Aula 2: Redes Profundas (MLP) e Forward Propagation

Aléssio Miranda Júnior

Agosto - 2026/2

O que você verá neste capítulo

Este capítulo expande a estrutura primordial do Perceptron para a arquitetura de Redes Neurais Artificiais Profundas, conhecidas como *Multilayer Perceptrons* (MLPs). Iniciaremos com o "Paradoxo do XOR" que causou o Inverno da IA, exploraremos a prova matemática do Colapso Linear (por que camadas lineares empilhadas colapsam em uma única matriz equivalente), e introduziremos as Funções de Ativação (Sigmoid e ReLU) como motores matemáticos que curvam o espaço vetorial. Por fim, a propagação progressiva (*Forward Propagation*) será detalhada passo a passo — primeiro com tensores NumPy, depois conectada ao ecossistema Keras/TensorFlow.

1 Introdução: Da Limitação à Topologia de Rede

Como vimos no desfecho do capítulo anterior, um neurônio artificial solitário (o Perceptron) esbarra em uma limitação geométrica fatal: o **Paradoxo do XOR**. Por ser um modelo puramente aditivo e linear, ele é incapaz de traçar fronteiras de decisão complexas, falhando em separar dados que exigem contexto não-linear.

Historicamente, essa constatação levou ao "Inverno da IA", congelando pesquisas por uma década. A resposta natural a essa barreira seria conectar múltiplos neurônios em uma rede topológica. Se um neurônio isolado desenha uma reta, múltiplos neurônios trabalhando em paralelo podem desenhar polígonos e aproximações curvas. Hoje, vamos abrir a caixa preta das Redes Neurais Artificiais Profundas para entender exatamente como isso é possível e como essas redes contornam a barreira do Perceptron simples.

2 A Solução: Multi-Layer Perceptron (MLP)

A arquitetura que resolveu o problema da separabilidade não-linear é o **Multi-Layer Perceptron (MLP)**. Uma rede MLP abandona o processamento centralizado em um único neurônio e cria um organograma computacional massivamente paralelo em etapas.

2.1 A Analogia do Comitê Corporativo

Para entender de forma simples, imagine que você é o Presidente de um banco e precisa decidir se aprova ou não um empréstimo gigantesco para uma empresa. Você não lê as centenas de documentos brutos. Em vez disso, a informação flui por uma hierarquia:

- **Os Documentos Brutos (Camada de Entrada):** Os balanços financeiros, processos jurídicos e lista de imóveis da empresa chegam ao banco.
- **Analistas Júniores (Camada Oculta 1):** Um grupo inicial olha partes específicas. O Analista A verifica apenas o histórico jurídico. O Analista B soma o valor dos imóveis. Eles não tomam decisões, apenas extraem informações básicas.
- **Gerentes Seniores (Camada Oculta 2):** Eles recebem os resumos dos Júniores e cruzam os dados, criando conceitos mais complexos. O Gerente cruza o "valor dos imóveis" com o "risco jurídico" para definir um "Grau de Confiabilidade Sólida".
- **A Decisão Final do Presidente (Camada de Saída):** O Presidente recebe o "Grau de Confiabilidade" mastigado pelos gerentes e apenas diz: **Aprovado ou Negado**.

É exatamente assim que o *Deep Learning* aprende. As primeiras camadas detectam padrões simples (ex: bordas em uma imagem), as camadas do meio cruzam esses padrões (ex: formatos de orelhas e focinhos), e a última camada decide (ex: "É um cachorro").

2.2 Feature Extraction: A Progressão Hierárquica

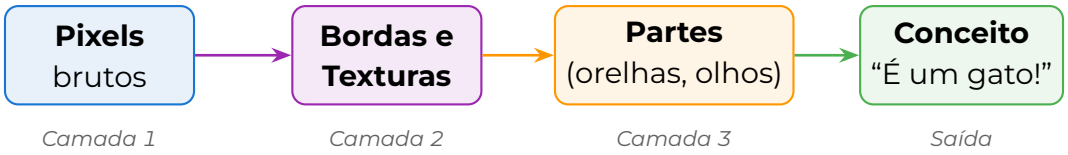


Figura 1: Progressão hierárquica da extração de padrões (*Feature Extraction*). Cada camada oculta detecta conceitos progressivamente mais abstratos — de pixels brutos até a decisão final.

2.3 Definição Técnica

Tecnicamente, essa arquitetura é dividida em três partes fundamentais:

- **Camada de Entrada (Input Layer):** É a porta de entrada dos tensores brutos. Não faz cálculos, apenas repassa as características numéricas para a próxima etapa. O número de "neurônios" aqui deve corresponder estritamente à dimensionalidade da amostra de entrada (N).
- **Camadas Ocultas (Hidden Layers):** Onde ocorre a extração de padrões. São fileiras de neurônios que recebem os dados da camada anterior, multiplicam por seus próprios pesos e geram novos padrões conceituais. Quanto mais camadas encadeadas, mais "profunda" é a rede.
- **Camada de Saída (Output Layer):** O neurônio (ou conjunto deles) final que consolida as avaliações e fornece a predição probabilística definitiva.

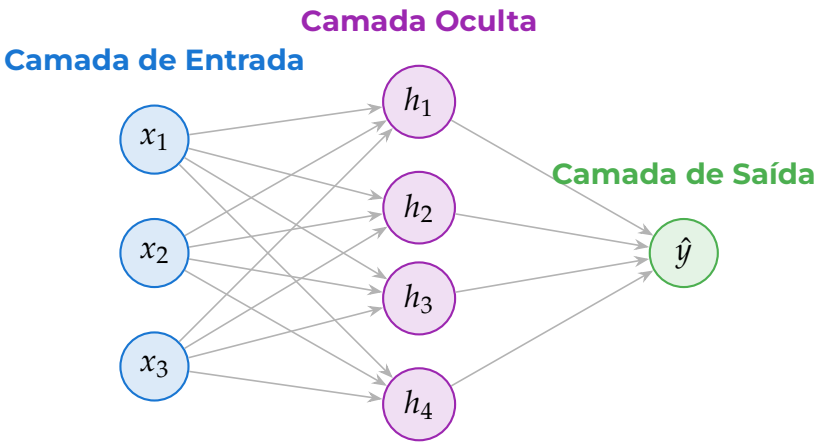


Figura 2: Arquitetura de uma Rede Neural Multicamadas (MLP) densamente conectada (*Fully Connected*). Cada neurônio de uma camada se conecta a todos da camada seguinte.

3 O Colapso Linear e o Teorema da Composição

Poderíamos imaginar que empilhar dezenas dessas camadas automaticamente tornaria a rede "superinteligente". Mas há uma armadilha fatal. Suponha que liguemos a Camada 1 a uma Camada 2 puramente através do produto matricial, sem nenhuma outra intervenção matemática. A predição $\hat{y}$ seria:

$$\hat{y} = (\mathbf{X} \cdot \mathbf{W}_1) \cdot \mathbf{W}_2 \tag{1}$$

A Álgebra Linear nos ensina a propriedade associativa da multiplicação de matrizes. Isso significa que podemos reescrever a equação isolando os tensores de pesos:

$$\hat{y} = \mathbf{X} \cdot (\mathbf{W}_1 \cdot \mathbf{W}_2) \tag{2}$$

Uma vez que o produto de duas matrizes fixas ($\mathbf{W}_1$ e $\mathbf{W}_2$) resulta em uma terceira matriz preexistente equivalente (vamos chamá-la de $\mathbf{W}_{eq}$), provamos categoricamente que:

$$\hat{y} = \mathbf{X} \cdot \mathbf{W}_{eq} \tag{3}$$

△ Importante: Conclusão Fatal

Uma rede de 100 camadas conectadas em sequência, cujas operações baseiam-se unicamente em produtos matriciais lineares, **colapsa matematicamente para um único Perceptron linear**. A rede não ganha "profundidade" real, perdendo toda e qualquer capacidade de desenhar fronteiras de decisão complexas. Isso explica por que apenas empilhar camadas sem ativação é inútil.

4 Funções de Ativação: Dobrando o Espaço Vetorial

Para impedir o colapso, a engenharia matemática exige que injetemos uma operação **não-linear** estritamente no meio do encadeamento, isto é, imediatamente após o cálculo de cada camada. É ela quem impede que a multiplicação se funda na camada seguinte, introduzindo assimetrias e quebras na linearidade (Teorema da Aproximação Universal).

4.1 O Legado do Perceptron: A Função Limiar (Step)

No modelo clássico do Perceptron, a "maquininha" de ativação era uma **Função Limiar** (degrau), que dispara 1 se a soma Z atingir um determinado limite (threshold), e 0 caso contrário. Embora perfeita para criar fronteiras perfeitamente retas (como prever Aprovação/Reprovação isolando alunos bons e ruins com uma nota de corte), o degrau rígido é inflexível. Ele não permite que a rede dobre o espaço ou faça nuances probabilísticas, forçando-nos a usar alternativas contínuas (curvas e rampas) nas Redes Profundas.

4.2 A Função Sigmoid

Historicamente, a Sigmoid (ou curva logística) dominou os primórdios das redes MLP:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

Ela mapeia qualquer valor de entrada para o intervalo probabilístico contínuo entre 0 e 1. Contudo, a Sigmoid sofre de um defeito grave conhecido como *Desaparecimento do Gradiente (Vanishing Gradient)*. Para valores escalares muito altos ou muito baixos, a curva satura, sua derivada tende a zero, e a rede

paralisa o próprio aprendizado, impossibilitando a construção de redes muito profundas.

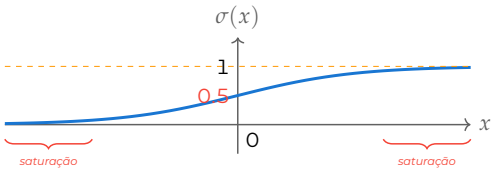


Figura 3: A Curva Logística (Sigmoid). Note como os valores nas extremidades geram uma curva quase plana (derivada tendendo a zero), causando o problema de *Vanishing Gradient*.

4.3 A Função ReLU (Rectified Linear Unit)

A revolução do Deep Learning moderno só foi possível graças à popularização da *ReLU*. Ela possui uma lógica matematicamente trivial, mas computacionalmente brilhante:

$$f(x) = \max(0, x) \tag{5}$$

Se a soma ponderada de um neurônio for negativa, a ReLU imediatamente desliga o sinal (passa 0). Se for positiva, o sinal flui intacto, preservando integralmente o gradiente (derivada exata igual a `1` para o lado positivo).

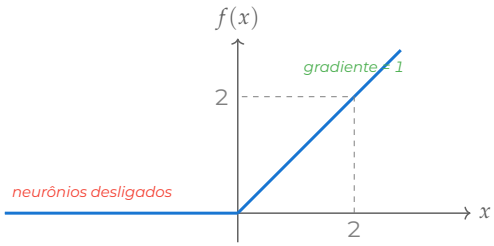


Figura 4: A Função ReLU. Simples e eficiente: zera tudo que for negativo e preserva intacto tudo que for positivo, evitando o problema de *Vanishing Gradient*.

▶ Prática: A Mágica da ReLU no NumPy

Implementar a não-linearidade que move o estado da arte das IAs na biblioteca NumPy não exige cálculos transcendentais. Basta o comando vetorizado `np.maximum(0, matriz_Z)`, que instantaneamente varre o tensor e poda valores negativos.

5 Forward Propagation: A Viagem da Informação

O fluxo das informações, saindo da entrada, recebendo ativação, multiplicando pela camada seguinte, repetidamente até a saída é o famigerado *Forward Pass*. Antes de vermos o código, visualizemos a cascata completa:

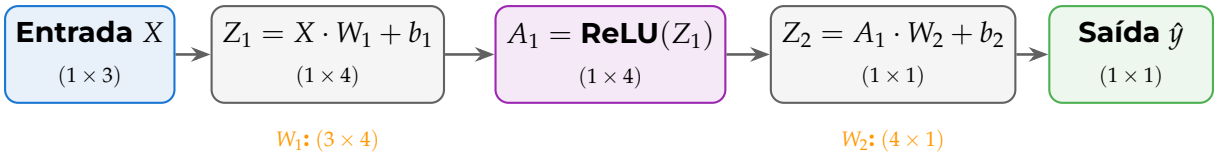


Figura 5: Pipeline completo do *Forward Propagation* em uma MLP de 2 camadas. As dimensões (*shapes*) estão anotadas abaixo de cada etapa para facilitar o rastreamento dimensional.

5.1 Passo-a-Passo Numérico

Para consolidar a intuição, vamos calcular manualmente um Forward Pass completo:

▷ Exemplo: Cálculo Manual do Forward Pass

Dados de entrada: $X = [1.0,\ 2.0]$, Pesos: $W_1 = \begin{bmatrix} 0.5 & -1.0 \\ 0.2 & 0.8 \end{bmatrix}$, Viés: $b_1 = [0.1,\ 0.1]$

Passo 1 — Multiplicação matricial:

$$Z_1 = [(1)(0.5) + (2)(0.2), \quad (1)(-1.0) + (2)(0.8)] + [0.1,\ 0.1]$$
$$Z_1 = [0.9, \quad 0.6] + [0.1,\ 0.1] = [1.0,\ 0.7]$$

Passo 2 — Ativação ReLU:

$$A_1 = \text{ReLU}([1.0,\ 0.7]) = [1.0,\ 0.7] \quad \text{(Ambos positivos!)}$$

5.2 Forward Propagation via Código Python

Escrito em linguagem Python pura (utilizando apenas a álgebra do NumPy), o encadeamento processando um *Batch* (lote) de alunos com **ReLU** na camada oculta e **Sigmoid** na saída fica assim:

```python
import numpy as np

def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))

# 1. Definindo o Batch de Entrada X (Ex: 2 Alunos, 3 features)
X = np.array([[1.0, 2.0, -1.0],
              [0.8, 0.5,  0.7]])

# 2. Pesos da Camada Oculta H1 (3 features para 4 neurônios)
W1 = np.random.randn(3, 4)
b1 = np.zeros((1, 4))

# --> Forward H1
Z1 = np.dot(X, W1) + b1
A1 = np.maximum(0, Z1)  # Ativação ReLU Aplicada!

```

```
# 3. Pesos da Camada de Saída (4 ocultos para 1 predição binária)
W2 = np.random.randn(4, 1)
b2 = np.zeros((1, 1))

# --> Forward Final
Z2 = np.dot(A1, W2) + b2
A2 = sigmoid(Z2) # Probabilidade de cada aluno passar!
```

Listing 1: Forward Propagation completo de uma MLP para Classificação Binária processando um Batch via NumPy.

5.3 A Gestão de Tensores (*Shapes*)

O pesadelo do iniciante em IA não é a matemática em si, mas a colisão de matrizes incompatíveis (o erro `ValueError: shapes not aligned`). A regra de ouro é:

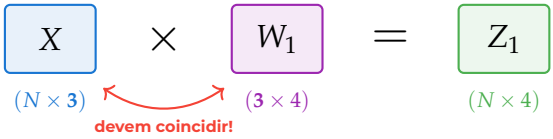


Figura 6: Regra de ouro: a dimensão interna de X e W_1 deve ser idêntica. O domínio das dimensões define a maturidade do Engenheiro de IA.

6 O Ecossistema Keras e TensorFlow

Construir matrizes manualmente via `NumPy` provou que o *Forward Propagation* nada mais é do que fornos em série assando dados. Entretanto, quando lidamos com milhares de matrizes de alta ordem de grandeza espacial, Python puro engasga.

É por isso que a indústria adota orquestradores massivos em C++ ou CUDA (que rodam direto na placa de vídeo, GPU). O **TensorFlow**, criado pelo Google Brain, assumiu a liderança na compilação eficiente desses grafos paralelos. Contudo, a notação do TensorFlow puro exige uma prolixidade assustadora.

Para que humanos consigam expressar as ideias em poucas linhas sem se afogarem nas declarações computacionais, François Chollet desenhou a **Keras** — uma API de alto nível (*High-Level Wrapper*) que encapsula toda a dor matemática em comandos extremamente semânticos.

A ponte entre a fundação matemática do NumPy e a sintaxe escalável do mercado moderno fica evidente na comparação a seguir:

```
from tensorflow import keras

modelo = keras.Sequential([
    keras.layers.Dense(4, activation='relu', input_shape=(3,)),
    keras.layers.Dense(1)
])

```

```
# O forward pass agora é uma unica linha:
predicao = modelo.predict(X)
```

Listing 2: A mesma MLP de 2 camadas expressa em Keras — a topologia neural é declarada sem lidar com matrizes W ou bias b diretamente.

Observe como a Keras abstraiu completamente as operações de inicialização de W_1, b_1, W_2, b_2 e as multiplicações matriciais. A linha `Dense(4, activation='relu')` encapsula exatamente o que escrevemos manualmente: *crie 4 neurônios ocultos, aplique ReLU*. No entanto, é fundamental entender que a Keras não substitui o conhecimento matemático — ela o **empacota**.

✔ Takeaways

- Um único Perceptron falha em problemas não-linearmente separáveis (como o XOR), o que exige o uso de múltiplas camadas (MLP).
- O *Forward Propagation* em uma MLP é essencialmente uma cascata de multiplicações de matrizes que fluem da Entrada para a Saída.
- Sem Funções de Ativação, empilhar infinitas camadas equivale matematicamente a uma única camada (**Colapso Linear**).
- A função Sigmoid foi fundamental no passado, mas sofre de "Desaparecimento do Gradiente" (saturação) nas camadas ocultas.
- A ReLU ($f(x) = \max(0, x)$) é o padrão da indústria moderna, pois zera valores negativos e propaga perfeitamente o gradiente dos positivos.
- O Keras encapsula a álgebra matricial em declarações semânticas, mas **não substitui** o entendimento fundamental.

7 Conclusão

Nesta aula, escalamos do Perceptron solitário para a arquitetura MLP, provamos que camadas lineares empilhadas colapsam em uma só, descobrimos as funções de ativação como a peça que faltava, e rastreamos a viagem da informação pelo *Forward Pass*. No próximo capítulo, enfrentaremos a pergunta central que falta: *como os pesos W são ajustados?* A resposta vem da **Descida do Gradiente** e do **Backpropagation**.

✎ Questões: Autoavaliação

1 **[Direta]**Demonstre algebricamente por que uma rede neural com 50 camadas ocultas puramente lineares é matematicamente equivalente a uma rede de apenas 1 camada.

2 **[Direta]**Explique a diferença de comportamento matemático entre as funções de ativação Sigmoid e ReLU quando recebem como entrada o valor -10. E para o valor $+10$?

3 **[Direta]**Descreva o papel de cada uma das três estruturas principais de uma MLP (Camada de Entrada, Camadas Ocultas e Camada de Saída) utilizando a analogia do comitê corporativo.

4 **[Reflexiva]**Se a ReLU simplesmente zera qualquer valor negativo, perdendo parte da informação original, por que ela funciona tão incrivelmente bem para construir o conhecimento das IAs de visão computacional e texto?

5 **[Reflexiva]**O uso de bibliotecas de alto nível como Keras e TensorFlow "esconde" a álgebra matricial do desenvolvedor. Em que cenário a dependência exclusiva dessas bibliotecas pode se tornar um risco crítico para um engenheiro de IA tentando otimizar ou debugar uma arquitetura inovadora?

Lab. 02: Redes Profundas (MLP)

⋆ Objetivo do Laboratório

Nesta atividade, você construirá sua primeira rede neural multicamadas (MLP), programando o *Forward Propagation* com tensores do NumPy. Ao final, sua rede será capaz de receber uma entrada, propagá-la por duas camadas e devolver uma predição — tudo com matemática vetorial pura. É o momento de provar que você domina o casamento dimensional de matrizes (*Shapes*) e a aplicação de funções de ativação não-lineares.

8 Parte 1: O Desafio da DeepTech

Na aula teórica, vimos que um único neurônio (Perceptron) não consegue resolver problemas não-lineares como o XOR. A solução é empilhar neurônios em camadas — o que chamamos de **Multilayer Perceptron (MLP)**.

Problema B: A Peneira de Talentos

Contexto: A empresa *DeepTech* recebe milhares de currículos para a vaga de Engenheiro de IA. O RH pediu a você que construa uma **Rede Neural MLP** que receba as notas de um candidato e retorne a **probabilidade** dele ser aprovado para entrevista.

Modelo de Entrada (X): Vetor com 3 métricas de avaliação (valores contínuos entre 0 e 1):

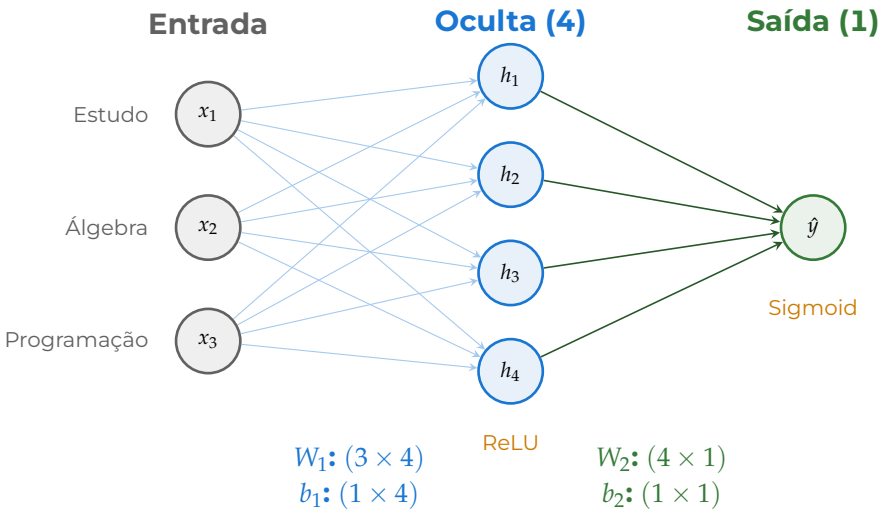
- x_1: Horas de Estudo Autodidata (normalizado)
- x_2: Conhecimento Prévio em Álgebra (normalizado)
- x_3: Teste Prático de Programação (normalizado)

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

- $\hat{y} > 0.5 \quad \rightarrow$ Candidato recomendado para entrevista
- $\hat{y} \leq 0.5 \quad \rightarrow$ Candidato em espera

A. A Arquitetura da Rede

Os engenheiros seniores da *DeepTech* já definiram a topologia da rede. Observe o diagrama abaixo — sua tarefa é **traduzir essa arquitetura em código NumPy**:



• Teoria: A Regra de Ouro dos Shapes

A dimensão interna deve sempre **coincidir**. Se a entrada X é $(N, 3)$ e a camada oculta tem 4 neurônios, então W_1 obrigatoriamente tem shape $(3, 4)$. A saída parcial será $(N, 4)$. Erre essas dimensões e o NumPy gritará: `ValueError: shapes not aligned`.

9 Parte 2: Funções de Ativação (O Fim do Degrau)

No Lab 01, usamos a Função Degrau — rígida e binária. Redes profundas precisam de curvas suaves para "dobrar o espaço" e aprender fronteiras complexas. Abra o arquivo `src/mlp.py` no seu repositório e implemente as duas funções de ativação:

```python
import numpy as np

def relu(x):
    """Rectified Linear Unit (ReLU).
    Zera negativos, mantém positivos. Quebra a linearidade!"""
    return np.maximum(0, x)

def sigmoid(x):
    """Função Sigmoid (Curva em S).
    Esmaga a saída para o intervalo (0, 1) -> probabilidade."""
    return 1.0 / (1.0 + np.exp(-x))
```

10 Parte 3: A Camada Densa e o Forward Pass

Cada camada da rede executa a mesma operação: $Z = X \cdot W + b$. Em vez de repetir código, vamos encapsular isso em uma classe `CamadaDensa`. Adicione ao seu `src/mlp.py`:

```python
class CamadaDensa:
    """Camada Densa (Fully Connected) de uma MLP."""

    def __init__(self, n_entradas, n_neuronios):
        # Inicialização He (boa prática para ReLU)
        self.pesos = np.random.randn(n_entradas, n_neuronios) \
                     * np.sqrt(2.0 / n_entradas)
        self.vieses = np.zeros((1, n_neuronios))

    def forward(self, entradas):
        """Forward propagation: Z = X . W + b"""
        return np.dot(entradas, self.pesos) + self.vieses
```

△ Importante: Atenção à Inicialização

Usamos a **Inicialização He** (`* np.sqrt(2.0 / n_entradas)`) em vez de pesos puramente aleatórios. Isso evita que os valores explodam ou desapareçam conforme a rede fica mais profunda — um problema real chamado *vanishing/exploding gradients* que vocês estudarão na próxima aula.

Montando a Rede Completa

Agora, no bloco `if __name__`, monte a rede conforme o diagrama e propague um candidato de exemplo:

```python
if __name__ == "__main__":
    # Candidato fictício: Estudo=0.8, Álgebra=0.6, Programação=0.9
    X = np.array([[0.8, 0.6, 0.9]])  # shape: (1, 3)

    # Montar as camadas conforme o diagrama
    camada_oculta = CamadaDensa(n_entradas=3, n_neuronios=4)
    camada_saida  = CamadaDensa(n_entradas=4, n_neuronios=1)

    # Forward Pass: Entrada -> Oculta (ReLU) -> Saída (Sigmoid)
    Z1 = camada_oculta.forward(X)  # (1,3) x (3,4) = (1,4)
    A1 = relu(Z1)                  # Ativação ReLU

    Z2 = camada_saida.forward(A1)  # (1,4) x (4,1) = (1,1)
    A2 = sigmoid(Z2)               # Probabilidade final

    print(f"Entrada:        {X}")
    print(f"Oculta (ReLU):  {A1} -> shape {A1.shape}")
    print(f"Probabilidade:  {A2[0,0]:.4f}")
```

Exemplo Numérico Passo-a-Passo

Para que você entenda exatamente o que cada linha faz, considere que a Inicialização He gerou os seguintes pesos fictícios (na prática, serão diferentes a cada execução):

Operação	Cálculo	Shape
Entrada X	$[0.8,\ 0.6,\ 0.9]$	$(1,3)$
$Z_1 = X \cdot W_1 + b_1$	Produto escalar + viés	$(1,4)$
$A_1 = \text{ReLU}(Z_1)$	Zera negativos	$(1,4)$
$Z_2 = A_1 \cdot W_2 + b_2$	Produto escalar + viés	$(1,1)$
$\hat{y} = \text{Sigmoid}(Z_2)$	$\frac{1}{1+e^{-Z_2}}$	$(1,1)$

O importante é verificar que os **shapes casam** em cada etapa — essa é a habilidade primária do Engenheiro de IA.

11 Parte 4: Garantia da Qualidade — Testes Unitários

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

```python
import numpy as np
from src.mlp import sigmoid, relu, CamadaDensa

def test_funcoes_ativacao():
    """Garante que as ativações se comportam como nos slides"""
    # Sigmoid de 0 deve ser exatamente 0.5
    np.testing.assert_almost_equal(
        sigmoid(np.array([0.0])), np.array([0.5])
    )

    # ReLU zera negativos, mantém positivos
    arr = np.array([-2.0, 0.0, 2.0])
    np.testing.assert_array_equal(
        relu(arr), np.array([0.0, 0.0, 2.0])
    )

def test_camada_densa_shape():
    """Garante que o shape da saída está correto"""
    camada = CamadaDensa(n_entradas=4, n_neuronios=8)
    X = np.ones((2, 4))  # Batch de 2 amostras, 4 features
    saida = camada.forward(X)
    assert saida.shape == (2, 8), \
        f"Shape esperado (2,8), obteve {saida.shape}"
```

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

12 Parte 5: Commit, Push e CI/CD

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

```
git add src/mlp.py
git commit -m "Lab 02: Forward Pass MLP com ReLU e Sigmoid"
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 `import numpy as np`, (2) nome de classe/função diferente do esperado (`CamadaDensa` com C maiúsculo), (3) shape incompatível na multiplicação. O CI roda os **mesmos testes** que você rodou localmente.

Conclusão: O que Você Construiu Hoje

✔ Takeaways

- Você implementou do zero uma **Rede Neural Multicamadas (MLP)** com 2 camadas usando apenas NumPy — sem frameworks, sem caixas pretas.
- Sua rede aplica a **ReLU** na camada oculta (quebrando a linearidade) e a **Sigmoid** na saída (produzindo probabilidades).
- Ela processa tanto amostras individuais quanto **batches inteiros** graças à álgebra matricial vetorizada.
- A classe `CamadaDensa` encapsula a operação $Z = X \cdot W + b$, tornando o código reutilizável e extensível.

◇ Informação

Gancho para a Próxima Aula: Sua rede propaga informações perfeitamente, mas os pesos W_1 e W_2 ainda são **inicializados aleatoriamente** — ela está "chutando". Na próxima aula, vamos ensinar a rede a *aprender*, ajustando esses pesos automaticamente usando a **Descida do Gradiente** e o **Backpropagation**. A pergunta será: *"quanto cada peso contribuiu para o erro?"*