Lab. 03: Descida do Gradiente e Exploding Gradients

Aléssio Miranda Júnior

Agosto - 2026/2

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.

1 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** (α).

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)
- α (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)

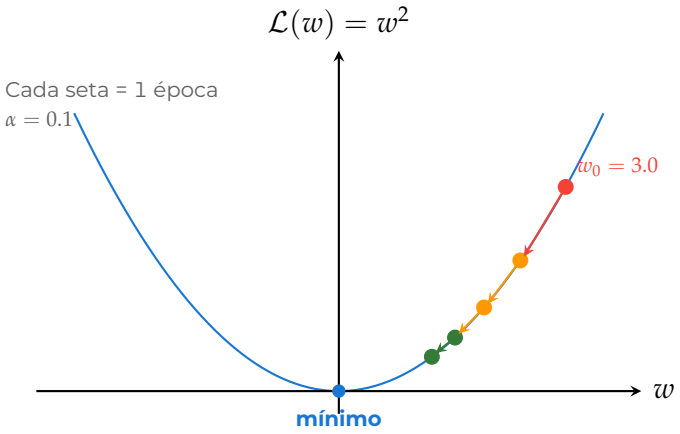
A. A Matemática por Trás

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**:

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



2 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:

```
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.

3 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`:

```
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 α 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!

4 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:

```
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.

5 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 α é tão importante.

A. Provocando a Explosão

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

```
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	$-68\,590$	$\approx 4.7 \times 10^9$	$\rightarrow$ **Explosão para $\pm\infty$ / NaN**	

△ Importante: Lição Fundamental

Com α 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.

6 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.

7 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.