# Lab. 02: Redes Profundas (MLP) e o Forward Pass

Aléssio Miranda Júnior

Agosto - 2026/2

## 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.

## 1 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):

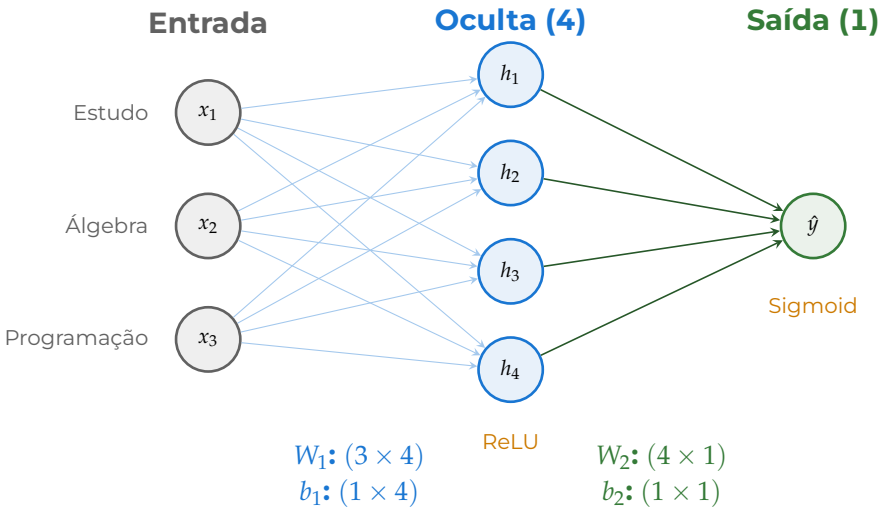
- $x_1$: Horas de Estudo Autodidata (normalizado)
- $x_2$: Conhecimento Prévio em Álgebra (normalizado)
- $x_3$: Teste Prático de Programação (normalizado)

**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`.

## 2 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))
```

## 3 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.

# 4 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.

# 5 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?"*