# Aula 3: Como a Máquina Aprende (Descida do Gradiente)

Aléssio Miranda Júnior

Agosto - 2026/2

## O que você verá neste capítulo

As redes neurais não seriam sistemas de "inteligência" se fossem incapazes de ajustar seus parâmetros de forma autônoma. Nas aulas anteriores, nossa arquitetura apenas propagava dados para frente de maneira passiva (*Forward Propagation*). Este capítulo rompe com o modelo estático e introduz a mecânica da Descida do Gradiente (*Gradient Descent*). Mapearemos a Função de Perda (MSE), revisitaremos os conceitos vitais de Cálculo I e III (derivadas e gradientes) através de analogias intuitivas, detalharemos a heurística da atualização (*Learning Rate*), demonstraremos o algoritmo de *Backpropagation* com um exemplo numérico completo e a metáfora das engrenagens, e analisaremos os riscos reais do treinamento: o *Exploding Gradient* e o *Vanishing Gradient*.

## 1 O Problema da Otimização: Avaliando o Próprio Erro

Na etapa de Propagação (Aula 2), a nossa rede neural recebe matrizes de Entrada $\mathbf{X}$ e gera uma predição $\hat{y}$. Contudo, como os tensores de pesos $\mathbf{W}$ e viés $\mathbf{b}$ são inicializados com ruído aleatório gaussiano (por exemplo, via `np.random.randn`), as predições geradas no instante $t = 0$ são puramente estocásticas e altamente errôneas.

O aprendizado de máquina nada mais é do que o **processo de otimização matemática iterativa** que visa a redução gradual desse erro. Para isso, precisamos, antes de tudo, quantificá-lo de forma que o computador entenda a gravidade da sua falha.

## 1.1 A Função de Perda (Loss Function): MSE

Para sabermos o quanto a nossa rede está errando, instituímos a **Função de Perda** (ou *Loss Function*). Ela compara a previsão feita pela rede ($\hat{y}$) com a resposta real que esperávamos obter ($y$).

A Média dos Erros Quadráticos (MSE — *Mean Squared Error*) é a métrica padrão para problemas de regressão contínua. Ela calcula a punição rigorosa para cada tentativa da rede ao longo de $N$ amostras:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{1}$$

Ao elevar a diferença ao quadrado em vez de usar um valor absoluto, a função assegura dois efeitos colaterais vitais para a otimização de matrizes:

1. **Anulação de Sinal:** Impede que erros negativos anulem erros positivos, lidando puramente com magnitudes e distâncias euclidianas reais.
2. **Severidade Exponencial:** Penaliza severamente erros longínquos (se o erro for 10, a punição será 100; se o erro for 100, a punição será 10.000). A rede passa a priorizar o conserto imediato de amostras onde a resposta foi catastrófica.

## 1.2 Exemplo Numérico: MSE com um Batch

Considere que a MLP do Lab 02 recebeu um batch de 3 candidatos ao emprego na *DeepTech*:

| **Candidato** | $y_{real}$ | $\hat{y}_{predito}$ | $(y - \hat{y})^2$ |
|---|---|---|---|
| Ana | 1.0 | 0.7 | $(0.3)^2 = 0.09$ |
| Bruno | 0.0 | 0.4 | $(-0.4)^2 = 0.16$ |
| Carlos | 1.0 | 0.9 | $(0.1)^2 = 0.01$ |

$$\text{MSE} = \frac{0.09 + 0.16 + 0.01}{3} = \frac{0.26}{3} \approx 0.0867 \tag{2}$$

Observe que o candidato Bruno (erro de 0.4) contribui **proporcionalmente mais** para a perda total do que Carlos (erro de 0.1). Isso é intencional: o quadrado amplifica erros grandes, forçando a rede a focar neles.

## 1.3 Por que o Quadrado e Não o Valor Absoluto?

Uma pergunta natural é: por que não usamos simplesmente $|y - \hat{y}|$? A resposta está na **diferenciabilidade**. A função $|e|$ possui uma "quina" no ponto $e = 0$ — sua derivada não existe nesse ponto. Já a função $e^2$ é uma parábola

suave, diferenciável em todo o domínio. Como veremos na próxima seção, o Gradient Descent **depende** da existência de derivadas em todos os pontos. Sem derivada, não há direção; sem direção, não há aprendizado.


$|e|$**: não-diferenciável em** $e = 0$ $e^2$**: suave e diferenciável** ✓

Figura 1: Comparação entre Erro Absoluto (esquerda) e Erro Quadrático (direita). O quadrado é matematicamente "liso", permitindo que o Gradient Descent funcione corretamente.

**▶ Prática: MSE em NumPy**

A implementação da MSE em uma única linha vetorizada é: `mse = np.mean((y_real - y_predito) ** 2)`. Observe como o NumPy vetoriza a subtração, a potência e a média sem nenhum laço `for`.

# 2 Derivadas e Gradientes: A Bússola da Otimização

Antes de mergulharmos no algoritmo da Descida do Gradiente, é essencial resgatar a intuição geométrica que vocês construíram em **Cálculo I e Cálculo III**. Naquelas disciplinas, a matemática podia parecer abstrata, mas aqui ela se torna a verdadeira bússola que guia o aprendizado de máquina.

**◇ Informação: Resgatando Cálculo I e III: Direção e Intensidade**

- **Cálculo I (A Derivada** $f'(x)$**):** Mede a taxa de variação instantânea (a inclinação da reta tangente). Em IA, a derivada responde a uma pergunta vital: *"Se eu aumentar este peso* $w$ *em uma fração minúscula, o erro da rede vai subir ou descer?"*
- **Cálculo III (O Vetor Gradiente** $\nabla\mathcal{L}$**):** Em redes com múltiplos pesos, calculamos a **derivada parcial** para cada peso, isolando a sua "culpa"no erro total. O vetor gradiente reúne todas essas derivadas parciais.

Para materializar essa teoria, considere uma função de perda hipotética (e bastante simplificada) com apenas dois pesos:

$$\mathcal{L}(w_1, w_2) = w_1^2 + w_2^2 \tag{3}$$

Para descobrir a influência de cada parâmetro no erro, calculamos suas derivadas parciais (derivando em relação a um peso e tratando o outro como constante):

$$\frac{\partial \mathcal{L}}{\partial w_1} = 2w_1 \qquad \frac{\partial \mathcal{L}}{\partial w_2} = 2w_2 \tag{4}$$

O **vetor gradiente** $\nabla \mathcal{L}$ é simplesmente a aglutinação dessas derivadas parciais:

$$\nabla \mathcal{L} = [2w_1, 2w_2]^T \tag{5}$$

O *Teorema da Descida Mais Íngreme* do Cálculo III prova que este vetor aponta sempre para a direção de **maior crescimento** da função. Como o nosso objetivo em IA é **minimizar** o erro o mais rápido possível, a estratégia é obvia: devemos caminhar na direção **oposta** ($-\nabla \mathcal{L}$). É exatamente esse movimento reverso que batiza o famoso algoritmo *Gradient Descent* (Descida do Gradiente).

# 3 A Intuição da Descida do Gradiente

Se visualizarmos a Função de Perda de uma rede neural em relação a todos os seus parâmetros de peso livre $\mathbf{W}$, veremos uma superfície hiperdimensional. Essa topografia é um relevo montanhoso e irregular. O objetivo da inteligência artificial é alcançar a altitude mais baixa possível (o vale absoluto do erro mínimo).

**• Teoria: Metáfora I: O Alpinista Cego na Neblina Densa da Mantiqueira**

Imagine que você está no topo do Pico da Bandeira e uma neblina espessa zera totalmente a sua visibilidade. Seu objetivo é descer até o acampamento no vale (o mínimo global da função de perda). Sem GPS, bússola ou visão panorâmica do relevo completo da montanha, o que você faz?
Com a sola da bota de caminhada, você sente a inclinação do terreno sob seus pés no exato ponto em que está pisando naquele milissegundo. Se o chão sob o pé esquerdo está mais alto que sob o pé direito, você dá um passo para a direita. Repetindo esse ajuste local passo a passo, você desce a montanha com segurança sem jamais ter enxergado a montanha inteira!
É **exatamente assim** que a rede neural otimiza milhões de parâmetros: avaliando o gradiente local sob seus pés a cada iteração, sem precisar conhecer a equação global do terreno.

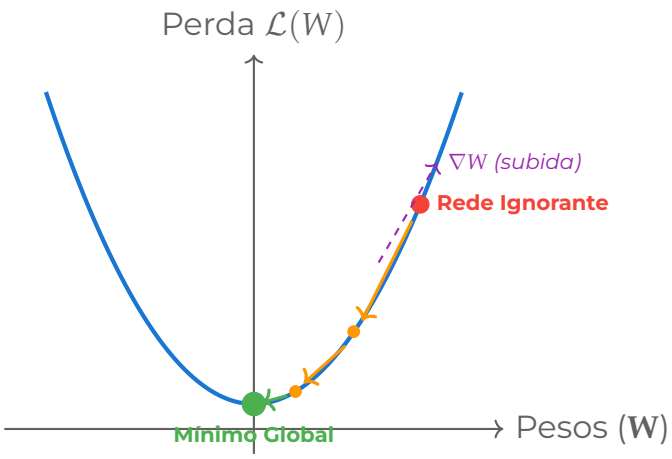



Figura 2: Ilustração vetorial da Descida do Gradiente percorrendo a topografia da função de perda rumo ao erro zero. O vetor gradiente (roxo) aponta para cima; subtraímos para descer.

**★ Checkpoint do Projeto: Checkpoint Geométrico: Curvas de Nível**

Em Cálculo III, vocês aprenderam a representar superfícies tridimensionais usando *curvas de nível* (mapas topográficos com curvas equidistantes de altitude). No Gradient Descent, o vetor gradiente $\nabla \mathcal{L}$ é sempre **perpendicular** às curvas de nível de erro constante. O algoritmo caminha ortogonalmente a essas curvas em direção ao fundo da bacia.

## 4 A Atualização dos Pesos (Update Rule)

O coração do aprendizado não está na propagação, mas na **Atualização dos Pesos** (Update Rule). Após calcularmos o gradiente $\nabla W$ (o quão rápido o erro muda se alterarmos levemente o peso), executamos a alteração real nas matrizes:

$$W_{novo} = W_{atual} - \alpha \cdot \nabla W \tag{6}$$

Onde $\alpha$ é o hiperparâmetro mais vital de toda a Ciência de Dados: o **Learning Rate** (Taxa de Aprendizado).

**▷ Exemplo: Metáfora II: O Registro do Chuveiro e a Taxa de Aprendizado ($\alpha$)**

Ajustar a Taxa de Aprendizado ($\alpha$) no treinamento de IA é rigorosamente idêntico a calibrar o registro de um chuveiro antigo num dia frio de inverno:

- $\alpha$ **muito pequeno (Passo de Formiga):** Você gira o registro apenas $0,1$ mm por vez. Leva 20 minutos congelando no banho até a água esquentar um pouquinho. O treino da rede é absurdamente lento e custoso.
- $\alpha$ **muito grande (Passo Violento):** Você gira o registro $180°$ de uma vez só. A água passa de congelante para fervendo, queimando suas costas! Você dá um pulo e gira $180°$ de volta, fazendo a água trincar de gelada. Em IA, esse movimento brusco faz os pesos quicarem para fora da curva, gerando o *Exploding Gradient* e estourando a memória em `NaN`.
- $\alpha$ **ideal ($\alpha \approx 0.001 \ldots 0.01$):** Um giro firme e calibrado: rápido o bastante para esquentar a água em segundos, suave o suficiente para estabilizar sem queimar ninguém.

## 4.1 Passo-a-Passo Numérico

Para consolidar a intuição, vamos calcular manualmente 5 épocas de treinamento usando uma função de perda simplificada $\mathcal{L}(W) = W^2$ (derivada: $\nabla W = 2W$), com $W_0 = 10.0$ e $\alpha = 0.1$:

**▷ Exemplo: Descida do Gradiente Manual**

| **Época** | $W$ | $\mathcal{L} = W^2$ | $\nabla W = 2W$ | $W_{novo} = W - 0.1 \cdot \nabla W$ |
|---|---|---|---|---|
| 0 | 10.0 | 100.0 | 20.0 | $10 - 2.0 = 8.0$ |
| 1 | 8.0 | 64.0 | 16.0 | $8 - 1.6 = 6.4$ |
| 2 | 6.4 | 40.96 | 12.8 | $6.4 - 1.28 = 5.12$ |
| 3 | 5.12 | 26.21 | 10.24 | $5.12 - 1.024 = 4.096$ |
| 4 | 4.096 | 16.78 | 8.192 | $4.096 - 0.819 = 3.277$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| 20 | $\approx 0.12$ | $\approx 0.015$ | $\approx 0.24$ | $\approx 0.10$ |

Observe: a perda caiu de **100.0** para **0.015** em 20 épocas. A rede **aprendeu**!

◇ **Informação: Como ler essa tabela (O Ciclo de Aprendizado)?**

Cada linha representa uma **época** (1 ciclo de atualização). Lendo a **Época 0** da esquerda para a direita:

- $W$ **(Chute inicial):** A máquina chutou um peso inicial $W = 10.0$.
- $\mathcal{L}$ **(Erro):** Com esse peso, o erro foi altíssimo $(100.0)$. Estamos no topo da montanha.
- $\nabla W$ **(Inclinação):** O gradiente deu $20.0$, indicando que a montanha é muito íngreme neste ponto.
- $W_{novo}$ **(Atualização):** A máquina aplica a fórmula: pega o peso atual $(10.0)$ e subtrai uma fração do gradiente $(0.1 \times 20.0)$. O novo peso ajustado passa a ser $8.0$.

Na **Época 1**, o processo recomeça usando o $W$ atualizado de $8.0$. O erro despenca para $64.0$. Ao repetir esse ciclo linha por linha, a máquina "escorrega" pelo gradiente até o erro ser esmagado.

Note um padrão importante: à medida que $W$ se aproxima de zero, o gradiente $(2W)$ também diminui. Isso significa que os passos ficam naturalmente menores conforme nos aproximamos do mínimo — o alpinista desacelera quando chega ao vale. Esse comportamento é uma propriedade desejável da otimização.

# 5 A Taxa de Aprendizado e os Fenômenos de Divergência

O *Learning Rate* regula diretamente o **tamanho do passo** do nosso alpinista vendado:

- $\alpha$ **muito pequeno:** O treinamento leva semanas computacionais. O alpinista dá passos do tamanho de uma formiga. A rede eventualmente aprende, mas gasta energia e poder de processamento em nuvem excessivo.
- $\alpha$ **ótimo:** O aprendizado ocorre de forma estável, deslizando agressivamente pelas paredes íngremes e reduzindo a velocidade quando chega no plano.
- $\alpha$ **muito grande:** Isso causa o temível efeito da *divergência*. O alpinista tenta dar um passo tão gigantesco que varre o vale inteiro, sendo atirado para a encosta oposta, numa elevação ainda mais alta. Na próxima iteração, o gradiente será maior, e ele quicará com mais força ainda.

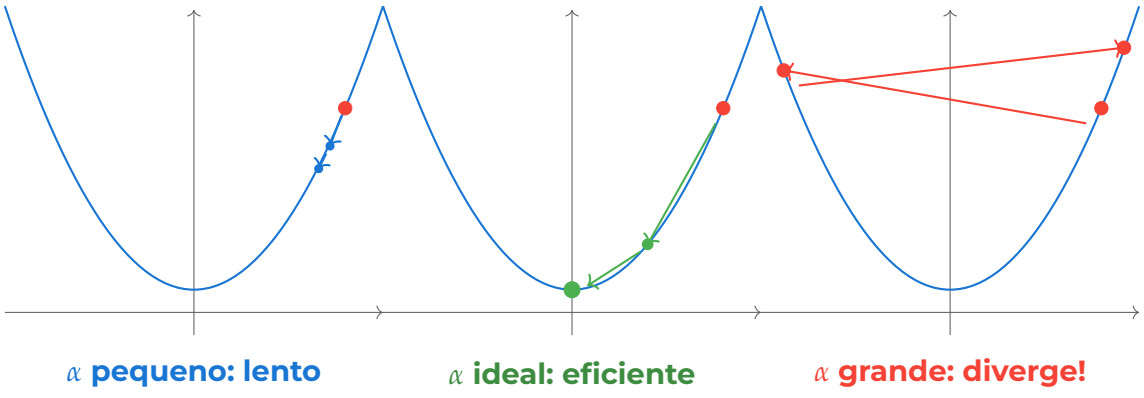



Figura 3: O impacto do Learning Rate na trajetória do Gradient Descent. Um $\alpha$ ideal converge em poucos passos; um $\alpha$ grande demais faz o peso "quicar" para fora do vale.

## 5.1 A Catástrofe do Exploding Gradient

Se o erro cresce a cada iteração porque o *Learning Rate* está desajustado, a matemática exponencial da MSE causa um problema grave no hardware. Vejamos o que acontece com $\alpha = 10.0$:

| **Época** | $W$ | $\mathcal{L} = W^2$ | $W_{novo} = W - 10 \cdot 2W$ |
|---|---|---|---|
| 0 | $10.0$ | $100$ | $10 - 200 = -190$ |
| 1 | $-190$ | $36\,100$ | $-190 + 3\,800 = 3\,610$ |
| 2 | $3\,610$ | $\approx 13 \times 10^6$ | $\rightarrow -68\,590$ |
| 3 | $-68\,590$ | $\approx 4.7 \times 10^9$ | $\rightarrow$ `inf` $\rightarrow$ **NaN** |

O cálculo estoura os limites de arquitetura de 64 bits da memória RAM ou GPU, e as matrizes explodem. O Python interromperá o treinamento e devolverá a famigerada *flag* `NaN` (*Not a Number*). Denominamos este fenômeno de **Exploding Gradient**.

**△ Importante: Regra de Sobrevivência**

Se o seu treino retornar `NaN`: **reduza o Learning Rate**. Uma estratégia segura é começar com $\alpha = 0.001$ e ir subindo gradualmente até encontrar o ponto ótimo. Frameworks modernos como o Keras oferecem *Learning Rate Schedulers* que ajustam $\alpha$ automaticamente durante o treino.

**◇ Informação: Para que servem os Frameworks (Keras, PyTorch, etc)?**

Como acabamos de citar o Keras, vale um parêntese: o objetivo principal de *frameworks* modernos de IA não é apenas prover funções utilitárias como o *Scheduler*. Eles nasceram para **abstrair a complexidade matemática e de infraestrutura**.
Em vez do engenheiro calcular derivadas à mão, alocar matrizes na memória da GPU e escrever *loops* de otimização linha a linha, o *framework*:

- Computa os gradientes de forma 100% automática (recurso conhecido como *Autograd*).
- Gerencia a comunicação transparente entre a CPU e o processamento paralelo da placa de vídeo.
- Implementa defesas matemáticas nativas contra instabilidades numéricas.

Nesta disciplina, estamos abrindo a "caixa-preta" e aprendendo a física por trás do algoritmo. Isso garante que, ao usar o Keras no mercado de trabalho, você não seja um operador cego, mas sim um engenheiro capaz de diagnosticar por que a rede falhou e como consertá-la.

## 5.2 O Irmão Silencioso: Vanishing Gradient

O fenômeno oposto ao Exploding Gradient é o **Vanishing Gradient** (Desvanecimento do Gradiente). É um erro silencioso, porém devastador.

**O Efeito:** Nele, os gradientes tornam-se tão minúsculos (próximos de zero) que a fórmula de atualização ($W_{novo} = W - \alpha \cdot \nabla W$) perde o efeito. Se você tenta atualizar um peso subtraindo um gradiente de $0.00000001$, o peso na prática **não muda**. Como consequência, os pesos das primeiras camadas da rede ficam "congelados". A rede estagna e para de aprender as características básicas dos dados.

▷ **Exemplo: Metáfora IV: O Grito do Erro e a Tubulação**

Para entender esse fenômeno de forma intuitiva, precisamos separar duas coisas: o **Erro da Rede** (o grito) e a **Função de Ativação** (a tubulação). Quando a rede erra feio na última camada, ela dá um "grito": *"Erramos! Voltem e mudem os pesos!"* (esse é o gradiente do Erro). Para que esse grito viaje de ré até a primeira camada, ele é obrigado a passar por dentro da *derivada* da Função de Ativação.

**O Problema da Sigmoid (O Tubo Enferrujado):** A matemática diz que a derivada da Sigmoid é no máximo **0.25**. Isso significa que nada que passa por ela sai com mais de 25% do tamanho. Se a rede grita um Erro 100, ao passar pela primeira Sigmoid ele vira 25; na segunda vira 6.25; na terceira 1.5. Quando o grito chega na primeira camada, ele é apenas um sussurro de **0.001**. A camada não escuta o erro, acha que está tudo bem, e a rede congela.

**A Prova Matemática do Colapso da Sigmoid**

O culpado histórico desse problema é a função de ativação **Sigmoid**. Se analisarmos sua derivada:

$$\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z)) \tag{7}$$

Como vimos na metáfora, o **valor máximo absoluto** que essa derivada atinge é **0.25** (ou $\frac{1}{4}$). No algoritmo de Backpropagation, nós precisamos **multiplicar** as derivadas em cadeia. Veja o que acontece com o sinal do erro sendo multiplicado pelas derivadas de 5 camadas Sigmoids:

$$0.25 \times 0.25 \times 0.25 \times 0.25 \times 0.25 = 0.25^5 \approx 0.001 \tag{8}$$

O sinal do erro chega à primeira camada **1.000 vezes menor** do que saiu! O gradiente literalmente "evapora" pelo caminho — multiplicando frações repetidas vezes, a informação vira pó. É por isso que redes profundas fracassavam antes de 2012.

**• Teoria: A Solução Definitiva: A Tubulação Perfeita da ReLU**

Se o problema da Sigmoid é que sua derivada máxima é uma fração ($0.25$), a solução seria usar uma função cuja derivada não encolha os números.
A **ReLU** ($f(z) = \max(0, z)$) é o nosso "tubo desimpedido". Para qualquer valor positivo, o gráfico da ReLU é uma reta diagonal ($y = x$), e a derivada (inclinação) dessa reta é exatamente **1.0**, o tempo todo!
O que acontece no Backpropagation? Em vez de multiplicar o sinal do erro por $0.25$ a cada camada, nós o multiplicamos por $1$:

$$1.0 \times 1.0 \times 1.0 \times 1.0 \times 1.0 = 1.0 \tag{9}$$

Quando você multiplica um número por $1$, ele **não diminui**. Se a rede grita um Erro 100, ele é multiplicado por $1$ repetidas vezes e viaja intacto até a primeira camada da rede profunda. O gradiente não atrofia mais!

# 6 Backpropagation: A Regra da Cadeia em Ação

Nas seções anteriores, afirmamos que o computador "calcula o gradiente" de cada peso. Mas como exatamente ele computa a derivada parcial do erro em relação a um peso profundo, localizado em uma camada intermediária da rede? A resposta é o algoritmo de **Backpropagation** (Retropropagação do Erro), formalizado por Rumelhart, Hinton e Williams em 1986.

**• Teoria: Metáfora III: A Regra da Cadeia e as Engrenagens do Relógio**

Imagine o mecanismo de um relógio analógico com engrenagens conectadas em série:

$$\text{Engrenagem de Entrada } (W_1) \longrightarrow \text{Engrenagem Intermediária } (A_1) \longrightarrow \text{Ponteiro de Saída } (\hat{y})$$

Se o ponteiro de saída atrasou 5 minutos em relação ao horário correto ($y$), como descobrimos o quanto a primeira engrenagem contribuiu para o erro? A **Regra da Cadeia** do Cálculo I/II calcula exatamente essa transmissão de movimento: multiplicamos a variação do ponteiro pela razão de transmissão da engrenagem intermediária e pela razão da engrenagem inicial. O Backpropagation é esse cálculo de engrenagens feito de trás para frente, atribuindo a cada peso sua parcela exata de responsabilidade pelo erro final.

A ideia é aplicar a **Regra da Cadeia** do cálculo diferencial de trás para frente (da saída para a entrada), decompondo a derivada composta em uma série de derivadas simples:

$$\frac{\partial \mathcal{L}}{\partial W_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial A_1} \cdot \frac{\partial A_1}{\partial Z_1} \cdot \frac{\partial Z_1}{\partial W_1} \tag{10}$$

Cada fator dessa cadeia é uma derivada local que pode ser calculada de forma trivial:

- $\frac{\partial \mathcal{L}}{\partial \hat{y}}$: Derivada da função de perda (ex: $2(\hat{y} - y)$ para MSE).
- $\frac{\partial \hat{y}}{\partial Z}$: Derivada da função de ativação (ex: $\sigma(z)(1 - \sigma(z))$ para Sigmoid; 0 ou 1 para ReLU).
- $\frac{\partial Z}{\partial W}$: Derivada da soma ponderada em relação ao peso (é simplesmente a entrada $X$).

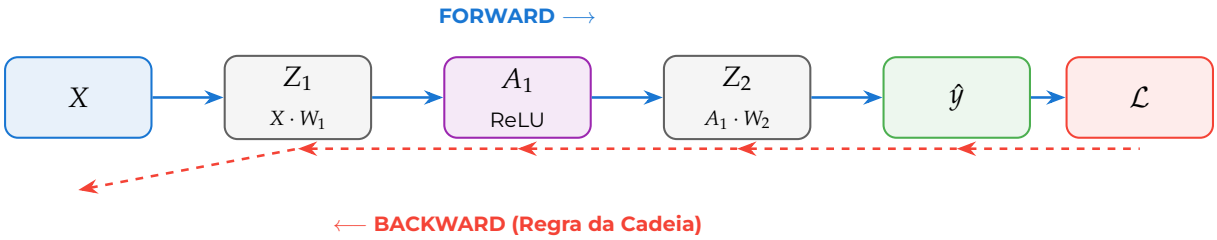



Figura 4: O Backpropagation percorre o grafo computacional de trás para frente, propagando os "erros locais" de cada nó via Regra da Cadeia.

## 6.1 Exemplo Numérico Completo

Para desmistificar o Backpropagation, vamos executar um passo completo (forward + backward) em um neurônio simples com **ReLU**: $\hat{y} = \max(0, w \cdot x + b)$, com $\mathcal{L} = (\hat{y} - y)^2$.

**Dados iniciais:** $x = 2.0$, $w = 0.5$, $b = 0.1$, $y_{real} = 2.5$, $\alpha = 0.1$.

**▷ Exemplo: Forward + Backward Completo (com ReLU)**

| Passo | Cálculo | Valor |
|---|---|---|
| **— Forward Pass —** | | |
| $z$ | $w \cdot x + b = 0.5 \times 2.0 + 0.1$ | 1.1 |
| $\hat{y}$ | $\text{ReLU}(1.1) = \max(0, 1.1)$ | 1.1 |
| $\mathcal{L}$ | $(\hat{y} - y)^2 = (1.1 - 2.5)^2 = (-1.4)^2$ | 1.96 |
| **— Backward Pass (Regra da Cadeia) —** | | |
| $\frac{\partial \mathcal{L}}{\partial \hat{y}}$ | $2(\hat{y} - y) = 2(1.1 - 2.5)$ | $-2.8$ |
| $\frac{\partial \hat{y}}{\partial z}$ | Derivada da ReLU para $z > 0$ | 1.0 |
| $\frac{\partial z}{\partial w}$ | $x$ | 2.0 |
| **— Gradiente Final —** | | |
| $\frac{\partial \mathcal{L}}{\partial w}$ | $(-2.8) \times 1.0 \times 2.0$ | $\mathbf{-5.6}$ |
| **— Update Rule —** | | |
| $w_{novo}$ | $0.5 - 0.1 \times (-5.6)$ | **1.06** |

O peso subiu de 0.5 para 1.06. Isso fará $z$ subir na próxima época (se aproximando de $y = 2.5$), reduzindo o erro. O mecanismo funciona e, diferentemente da Sigmoid, o sinal do erro ($-2.8$) passou intacto (multiplicado por 1.0) pela função de ativação!

**◇ Informação: Como ler o Exemplo Completo passo a passo?**

A tabela ilustra o fluxo de dados em um neurônio. Ela é dividida em quatro blocos lógicos:

**1. Forward Pass (A Ida):** A rede recebe o dado e faz a previsão.

- $z$: É a soma ponderada. O neurônio pega o dado ($x = 2.0$), multiplica pelo seu peso atual ($w = 0.5$) e soma o viés ($b = 0.1$). O resultado bruto é $1.1$.
- $\hat{y}$ (y-chapéu): É a previsão final. Passando $1.1$ pela ReLU (que preserva os positivos), a saída continua $1.1$.
- $\mathcal{L}$ (Loss): É o tamanho do erro. A resposta correta ($y$) era $2.5$. O Erro Quadrático resultou em $1.96$.

**2. Backward Pass (A Volta):** Calculando a culpa (as derivadas parciais).

- $\frac{\partial \mathcal{L}}{\partial \hat{y}}$: É o erro da previsão. O sinal negativo ($-2.8$) indica à rede: *"previmos para baixo, precisamos aumentar a previsão"*.
- $\frac{\partial \hat{y}}{\partial z}$: É o "tubo" da ReLU. Como o valor de $z$ era positivo, a porta estava totalmente aberta (derivada exata igual a $1.0$).
- $\frac{\partial z}{\partial w}$: É o peso que o próprio dado de entrada teve na conta. Como a entrada $x$ era $2.0$, o valor repassado é $2.0$.

**3. Gradiente Final (Regra da Cadeia):** Para descobrir a culpa exata do peso $w$, basta multiplicar as três partes da volta em cadeia: $(-2.8) \times 1.0 \times 2.0 = \mathbf{-5.6}$. Esse número é o **gradiente** do peso $w$.

**4. Update Rule (A Atualização):** Com o gradiente em mãos, corrigimos o peso antigo usando o *Learning Rate* ($\alpha = 0.1$). A regra manda *subtrair* a culpa: $0.5 - (0.1 \times -5.6)$. Como menos com menos dá mais, o peso é fortemente empurrado para cima (para $1.06$).

**• Teoria: A Essência do Backpropagation**

O Backpropagation não é um algoritmo de aprendizado em si — é um algoritmo de **computação eficiente de gradientes**. Ele reutiliza os cálculos intermediários (os "erros locais" de cada camada) para evitar recomputações redundantes. Sem o Backpropagation, calcular o gradiente de cada peso individualmente teria custo $O(n^2)$ por peso. Com ele, o custo total é apenas $O(n)$ — proporcional ao Forward Pass. Vocês já viram a Regra da Cadeia em Cálculo II — aqui ela é aplicada repetidamente ao longo do grafo computacional da rede.

Na prática, frameworks como TensorFlow e PyTorch implementam o Backpropagation automaticamente através de um mecanismo chamado **Auto-**

**grad** (Diferenciação Automática). O engenheiro de IA moderno raramente precisa calcular derivadas manualmente, mas compreender o mecanismo é fundamental para diagnosticar problemas de treinamento.

## 7 Batch, Mini-Batch e SGD Estocástico

Até agora, implicitamente assumimos que o gradiente é calculado usando **todas** as amostras do dataset de uma vez. Na prática, isso é raramente viável para datasets grandes. A indústria adota três variantes:

- **Batch Gradient Descent:** Calcula o gradiente com todas as $N$ amostras. Convergência suave, mas lento e exigente em memória.
- **Stochastic Gradient Descent (SGD):** Calcula o gradiente com **uma única** amostra por vez. Rápido, mas ruidoso e instável.
- **Mini-Batch Gradient Descent:** O padrão da indústria. Calcula o gradiente com um subconjunto (ex: 32, 64 ou 128 amostras). Combina a estabilidade do Batch com a velocidade do SGD.

O hiperparâmetro `batch_size` no Keras controla exatamente essa escolha:

```
model.fit(X, y, epochs=50, batch_size=32)
# A cada iteracao, 32 amostras sao processadas
# e o gradiente e calculado sobre esse mini-lote
```

## 8 Otimizadores Modernos: Além do SGD

O SGD básico tem limitações conhecidas: oscila em vales estreitos, pode ficar preso em mínimos locais, e exige calibração cuidadosa do Learning Rate. A pesquisa em otimização produziu variantes mais inteligentes:

- **SGD com Momentum:** Acumula "velocidade" ao longo das épocas, como uma bola rolando montanha abaixo. Isso permite que o otimizador "ganhe inércia" e atravesse regiões planas sem parar.
- **Adam (*Adaptive Moment Estimation*):** Combina a ideia de momentum com um **Learning Rate adaptativo por peso**. Cada peso recebe um $\alpha$ diferente, ajustado automaticamente com base no histórico de gradientes. É o otimizador padrão da indústria desde 2015.

**▶ Prática: Adam no Keras**

`model.compile(optimizer='adam', loss='mse')` — esta única linha configura o otimizador Adam com valores padrão ($\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$) que funcionam bem para aproximadamente 90% dos problemas.

# 9 Guia de Diagnóstico do Treinamento

A **Curva de Aprendizado** (gráfico de Loss × Épocas) é o "eletrocardiograma" da rede neural. Saber interpretá-la é uma habilidade essencial do engenheiro de IA:

| Sintoma | Diagnóstico | Solução |
|---|---|---|
| Loss = `NaN` | Exploding Gradient | Reduzir $\alpha$ (ex: $0.001$) |
| Loss não desce | Vanishing Gradient ou $\alpha$ minúsculo | Usar ReLU, aumentar $\alpha$ |
| Loss oscila sem convergir | $\alpha$ grande demais | Reduzir $\alpha$ por fator de 10 |
| Loss converge muito devagar | $\alpha$ pequeno demais | Aumentar $\alpha$ ou usar Adam |
| Loss cai e depois sobe | Overfitting | Mais dados, regularização |

# 10 O Loop Completo de Treinamento

O treinamento de uma rede profunda é, portanto, um laço de repetição (*for loop*) computado sobre **Épocas**. Em cada época de treinamento o computador realiza as 4 etapas fundamentais:

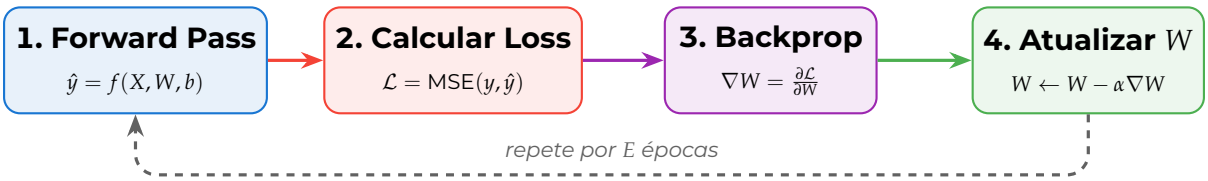



Figura 5: As 4 etapas fundamentais de cada época de treinamento. O ciclo se repete centenas ou milhares de vezes até que a perda convirja para um valor aceitável.

Dominar a taxa de aprendizado e mitigar a explosão de gradientes permite que se treine desde sistemas triviais (como regressores de dados tabulares) até Modelos de Linguagem Giga-parametrizados (LLMs) como os que dominam o mercado hoje.

### ✔ Takeaways

- A **Função de Perda (MSE)** transforma o erro da rede em um número que o computador pode minimizar. O quadrado garante diferenciabilidade.
- O **Gradiente** ($\nabla W$) é o vetor de derivadas parciais que aponta para a maior subida; subtraímos para descer (Cálculo III).
- A **Update Rule** ($W_{novo} = W - \alpha \nabla W$) é o coração de todo treinamento de IA.
- O **Learning Rate** ($\alpha$) controla o tamanho do passo — o ajuste do registro do chuveiro (muito grande causa *Exploding Gradient* e NaN, muito pequeno trava o aprendizado).
- O **Backpropagation** aplica a Regra da Cadeia de trás para frente como engrenagens de um relógio, calculando gradientes de forma eficiente ($O(n)$) ao longo do grafo computacional.
- **SGD e Mini-Batch:** Em vez de calcular o gradiente usando o *dataset* inteiro de uma vez (pesado), a indústria usa o *Mini-Batch* (blocos de 32 ou 64 amostras), equilibrando velocidade e estabilidade.
- O **Adam** é o otimizador padrão da indústria, combinando momentum e Learning Rate adaptativo por peso.

## 11 Conclusão

Neste capítulo, quebramos a barreira entre uma rede neural que apenas "chuta" e uma rede que efetivamente **aprende**. A Função de Perda MSE quantifica o erro; as derivadas parciais medem a "culpa" de cada peso; a Descida do Gradiente indica a direção de correção; e a Update Rule ajusta os pesos iterativamente. O Backpropagation torna esse processo computacionalmente viável mesmo para redes profundas, enquanto o Vanishing Gradient explica por que a ReLU é tão importante. Na prática, otimizadores como Adam automatizam boa parte dessas decisões. No próximo capítulo, veremos como o **Keras** e o **TensorFlow** encapsulam todo esse pipeline em poucas linhas de código.

## ✎ Questões: Autoavaliação

1 **[Direta]** Explique por que a MSE eleva o erro ao quadrado em vez de usar o valor absoluto. Quais são as duas vantagens matemáticas dessa escolha?

2 **[Direta]** Dado $W_0 = 5.0$, $\mathcal{L}(W) = W^2$ e $\alpha = 0.2$, calcule o valor de $W$ e da perda $\mathcal{L}$ após 3 épocas de Gradient Descent.

3 **[Direta]** Explique o fenômeno do Vanishing Gradient e por que a ReLU resolve esse problema melhor que a Sigmoid.

4 **[Reflexiva]** Se o Gradient Descent sempre segue a direção de maior descida *local* (como o alpinista na neblina), é possível que ele fique "preso" em um vale que não é o mínimo global? Como a dimensão hiperdimensional em redes profundas afeta esse problema?

5 **[Reflexiva]** Frameworks como Keras e PyTorch automatizam completamente o Backpropagation (via Autograd). Em que cenário um engenheiro de IA que *não compreende* a Regra da Cadeia e a física da taxa de aprendizado pode tomar decisões desastrosas de projeto?

# Lab. 03: Descida do Gradiente

## ⋆ Objetivo do Laboratório

Nos Labs 01 e 02, seus neurônios propagavam dados, mas os pesos eram fixos ou aleatórios — a rede "chutava". Hoje você constrói o mecanismo que faz a rede **aprender**: o laço de treinamento por *Épocas*. Você implementará a Função de Perda (MSE), o loop de Gradient Descent com a Update Rule, visualizará a curva de aprendizado, e provocará deliberadamente um *Exploding Gradient* para entender o risco de uma taxa de aprendizado mal calibrada.

# 12 Parte 1: O Desafio do Alpinista Vendado

Na aula teórica, vimos a analogia do **Alpinista Vendado**: ele está perdido numa montanha (a superfície de erro) e quer chegar ao vale (o ponto de menor erro). A cada passo, ele tateia o chão ao redor (calcula o gradiente) e dá um passo na direção de descida. O tamanho do passo é controlado pela **Taxa de Aprendizado** ($\alpha$).

## Problema C: Calibrando o Alpinista

**Contexto:** Você é o engenheiro responsável por calibrar o sistema de navegação do Alpinista. A superfície de erro é uma parábola simples $\mathcal{L}(w) = w^2$ (cujo mínimo global é $w = 0$). Sua missão é implementar o algoritmo de otimização e observar como o comportamento muda conforme a taxa de aprendizado varia.

**Modelo de Entrada:**

- $w_0 = 10.0$ (posição inicial do alpinista — longe do vale)
- $\alpha$ (taxa de aprendizado — o tamanho do passo)
- $E$ (número de épocas — quantos passos o alpinista dará)

**Modelo de Saída:**

- Vetor de erros $[\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_{E-1}]$ ao longo das épocas
- Gráfico da curva de aprendizado (MSE $\times$ Épocas)

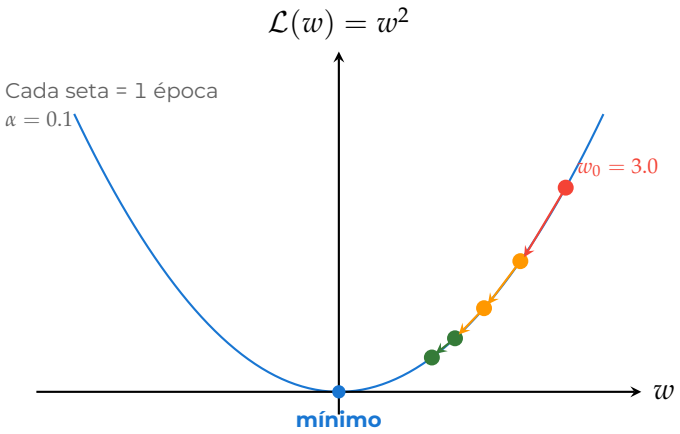
### A. A Matemática por Trás

Para a função de perda $\mathcal{L}(w) = w^2$, a derivada (gradiente) é conhecida analiticamente:

$$\nabla_w \mathcal{L} = \frac{d}{dw}(w^2) = 2w$$

A cada época, o alpinista executa a **Update Rule**:

$$w_{\text{novo}} = w - \alpha \cdot \nabla_w \mathcal{L} = w - \alpha \cdot 2w$$




# 13 Parte 2: Implementando a Função de Perda (MSE)

Antes de treinar, precisamos de uma função que meça o erro. Crie o arquivo `src/gradiente.py` no seu repositório e implemente a MSE:

```python
import numpy as np

def calcular_mse(y_real, y_predito):
    """Calcula a Media dos Erros Quadraticos (MSE).
    Quanto maior o valor, pior a rede esta performando."""
    return np.mean((y_real - y_predito) ** 2)
```

**• Teoria: Verificação Rápida**

Teste mentalmente: se $y_{real} = [3.0, 5.0]$ e $y_{pred} = [1.0, 5.0]$, então:

$$\text{MSE} = \frac{(3-1)^2 + (5-5)^2}{2} = \frac{4+0}{2} = 2.0$$

Se a sua função não retornar 2.0 para esse caso, algo está errado.

# 14 Parte 3: O Loop de Treinamento

Agora vamos construir o coração do aprendizado: o laço de épocas com a Update Rule. Para simplificação didática (a derivação completa do Backpropagation multicamadas virá nas próximas aulas), usaremos a função de perda analítica $\mathcal{L}(w) = w^2$ com derivada conhecida $\nabla w = 2w$.

Adicione ao `src/gradiente.py`:

```python
def treinar(learning_rate, epochs):
    """Executa a Descida do Gradiente em uma parabola simples.
    Retorna o historico de erros para analise."""
    peso = 10.0  # Posicao inicial (longe do minimo)
    historico_erros = []

    for epoch in range(epochs):
        # 1. Calcular a perda: L(w) = w^2
        erro = peso ** 2
        historico_erros.append(erro)

        # 2. Calcular o gradiente: dL/dw = 2w
        gradiente = 2 * peso

        # 3. UPDATE RULE: w_novo = w - alpha * gradiente
        peso = peso - (learning_rate * gradiente)

    return historico_erros
```

**△ Importante: Entendendo a Linha Crucial**

O segredo está na linha `peso = peso - (learning_rate * gradiente)`. Esta é a **Update Rule** $w_{novo} = w - \alpha \cdot \nabla w$ traduzida em Python. Se $\alpha$ for grande demais, o peso "pula" para o outro lado da parábola — é o início do *Exploding Gradient*.

## Tabela: Primeiras 5 Épocas ($\alpha = 0.1$, $w_0 = 10.0$)

Acompanhe na mão as primeiras épocas para confirmar que seu código está correto:

| Época | $w$ | $\mathcal{L} = w^2$ | $\nabla = 2w$ | $w_{novo} = w - 0.1 \cdot \nabla$ |
|---|---|---|---|---|
| 0 | 10.0 | 100.0 | 20.0 | $10.0 - 2.0 = 8.0$ |
| 1 | 8.0 | 64.0 | 16.0 | $8.0 - 1.6 = 6.4$ |
| 2 | 6.4 | 40.96 | 12.8 | $6.4 - 1.28 = 5.12$ |
| 3 | 5.12 | 26.21 | 10.24 | $5.12 - 1.024 = 4.096$ |
| 4 | 4.096 | 16.78 | 8.192 | $4.096 - 0.819 = 3.277$ |

Observe como o erro cai consistentemente a cada época — é o alpinista descendo a montanha com passos seguros!

# 15 Parte 4: Visualizando a Curva de Aprendizado

Um gráfico vale mais que mil números. Adicione ao `src/gradiente.py` a função de plotagem e o bloco principal:

```python
import matplotlib
matplotlib.use('Agg')  # Backend sem janela (para CI/CD)
import matplotlib.pyplot as plt

def plotar_convergencia(historico, nome_arquivo):
    """Gera o grafico da curva de aprendizado (MSE x Epocas)."""
    plt.figure(figsize=(8, 5))
    plt.plot(historico, color='red', marker='o', markersize=3,
             linewidth=1.5, label='MSE')
    plt.title("Curva de Aprendizado (Gradient Descent)")
    plt.xlabel("Epocas")
    plt.ylabel("MSE (Erro)")
    plt.grid(True, alpha=0.3)
    plt.legend()
    plt.tight_layout()
    plt.savefig(nome_arquivo, dpi=150)
    plt.close()
    print(f"Grafico salvo em: {nome_arquivo}")

if __name__ == "__main__":
    # Treino com LR seguro
    erros = treinar(learning_rate=0.1, epochs=30)
    plotar_convergencia(erros, "grafico_convergencia.png")

    print(f"Erro inicial: {erros[0]:.4f}")
    print(f"Erro final:   {erros[-1]:.4f}")
```

Execute `python src/gradiente.py` e verifique que o gráfico `grafico_convergencia.png` mostra uma curva descendente. O erro deve cair de 100.0 para próximo de zero.

# 16 Parte 5: O Experimento do Exploding Gradient

Agora vem a parte mais instrutiva do laboratório: vamos **quebrar o treinamento de propósito** para entender por que a escolha de $\alpha$ é tão importante.

## A. Provocando a Explosão

No terminal Python interativo (ou em um bloco separado do seu script), execute:

```python
erros_explosivos = treinar(learning_rate=10.0, epochs=5)
print(erros_explosivos)
```

### B. O que Acontece Internamente

Acompanhe passo a passo com $\alpha = 10.0$:

| Época | $w$ | $\mathcal{L} = w^2$ | $\nabla = 2w$ | $w_{novo} = w - 10.0 \cdot \nabla$ |
|---|---|---|---|---|
| 0 | 10.0 | 100 | 20 | $10 - 200 = \mathbf{-190}$ |
| 1 | $-190$ | 36 100 | $-380$ | $-190 + 3800 = \mathbf{3\,610}$ |
| 2 | 3 610 | $\approx 13 \times 10^6$ | 7 220 | $\mathbf{-68\,590}$ |
| 3 | $-68\,590$ | $\approx 4.7 \times 10^9$ | → **Explosão para $\pm\infty$ / NaN** | |

**△ Importante: Lição Fundamental**

Com $\alpha$ grande demais, o peso "quica" de um lado para o outro da parábola com amplitude **crescente**. O erro não desce — ele **explode exponencialmente** até ultrapassar os limites de representação numérica do computador (`inf`), gerando depois `NaN` (*Not a Number*). Na prática, quando você vê "NaN" na saída do TensorFlow, quase sempre é um Learning Rate mal calibrado.

## 17 Parte 6: Testes Automatizados

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

```
import numpy as np
from src.gradiente import treinar, calcular_mse

def test_mse_calculo_correto():
    """Garante que a MSE computa corretamente."""
    y_real = np.array([3.0, 5.0])
    y_pred = np.array([1.0, 5.0])
    resultado = calcular_mse(y_real, y_pred)
    np.testing.assert_almost_equal(resultado, 2.0)

def test_convergencia():
    """Garante que após o treino, o erro despencou."""
    erros = treinar(learning_rate=0.1, epochs=30)
    assert erros[-1] < erros[0], "A rede não aprendeu!"
    assert erros[-1] < 0.1, f"Erro final alto: {erros[-1]}"

def test_sem_nan():
    """Garante que o treino com LR seguro não produz NaN."""
    erros = treinar(learning_rate=0.1, epochs=50)
    for i, e in enumerate(erros):
```

```
        assert not np.isnan(e), f"NaN na época {i}!"
        assert not np.isinf(e), f"Inf na época {i}!"

def test_exploding_gradient_detectado():
    """Confirma que LR absurdo causa divergência."""
    erros = treinar(learning_rate=10.0, epochs=5)
    assert erros[2] > erros[0], "LR=10 deveria divergir!"
```

1 Rode os testes no terminal: `pytest tests/test_gradiente.py -v`.

2 Se a tela exibir **4 passed** em verde, sua implementação está correta.

## 18 Parte 7: Commit e Push

Com todos os testes passando:

```
git add src/gradiente.py
git commit -m "Lab 03: Gradient Descent + Exploding Gradient"
git push origin main
```

O GitLab CI confirmará o Selo Verde automaticamente.

## Conclusão: O que Você Construiu Hoje

### ✔ Takeaways

- Você implementou do zero a **Função de Perda MSE** e o **Loop de Treinamento** com a Update Rule $w_{novo} = w - \alpha \cdot \nabla w$.
- Visualizou a **Curva de Aprendizado** mostrando o erro caindo de 100 para próximo de zero em 30 épocas.
- Provocou deliberadamente um **Exploding Gradient** com $\alpha = 10.0$ e observou os números explodindo para `inf`/`NaN`.
- Criou testes automatizados que validam convergência, ausência de NaN e detecção de divergência.

◇ **Informação**

**Gancho para a Próxima Aula:** Você acabou de implementar manualmente o que o Keras faz em uma única linha: `model.fit(X, y, epochs=30)`. Na próxima aula, vamos migrar toda essa engenharia manual para o ecossistema **Keras/TensorFlow**, onde o Forward Pass, a Loss, o Backpropagation e a Update Rule são executados automaticamente com aceleração em GPU. A pergunta será: *"se o Keras faz tudo sozinho, por que precisei aprender isso na mão?"* — e a resposta ficará clara quando você precisar **debugar** um modelo que não converge.