e laranjas à direita), esse único neurônio resolve o problema magistralmente.

Entretanto, o mundo real e a complexidade humana raramente são linearmente separáveis. O exemplo clássico que implodiu a primeira era de ouro da IA na década de 1960 foi a simples porta lógica **XOR (Ou Exclusivo)**. No XOR, valores iguais resultam em 0 e valores diferentes resultam em 1. Se plotarmos isso num gráfico:

- Ponto A $(0,0)$ e Ponto D $(1,1)$ pertencem à Classe Vermelha.
- Ponto B $(0,1)$ e Ponto C $(1,0)$ pertencem à Classe Azul.

O desafio se torna um paradoxo geométrico: é matematicamente impossível desenhar uma *única* reta reta que separe as classes sem cortar a linha adversária. O perceptron falha miseravelmente.

1.2 A Mágica das Camadas Ocultas (*Hidden Layers*)

A solução para dados não-lineares é o **Multilayer Perceptron (MLP)**. Ao introduzirmos uma "camada oculta", injetamos novos neurônios na jogada.

Qual a intuição geométrica disso? Cada neurônio novo na camada oculta é capaz de desenhar a sua *própria* reta. No caso do XOR, a matemática estrita nos diz que apenas **dois neurônios** seriam teoricamente suficientes: eles traçariam duas retas paralelas formando um "corredor" que isolaria as classes centrais.

Contudo, na prática, treinar uma rede rasa para o XOR é desafiador e exige compreender a geometria e a função de ativação. Através da experimentação em plataformas como o *TensorFlow Playground*, observamos três grandes lições empíricas:

1. **A Instabilidade da Sigmoid em Redes Rasas:** Se configurarmos uma única camada oculta com 4 neurônios e ativação *Sigmoid*, o resultado é incerto. Treinamentos idênticos frequentemente geram fronteiras diferentes. Isso ocorre porque a Sigmoid cria divisões curvas e suaves, mas o processo de otimização (Gradiente Descendente) sofre enorme sensibilidade à inicialização aleatória dos pesos, ficando frequentemente preso em mínimos locais diferentes a cada execução.
2. **A Geometria Rígida da ReLU:** Ao trocar a função de ativação para a ReLU (que é *piecewise linear*), a rede passa a traçar polígonos com quinas secas e retas. Isso resolve o problema dos gradientes e da estagnação da Sigmoid. Contudo, em uma arquitetura rasa de apenas 1 camada oculta, a rede consegue isolar o centro do XOR, mas sofre muita interferência nas áreas vizinhas, falhando em produzir quadrantes perfeitamente puros e generalizados.

3 **A Solução Definitiva (Profundidade):** A verdadeira mágica do Deep Learning surge ao adicionarmos uma segunda camada oculta (por exemplo, 4 neurônios na primeira e 2 neurônios na segunda). Em vez de alargar excessivamente uma única camada, a rede compõe representações **hierárquicas**. A primeira camada (4 neurônios) aprende "conceitos primitivos", como linhas isoladas que cortam o plano. A segunda camada (2 neurônios) age como portas lógicas combinando essas linhas brutas. O resultado é uma fronteira geométrica perfeita, que isola os quatro quadrantes do XOR com confiança absoluta.

• Teoria: A Arte Empírica da Topologia

Eis a maior lição da Engenharia de IA: **a topologia final é ditada pela experimentação prática.**
Redes pequenas demais (subdimensionadas) não terão retas o suficiente para recortar a complexidade do problema. Redes imensas e rasas (superdimensionadas na largura) terão retas de sobra e vão contornar individualmente cada ponto, memorizando ruído (*Overfitting*). A intuição geométrica aponta que a **profundidade** organiza as abstrações hierarquicamente, convergindo de maneira muito mais limpa que a simples largura bruta.

Essa é a verdadeira complexidade da Inteligência Artificial. E calcular o *Backpropagation* manual (via NumPy) para todas essas dobras espaciais com matrizes gigantescas é uma tarefa impraticável. É exatamente para gerenciar e abstrair essa arquitetura complexa que introduziremos, agora, os *Frameworks* industriais.

2 O Teto Computacional do NumPy e da CPU

O `NumPy` é a biblioteca seminal do Python para ciência de dados. Ele gerencia blocos contíguos de memória escritos em C, permitindo velocidades de iteração altíssimas. Contudo, suas raízes baseiam-se na arquitetura de **Processamento Central (CPU)**.

As CPUs são prodigiosas em lidar com rotinas complexas e desvios condicionais (`if/else`), operando em frequências de *clock* estupendas em seus 8, 16 ou 32 núcleos. O problema inerente à *Deep Learning* é que não temos "rotinas complexas", mas sim quatrilhões de multiplicações simplórias de números (álgebra linear pura — exatamente o que vocês viram nas Aulas 1–3).

As GPUs (*Unidades de Processamento Gráfico*), por sua vez, são desenhadas exatamente para isso: renderizar milhões de pixels simples simultaneamente em seus mais de 10.000 micro-núcleos massivamente paralelos. Ao migrar da CPU (NumPy) para tensores de GPU, cortamos o tempo de treinamento de redes profundas de **semanas ininterruptas** para simples **horas**.

• Teoria: CPU vs GPU — A Analogia

Imagine que você precisa somar 10.000 números. A CPU é como um professor brilhante que soma um número de cada vez, mas com velocidade impressionante. A GPU é como um ginásio com 5.000 alunos medianos, cada um somando 2 números simultaneamente. Para álgebra linear massiva, a GPU vence esmagadoramente.

	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	∼2 semanas	∼4 horas

Tabela 1: Comparação CPU vs GPU para treinamento de redes neurais. Deep Learning = 99% multiplicação de matrizes, o que favorece enormemente o paralelismo da GPU.

2.1 A Abstração dos Grafos Computacionais

Para orquestrar o envio massivo de matrizes para a placa de vídeo, a indústria criou orquestradores gigantescos. Hoje, o duopólio do mercado se divide entre dois gigantes, cada um com uma filosofia distinta:

- **PyTorch (Meta/Facebook):** É guiado por uma filosofia estritamente pitônica e transparente (*eager execution*). O desenvolvedor manipula tensores e otimizadores diretamente com código muito legível. Por essa clareza, o PyTorch domina de forma quase absoluta a **Pesquisa Acadêmica** (modelos de Visão Computacional, NLP, LLMs e Transformers nascem primeiro no PyTorch). Se o ecossistema de Inteligência Artificial fosse uma oficina de carros, o PyTorch seria um veículo com direção mecânica direta: você sente cada engrenagem e pode alterar o motor a qualquer momento.
- **TensorFlow (Google Brain):** É o pioneiro corporativo, projetado fundamentalmente para a **Indústria** e **Deploy** em larga escala. Ele possui um ecossistema infraestrutural imbatível para colocar modelos em servidores (TF Serving), dispositivos móveis e embarcados (*Edge AI* via TFLite), e até mesmo no navegador (TF.js). Na nossa analogia automotiva, o TensorFlow seria uma gigantesca plataforma de frota corporativa, super poderosa, mas repleta de painéis de controle industriais.

Felizmente, a curva de aprendizado inicial entre os dois diminuiu drasticamente nos últimos anos. Com a popularização da API **Keras** como interface principal para o TensorFlow, definir uma rede tornou-se altamente intuitivo e fácil. Além disso, a paridade de hardware foi atingida: ambos os frameworks tiram proveito perfeito de GPUs NVIDIA, e hoje até mesmo a hegemonia das TPUs do Google (tradicionalmente exclusivas do TensorFlow) já suporta PyTorch plenamente.

Qual escolher na prática?

- Se você está começando do zero, quer aprender as minúcias das redes neurais ou está focado em pesquisa científica de novos algoritmos: **PyTorch**.
- Se a sua empresa já possui um legado na ferramenta, ou você atua como Engenheiro de ML focado em escalar modelos corporativos em celulares, microcontroladores ou servidores de alta concorrência: **TensorFlow**.

O detalhe fundamental é que você não precisa escolher um para sempre. Os conceitos internos (Tensores, Backpropagation, Funções de Custo) são matematicamente os mesmos. Aprender o segundo framework, após dominar os conceitos no primeiro, é um processo perfeitamente natural e direto.

O problema raiz desses grandes ecossistemas em suas origens (em especial o TensorFlow `1.x`) era a altíssima barreira e complexidade sintática — exigindo mais de 50 linhas de código apenas para multiplicar duas matrizes simples no hardware.

2.2 A Guerra do Hardware: GPU vs TPU

Para fechar a discussão de ecossistema, precisamos desmistificar a camada de hardware. O processamento de Redes Neurais é, essencialmente, a multiplicação massiva de gigantescas matrizes ($A \times B$).

Uma CPU tradicional possui relativamente poucos núcleos muito poderosos, o que a torna ineficiente para operações massivas em paralelo. Por isso, a **GPU (Graphics Processing Unit)** tornou-se o padrão: ela possui milhares de unidades de processamento menores. Originalmente concebidas pela NVIDIA para renderizar pixels em jogos, essas placas mostraram-se perfeitas para a álgebra da IA.

No entanto, é crucial distinguir a GPU do **CUDA**. CUDA não é uma placa de vídeo. É o ecossistema de *software* (a plataforma e API) da NVIDIA que permite que programas (como o PyTorch) acessem e instruam o hardware da GPU. A hierarquia funciona assim: a GPU é o hardware físico, o CUDA é o driver/tradutor, o PyTorch é o framework e, no topo, está o seu modelo de IA. Quando escrevemos `model.to("cuda")` no PyTorch, estamos dizendo: *"Envie este modelo para a GPU utilizando o tradutor CUDA"*.

Do outro lado, o Google criou a **TPU (Tensor Processing Unit)**. Diferente da GPU (que é flexível e faz cálculos matemáticos gerais), a TPU é um *ASIC*

(Circuito Integrado de Aplicação Específica) focado exclusivamente em IA. Ela utiliza uma arquitetura chamada *Systolic Array*, que atua como uma linha de montagem super otimizada para multiplicar e somar matrizes instantaneamente, abdicando de flexibilidade geral em prol de velocidade bruta para IA.

Qual utilizar em casa?

Se você possui um computador pessoal com uma placa NVIDIA (RTX 3060, 4070, etc.), o melhor caminho é, sem dúvidas, **PyTorch + CUDA**. Você pode rodar tudo localmente e com suporte nativo. O fato de você treinar na sua própria GPU não o impede de usar TPUs no futuro: o projeto **PyTorch/XLA** permite que exatamente o mesmo código escrito no seu quarto rode diretamente nos aceleradores TPU do Google Cloud. Aprender na GPU em casa não fecha, mas sim abre as portas para todos os hardwares de ponta da nuvem corporativa.

3 O Paradigma Keras: Orientação Humana

Para democratizar a IA, François Chollet projetou a **API Keras**. Uma mudança de paradigma fundamental ocorreu recentemente: **Keras não é mais sinônimo de TensorFlow**.

Na sua versão atual (Keras 3.0), ele atua como uma API de alto nível puramente arquitetural que roda perfeitamente sobre os três maiores motores matemáticos da indústria: TensorFlow, JAX e **PyTorch**. Isso significa que você pode escrever sua arquitetura de forma limpa e intuitiva e, com uma única linha de configuração (como `os.environ["KERAS_BACKEND"] = "torch"`), instruir o Keras a compilar e executar aquele modelo utilizando o motor do PyTorch, que por sua vez acessará a sua GPU NVIDIA via CUDA.

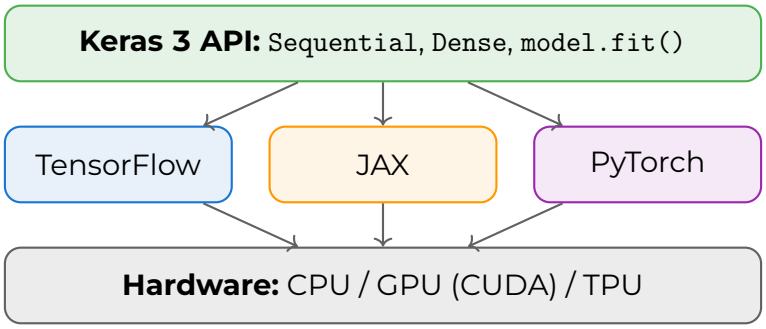


Figura 1: A arquitetura Multi-Backend do Keras 3.0. O engenheiro programa a lógica das camadas na interface Keras e escolhe qual motor fará o trabalho pesado no hardware por baixo dos panos.

Para fins didáticos, ao longo da nossa disciplina continuaremos utilizando a dupla clássica **Keras + TensorFlow** (que possui excelente integração nativa e é padrão da indústria corporativa). O que importa são os conceitos; aprendendo a lógica em Keras/TensorFlow durante as aulas, você estará perfeitamente

preparado para transpor todo esse conhecimento para PyTorch em estudos futuros.

Seu principal preceito é: “IAs são redes em camadas conectadas. Então a programação deve focar estritamente nisso”. Com a arquitetura declarativa do Keras, a criação complexa estudada na Aula 02 (Redes Profundas) se reduz a um código legível quase em linguagem humana:

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input

# Inicializa um container de rede vazia em cascata
model = Sequential()
model.add(Input(shape=(10,)))

# Camada oculta de 64 neurônios com ReLU (10 features de entrada)
model.add(Dense(64, activation='relu'))

# Camada de saída com 1 neurônio e Sigmoid (probabilidade)
model.add(Dense(1, activation='sigmoid'))
```

Este simples bloco instancia milhares de pesos, inicializa as matrizes estocasticamente, aloca espaço na memória da GPU e prepara as conexões do tensor de forma automatizada e protegida.

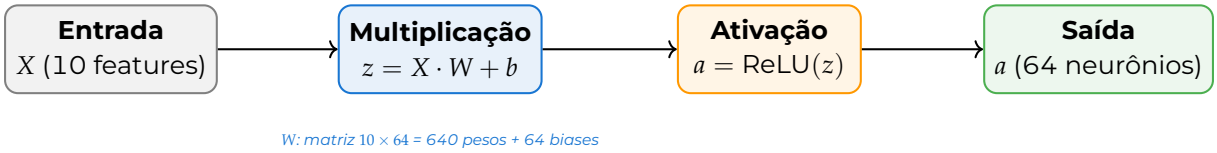


Figura 2: O que acontece dentro de uma camada `Dense(64, activation='relu')`: multiplicação matricial $X \cdot W + b$ seguida de ativação ReLU. É exatamente o que vocês fizeram manualmente na Aula 2.

▶ Prática: De NumPy para Keras

Compare o trabalho manual das Aulas 1–3 com o 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 domina os fundamentos.

4 O Mundo Real: Lidando com Dados (Data Ingestion)

Se os algoritmos já estão maduros e condensados no Keras, onde reside o verdadeiro trabalho do Engenheiro de Inteligência Artificial moderno? A resposta é inequívoca: **no tratamento rigoroso de dados sujos**.

Redes Neurais esperam ser alimentadas com números estritos organizados no formato exato de Tensores $(Amostras, Caractersticas)$. No mundo dos negócios, entretanto, os dados vêm em bancos relacionais (SQL), planilhas Excel corrompidas ou CSVs despadronizados.

A nossa primeira e mais poderosa ferramenta de engenharia para traduzir o mundo B2B para IA é o `pandas`. Através do `pandas`, ingerimos um arquivo CSV em um objeto relacional batizado de *DataFrame*, de onde fatiamos a tabela separando X e y.

• Teoria: A Divisão Fundamental: X e y

Toda Inteligência Artificial de aprendizado supervisionado exige que o Cientista divida o mundo em dois grupos:

- **X** (Features/Entradas): As colunas que detêm os atributos do problema (ex: Idade, Renda, Tempo na Plataforma).
- **y** (Target/Alvo): A coluna final, a resposta única que a IA precisa aprender a predizer (ex: O cliente encerrou a conta? [0 ou 1]).

4.1 Train/Test Split: Avaliando com Honestidade

Um modelo que é avaliado nos mesmos dados em que treinou é como um aluno que faz prova com o gabarito na mão — o resultado não significa nada. Por isso, a indústria adota a prática obrigatória de dividir o dataset em dois subconjuntos:

- **Treino (80%):** A rede aprende os padrões neste subconjunto.
- **Teste (20%):** A rede é avaliada em dados **nunca vistos** para medir a generalização real.

```
1 from sklearn.model_selection import train_test_split
2
3 X_train, X_test, y_train, y_test = train_test_split(
4     X, y, test_size=0.2, random_state=42)
```

5 A Tirania da Escala Numérica (Normalização)

Ao injetarmos o $\mathbf{X}$ puro no Keras, podemos desencadear o já estudado *Exploding Gradient*. Imagine um banco avaliando crédito, onde a coluna "Idade" varia de 18 a 60, e a coluna "Salário Bruto Anual" varia de 20.000 a 450.000.

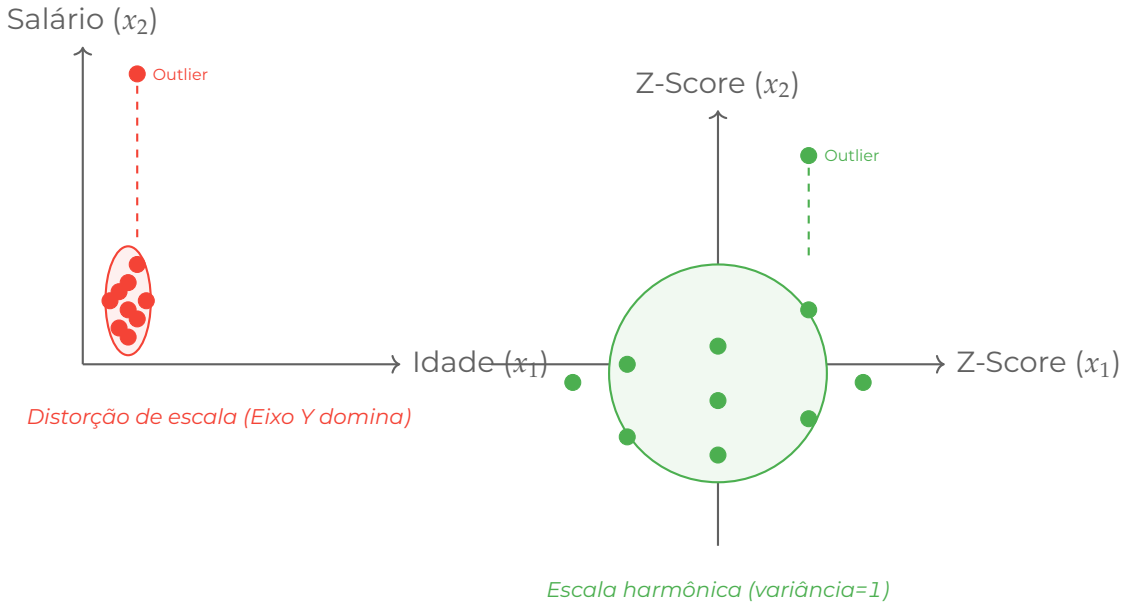


Figura 3: À esquerda, o espaço deformado por variáveis de escalas muito diferentes. À direita, a projeção padronizada centrada na média zero com variância unitária via `StandardScaler`.

Na matemática de IA, **números grandes têm peso magnético alto**. Como os pesos aleatórios (W) iniciam iguais para ambas as características, a rede neural dará prioridade instantânea ao "Salário" apenas por ele ter casas numéricas maiores, esmagando o conhecimento derivado da "Idade". Mais severamente, gradientes multiplicados por 450.000 explodirão a memória em nanosegundos (causando falha de NaN).

Para contornar essa distorção e estabilizar o gradiente descendente, nós introduzimos o **Scikit-Learn**, a principal biblioteca de Machine Learning clássico em Python. O Scikit-Learn atua aqui como uma camada de pré-processamento, limpando o terreno antes de passarmos a matriz para a rede neural profunda no Keras.

A ferramenta padrão da indústria para essa tarefa é o `StandardScaler`. Matematicamente, ele aplica a padronização **Z-Score** em cada característica individualmente: subtrai a média populacional da coluna e divide pelo seu desvio padrão ($z = \frac{x-\mu}{\sigma}$).

• Teoria: O que acontece com os Outliers?

É um equívoco comum acreditar que o `StandardScaler` recorta ou "limpa"dados anômalos. Como ilustrado na Figura 3, **o outlier não é removido nem comprimido contra a massa principal de dados**.
A transformação do Z-Score é puramente linear, o que significa que ela preserva a distância relativa original perfeitamente. A única diferença é que todo o espaço vetorial é re-centralizado na origem (a média se torna 0) e a dispersão é equalizada (o desvio padrão se torna 1). A rede neural continuará aprendendo que aquela linha específica representa uma anomalia gigantesca em relação aos demais, mas agora a operação ocorre de forma estabilizada (tipicamente em um intervalo de $[-3, +3]$), estancando a explosão matemática do gradiente.

△ Importante: Lei Inquebrável da IA

Nunca entregue dados em escala bruta à rede. Use as funções de Pré-Processamento do Scikit-Learn (`MinMaxScaler` ou `StandardScaler`). O escalonador comprimirá todas as colunas para um intervalo harmônico próximo de $[-1, 1]$.
Cuidado: Use `fit_transform()` apenas no treino. No teste, use apenas `transform()` com o scaler já ajustado — caso contrário, você estará vazando informação do futuro para o modelo.

▶ Prática: A Aplicação Prática no Scikit-Learn

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
# No treino, a IA aprende a distribuicao (media e desvio) E transforma:
X_train_norm = scaler.fit_transform(X_train)

# No teste, a IA APENAS aplica a transformacao,
# sem aprender a distribuicao dos dados novos:
X_test_norm = scaler.transform(X_test)
```

6 Compilando e Treinando (Compile & Fit)

Uma vez normalizados e divididos, os dados estão prontos para alimentar o Keras. O framework condensa o complexo loop `for` de 30 épocas da Aula 3 em um paradigma verbal elegante:

```
# Compilar: definir otimizador, loss e métricas
model.compile(optimizer='adam',           # Gradient Descent inteligente
              loss='binary_crossentropy', # Entropia Cruzada para classificacao binaria
              metrics=['accuracy'])       # Log legível para humanos
```

• Teoria: Por que não usar MSE na Classificação?

Note que trocamos a **MSE** pela **Binary Cross-Entropy (BCE)**. A MSE é ótima para *Regressão* (prever valores contínuos), mas terrível para classificação com Sigmoid/Softmax, pois cria gráficos de erro não-convexos e cheios de vales locais. A Entropia Cruzada avalia a distância entre distribuições de probabilidade, punindo de forma logarítmica quando a rede tem "alta certeza na resposta errada", forçando uma convergência muito mais limpa e robusta.

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

• Teoria: O Otimizador Adam

O `Adam` (Adaptive Moment Estimation) é uma evolução do Gradient Descent que ajusta o Learning Rate automaticamente para cada peso individual. Ele combina dois conceitos avançados: *momentum* (inércia na direção da descida) e *RMSProp* (normalização do gradiente por sua magnitude histórica). Na prática, é o otimizador *default* da indústria — funciona bem na maioria dos problemas sem necessidade de ajuste manual de α.

△ Importante: A Ilusão da "Caixa-Preta"

Embora o Keras e o otimizador Adam automatizem grande parte da matemática árdua, é um erro fatal para um engenheiro confiar cegamente no *framework*. O Adam ajusta dinamicamente a taxa de aprendizado durante a descida, mas **não** resolve os problemas estruturais: definir a quantidade de camadas e neurônios (capacidade da rede), o tamanho dos lotes de treinamento (*Batch Size*), e monitorar o risco da rede simplesmente decorar os dados em vez de aprender (*Overfitting*) ainda são responsabilidades exclusivas suas. Abordaremos o ajuste fino desses hiperparâmetros nas próximas aulas.

6.1 O Pipeline Completo

O fluxo de trabalho de um engenheiro de IA com Keras segue 6 etapas bem definidas:

Figura 4: O pipeline padrão de Machine Learning com Keras: da ingestão de dados ao modelo treinado.

Com apenas estes métodos, criamos modelos escaláveis e imunes à complexidade mecânica subjacente, focando totalmente a energia na estratégia do problema de negócio. Esta é a essência do profissional contemporâneo da IA.

7 Extensão: Classificação Multiclasse e Softmax

Até este ponto, modelamos um problema de classificação binária, onde a última camada tem apenas 1 neurônio e a função **Sigmoid** comprime a saída entre 0 e 1. Mas o mundo real é plural. Um sistema de visão computacional precisa distinguir entre dezenas de categorias (gato, cachorro, carro, avião...).

Para problemas com K classes mutuamente exclusivas, a arquitetura da camada de saída muda: passamos a ter K neurônios, e a função de ativação passa a ser a **Softmax**:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{1}$$

A Softmax converte um vetor de K valores brutos em uma **distribuição de probabilidades**. Cada saída fica entre 0 e 1, e a soma de todas as saídas é rigorosamente 1. A classe final predita é aquela com maior probabilidade.

▷ Exemplo: Softmax em Ação

Se a rede produz as saídas brutas $z = [2.0, 1.0, 0.1]$ para 3 classes, a Softmax calcula:

- Classe 0: $\frac{e^{2.0}}{e^{2.0}+e^{1.0}+e^{0.1}} \approx 0.659$ (65.9%)
- Classe 1: $\frac{e^{1.0}}{e^{2.0}+e^{1.0}+e^{0.1}} \approx 0.242$ (24.2%)
- Classe 2: $\frac{e^{0.1}}{e^{2.0}+e^{1.0}+e^{0.1}} \approx 0.099$ (9.9%)

A rede escolheria a Classe 0 com 65.9% de confiança.

Para treinar uma rede multiclasse, a Função de Perda adotada é a **Categorical Cross-Entropy**:

$$\mathcal{L} = -\sum_{i=1}^{K} y_i \cdot \log(\hat{y}_i) \tag{2}$$

No Keras, a transição para classificação multiclasse é cirúrgica:

```
1 # Camada de saida agora tem 10 neuronios e Softmax
```

```
model.add(Dense(10, activation='softmax'))

model.compile(
    optimizer='adam',
    loss='categorical_crossentropy', # Perda multiclasse
    metrics=['accuracy']
)
```

✔ Takeaways

- A **GPU** acelera o treinamento de Deep Learning de semanas para horas graças ao paralelismo massivo de seus milhares de micro-núcleos.
- O **Keras** é a API de alto nível do TensorFlow que abstrai grafos computacionais, alocação de memória e Autograd — o engenheiro define apenas a arquitetura.
- O **pandas** ingere dados tabulares (CSVs) em DataFrames, de onde separamos as Features (X) do Target (y).
- O **train_test_split** divide o dataset em treino (80%) e teste (20%) para garantir avaliação honesta.
- O **StandardScaler** normaliza os dados para evitar Exploding Gradient — **nunca** entregar dados brutos à rede.
- Os métodos **compile()** e **fit()** condensam todo o loop manual em 2 linhas de código industrial.
- A **Softmax** generaliza a Sigmoid para K classes, e a **Categorical Cross-Entropy** substitui a BCE para problemas multiclasse.

8 Conclusão

Neste capítulo, fizemos a transição fundamental de "artesão de matrizes" para "engenheiro de IA". O Keras automatiza tudo o que construímos manualmente nas Aulas 1–3 (inicialização de pesos, Forward Pass, Loss, Backpropagation, Update Rule), permitindo que o profissional foque no que realmente importa: a qualidade dos dados e a arquitetura do modelo. No próximo capítulo, enfrentaremos dois desafios que surgem naturalmente quando a rede começa a performar bem: a **classificação multiclasse** (Softmax) e o temido **Overfitting**.

✎ Questões: Autoavaliação

1 **[Direta]**Explique por que a GPU é mais eficiente que a CPU para treinar redes neurais profundas, relacionando com o tipo de operação matemática predominante.

2 **[Direta]**Um colega treinou um modelo Keras sem normalizar os dados e obteve `NaN` no loss. Explique tecnicamente por que isso aconteceu e como resolver.

3 **[Direta]**Qual a diferença entre usar `fit_transform()` e `transform()` no `StandardScaler`? Por que é um erro grave usar `fit_transform()` no conjunto de teste?

4 **[Reflexiva]**O Keras automatiza praticamente todo o processo de treinamento. Se é tão simples, por que investimos 3 aulas inteiras aprendendo a fazer tudo "na mão" com NumPy? Em que situações esse conhecimento faz diferença?

5 **[Reflexiva]**O otimizador Adam ajusta o Learning Rate automaticamente. Isso significa que o engenheiro pode ignorar completamente hiperparâmetros e confiar cegamente no framework? Discuta riscos.

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?"*

9 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:

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

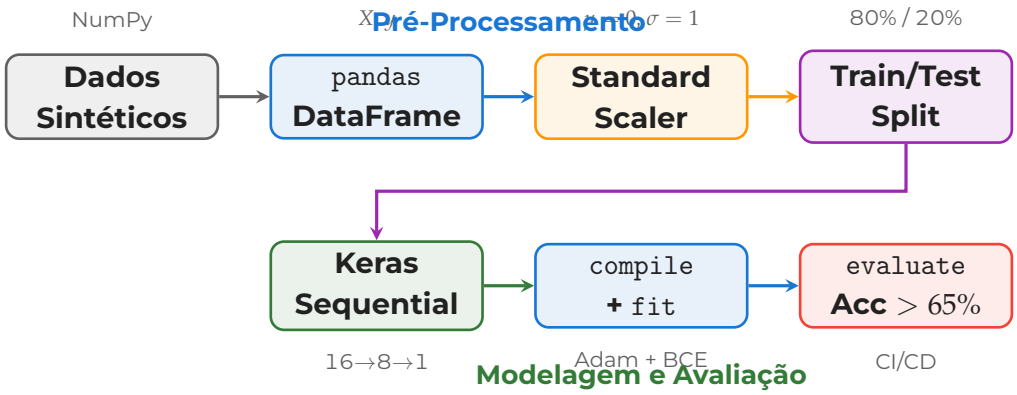
Modelo de Saída ($\hat{y}$): Probabilidade de churn via Sigmoid $\in (0, 1)$:

- $\hat{y} > 0.5 \quad \rightarrow$ 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**.

10 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

11 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	$[-1.7, +1.7]$

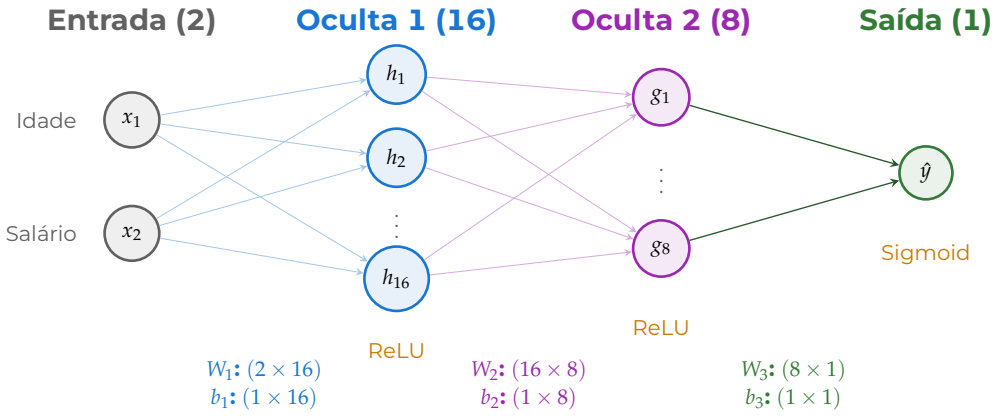
Sem normalização, o salário (escala de milhares) dominaria completamente a idade (escala de dezenas). O `StandardScaler` coloca ambas as features na mesma escala, com média ≈ 0 e desvio padrão ≈ 1.

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
```

12 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.

13 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.

14 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.

15 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→8→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?"*