

# T319 - Introdução ao Aprendizado de Máquina: *Regressão Linear (Parte II)*



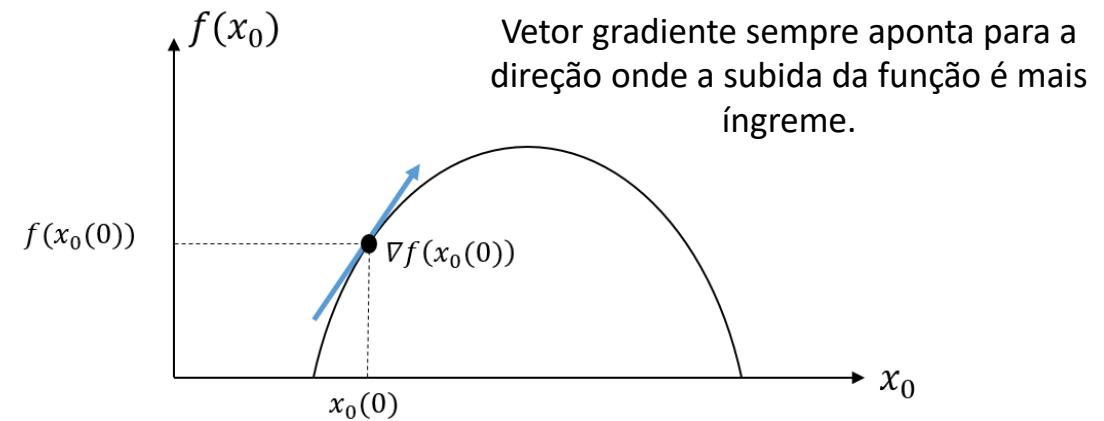
***Inatel***

Felipe Augusto Pereira de Figueiredo  
felipe.figueiredo@inatel.br

# Recapitulando

- Vimos a motivação por trás da regressão: encontrar curvas que nos ajudem a prever valores.
- Definimos o problema matematicamente.
- Vimos como resolver o problema da regressão, i.e., encontrar os pesos do modelo, através da equação normal.
- Aprendemos o que é uma superfície de erro.
- Discutimos algumas desvantagens da equação normal e apresentamos uma solução para essas desvantagens, a qual discutiremos a seguir.

# Vetor Gradiente



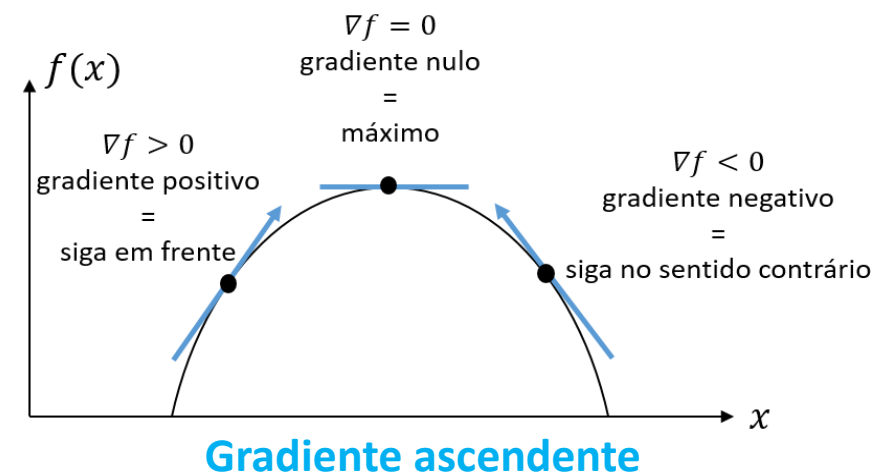
- Vocês se lembram das aulas de cálculo vetorial, onde vocês aprenderam sobre o ***vetor gradiente***?
  - ***Vetor gradiente*** é um vetor que indica a direção e o sentido no qual, por deslocamento a partir de um ponto específico, obtém-se o maior incremento possível no valor de uma função,  $f$ .
- O ***vetor gradiente*** de uma função,  $f(x_0, x_1, \dots, x_K)$ , em relação aos seus argumentos  $x_k, k = 0, \dots, K$ , é definido por

$$\nabla f(x_0, x_1, \dots, x_K) = \left[ \frac{\partial f(x_0, x_1, \dots, x_K)}{\partial x_0} \quad \frac{\partial f(x_0, x_1, \dots, x_K)}{\partial x_1} \quad \dots \quad \frac{\partial f(x_0, x_1, \dots, x_K)}{\partial x_K} \right]^T,$$

onde  $\nabla f(x_0, x_1, \dots, x_K)$  é o vetor que indica a direção e o sentido em que a função,  $f(x_0, x_1, \dots, x_K)$ , tem a taxa de crescimento mais rápida.

- Notem, que cada elemento do ***vetor gradiente*** indica a direção e o sentido de máxima variação em relação àquele argumento da função.
- Se imaginem parados em um ponto  $x_0(0), x_1(0), \dots, x_K(0)$  no domínio de  $f$ , o vetor  $\nabla f(x_0(0), x_1(0), \dots, x_K(0))$  diz em qual direção e sentido devemos caminhar para aumentar o valor de  $f$  mais rapidamente.

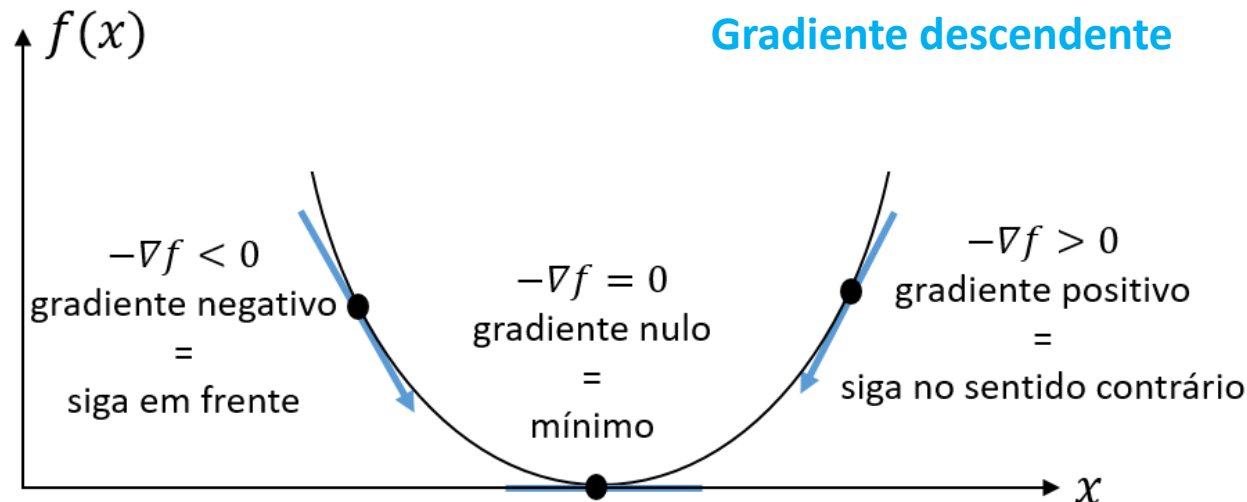
# Gradiente Ascendente



- O valor do ***vetor gradiente*** em um ponto é um ***vetor tangente*** àquele ponto, onde um elemento do vetor com valor:
  - + significa que o ponto de máximo está à frente.
  - - significa que o ponto de máximo está atrás.
  - 0 significa que ponto de máximo foi encontrado.
- Portanto, o ***vetor gradiente*** nos permite encontrar o ponto de ***máximo*** da função,  $f(x_0, x_1, \dots, x_K)$ .
  - Seguindo na direção e sentido indicados pelo ***vetor gradiente***, chegamos ao ponto de máximo da função.
- Assim, um algoritmo de otimização ***iterativo*** que siga a direção e sentido indicados pelo ***vetor gradiente*** para encontrar o ***ponto de máximo*** de  $f(x_0, x_1, \dots, x_K)$  é conhecido como ***gradiente ascendente***.

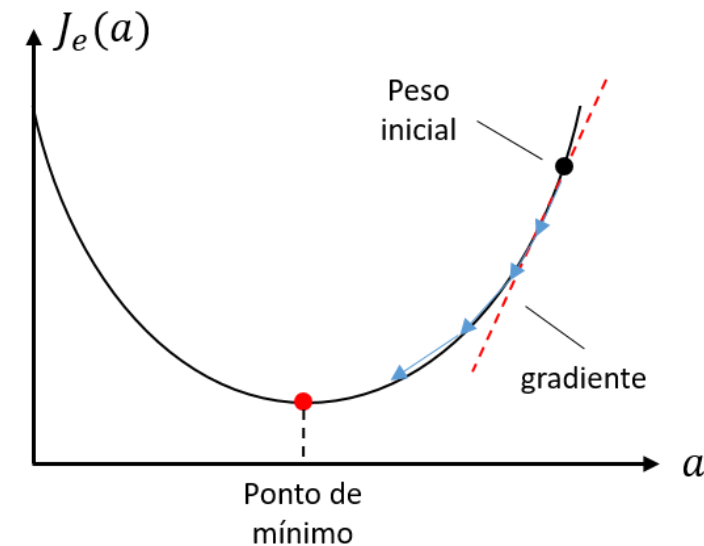
# Gradiente Descendente

- Mas e se formos no sentido contrário a da taxa de crescimento, dada pelo **vetor gradiente**,  $\nabla f(x_0, x_1, \dots, x_K)$ , ou seja  $-\nabla f(x_0, x_1, \dots, x_K)$ ?
  - Nesta caso, iremos na direção de **decréscimo** mais rápido da função,  $f(x_0, x_1, \dots, x_K)$ .
- Portanto, um algoritmo de otimização **iterativo** que siga a direção e sentido contrário ao indicado pelo **vetor gradiente** para encontrar o **ponto de mínimo** de  $f(x_0, x_1, \dots, x_K)$  é conhecido como **gradiente descendente**.



# Gradiente Descendente

- Algoritmo de otimização **iterativo** e **genérico**: encontra soluções ótimas para uma ampla gama de problemas.
- Por exemplo, é utilizado em vários problemas de aprendizado de máquina e otimização.
- Escalona melhor do que o método da **equação normal** para grandes conjuntos de dados.
- É de fácil implementação.
- Não é necessário se preocupar com matrizes mal-condicionadas (determinante próximo de 0, i.e., quase **singulares**).
- Pode ser usado com modelos não-lineares.
- O único requisito é que a **função de erro** seja **diferenciável**.
- Quando aplicado a problemas de **regressão**, a ideia geral é ajustar os pesos,  $a$ , iterativamente, a fim de **minimizar** a **função de erro**, ou seja, encontrar seu **ponto de mínimo**.
- A seguir, veremos como aplicar o algoritmo do **gradiente descendente** ao problema da **regressão linear**.

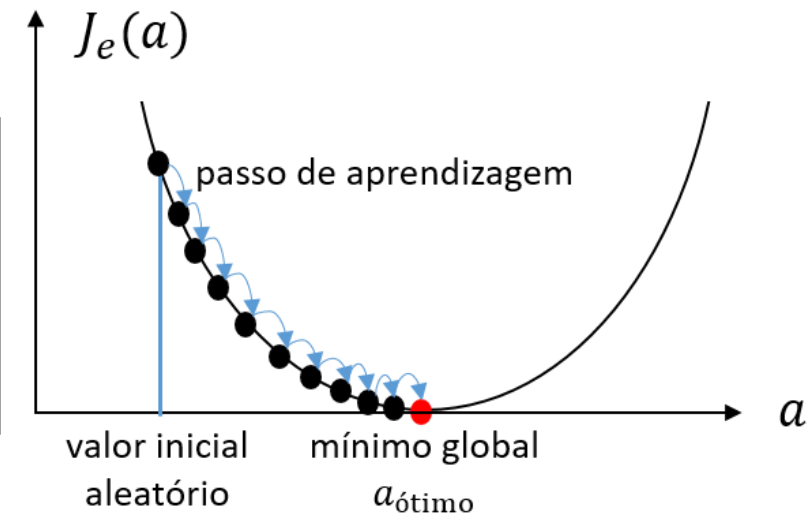


# O Algoritmo do Gradiente do Descendente (GD)

- O algoritmo inicializa os pesos,  $\mathbf{a}$ , em um ponto aleatório do **espaço de pesos** e então, os atualiza no **sentido oposto** ao do **gradiente** até que algum critério de convergência seja atingido, indicando que um **mínimo local** ou o **global** da **função de erro** foi encontrado.

$\mathbf{a} \leftarrow$  inicializa em um ponto qualquer do espaço de pesos  
**loop** até convergir **ou** atingir número máximo de épocas **do**  
    **for each**  $a_i$  **in**  $\mathbf{a}$  **do**

$$a_i \leftarrow a_i - \alpha \frac{\partial J_e(\mathbf{a})}{\partial a_i}$$



onde  $\alpha$  é a **taxa/passo de aprendizagem** e  $\frac{\partial J_e(\mathbf{a})}{\partial a_i}$  é o gradiente da **função de erro** em relação ao peso  $a_i$ .

- A **taxa de aprendizagem** dita o tamanho dos passos/deslocamentos dados na direção e sentido oposto ao do **gradiente**.

# O Algoritmo do Gradiente do Descendente (GD)

- O **passo de aprendizagem**,  $\alpha$ , pode ser constante ou pode decair com o tempo à medida que o processo de aprendizado prossegue.
- **OBS.:** Os pesos,  $\mathbf{a}$ , **devem ser atualizados simultaneamente**, caso contrário o algoritmo apresentará comportamento desconhecido.
- O pseudo-algoritmo abaixo apresenta a atualização simultânea de todos os **pesos**.

$$\begin{aligned} \mathbf{a} &\leftarrow \text{inicializa em um ponto qualquer do espaço de pesos} \\ \text{loop até convergir ou atingir número máximo de épocas do} \\ &\quad \mathbf{a} \leftarrow \mathbf{a} - \alpha \nabla J_e(\mathbf{a}) \end{aligned}$$

onde  $\mathbf{a}$  é o vetor com os **pesos** e  $\nabla J_e(\mathbf{a}) = \left[ \frac{\partial J_e(\mathbf{a})}{\partial a_0} \quad \dots \quad \frac{\partial J_e(\mathbf{a})}{\partial a_K} \right]^T$  é o **vetor gradiente**, o qual contém o gradiente com relação a todos os **pesos**.

- Na sequência, veremos como encontrar o **vetor gradiente** da função de erro e implementar o algoritmo do **gradiente descendente**.



# Exemplo #1

[Exemplo 1: linear regression with gradient descent exemplo1.ipynb](#)

Neste exemplo, usaremos uma **função hipótese** com 2 pesos,  $a_1$  e  $a_2$ , sendo  $a_0 = 0$

$$\hat{y}(n) = h(\mathbf{x}(n)) = a_1 x_1(n) + a_2 x_2(n).$$

A função de erro é dada por

$$J_e(\mathbf{a}) = \frac{1}{N} \sum_{n=0}^{N-1} [y(n) - (a_1 x_1(n) + a_2 x_2(n))]^2.$$

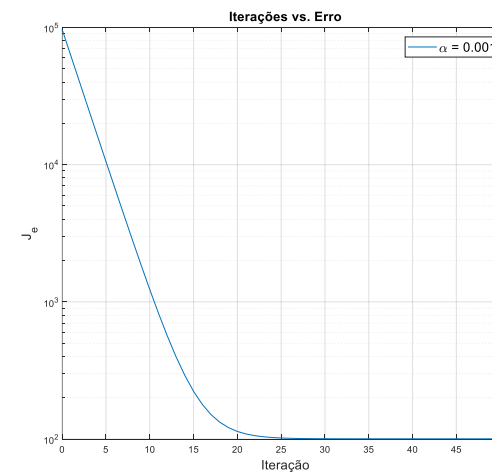
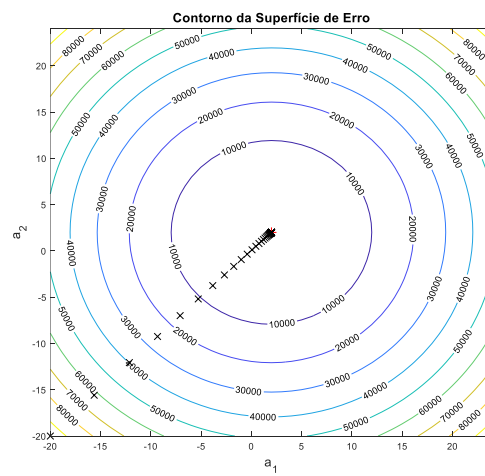
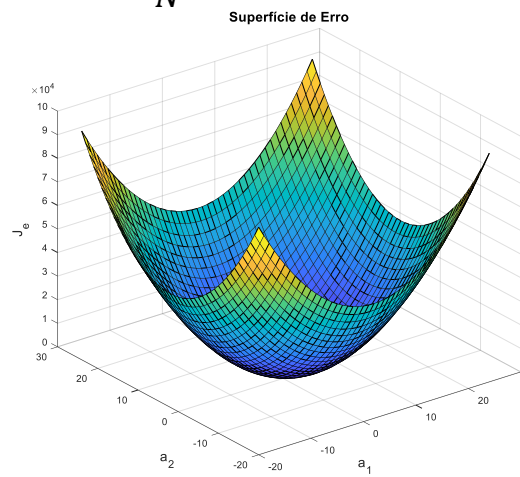
Operação da derivada parcial é distributiva.

E atualização dos pesos  $a_k$ ,  $k = 1$  e  $2$  dada por

$$\frac{\partial J_e(\mathbf{a})}{\partial a_k} = \frac{1}{N} \sum_{n=0}^{N-1} \frac{\partial [y(n) - (a_1 x_1(n) + a_2 x_2(n))]^2}{\partial a_k} = -\frac{2}{N} \sum_{n=0}^{N-1} [y(n) - (a_1 x_1(n) + a_2 x_2(n))] x_k(n), \quad k = 1, 2,$$

$$a_k = a_k - \alpha \frac{\partial J_e(\mathbf{a})}{\partial a_k} \therefore a_k = a_k + \alpha \sum_{n=0}^{N-1} [y(n) - (a_1 x_1(n) + a_2 x_2(n))] x_k(n), \quad k = 1, 2.$$

onde o termo  $\frac{2}{N}$  foi absorvido pelo **passo de aprendizagem**,  $\alpha$ .



# Exemplo #2

[Exemplo: linear regression with gradient descent exemplo2.ipynb](#)

Agora consideramos uma **função hipótese** com os pesos,  $a_0$  e  $a_1$ ,

$$\hat{y}(n) = h(\mathbf{x}(n)) = a_0 + a_1 x_1(n).$$

A **função de erro** é dada por

$$J_e(\mathbf{a}) = \frac{1}{N} \sum_{n=0}^{N-1} [y(n) - (a_0 + a_1 x_1(n))]^2.$$

E a atualização dos pesos  $a_k$ ,  $k = 0$  e  $1$  é dada por

$$\frac{\partial J_e(\mathbf{a})}{\partial a_k} = \frac{1}{N} \sum_{n=0}^{N-1} \frac{\partial [y(n) - (a_0 + a_1 x_1(n))]^2}{\partial a_k} = -\frac{2}{N} \sum_{n=0}^{N-1} [y(n) - (a_0 + a_1 x_1(n))] x_k(n), k = 0, 1,$$
$$a_k = a_k - \alpha \frac{\partial J_e(\mathbf{a})}{\partial a_k} \therefore a_k = a_k + \alpha \sum_{n=0}^{N-1} [y(n) - (a_0 + a_1 x_1(n))] x_k(n), k = 0, 1,$$

onde  $x_0(n) = 1 \forall n$ .

**OBS.1:** Temos o termo de bias nesta função hipótese, portanto, não se esqueçam da coluna de '1's na implementação do código.

**OBS.2:** Para executar este exemplo, é necessário instalar a biblioteca ffmpeg com o comando: `conda install ffmpeg`

# Generalizando a equação de atualização

- Baseado no que vimos nos exemplos anteriores, podemos generalizar a ***equação de atualização do pesos*** da seguinte forma:

$$\frac{\partial J_e(\mathbf{a})}{\partial a_k} = -\frac{2}{N} \sum_{n=0}^{N-1} [y(n) - \hat{y}(n)] x_k(n), \forall k,$$

$$a_k = a_k - \alpha \frac{\partial J_e(\mathbf{a})}{\partial a_k}$$

$$a_k = a_k + \alpha \sum_{n=0}^{N-1} [y(n) - \hat{y}(n)] x_k(n), \forall k.$$

- Essa equação pode ser aplicada a qualquer problema de regressão linear.
- Apenas não se esqueçam de que quando  $k = 0$ ,  $x_0(n) = 1, \forall n$ .

# Versões do Gradiente Descendente

Existem 3 diferentes versões para a implementação do algoritmo do Gradiente Descendente: Batelada, Estocástico e Mini-Batch.

- **Batelada (do inglês *batch*):** a cada iteração (nesse caso, uma época) do algoritmo, ***todos*** os exemplos de treinamento são considerados no processo de treinamento do modelo. Esta versão foi a utilizada nos exemplos anteriores.

$$a_k = a_k + \alpha \sum_{n=0}^{N-1} [y(n) - (a_1 x_1(n) + a_2 x_2(n))] x_k(n), \quad k = 1, \dots, K$$

## Características:

- Utilizado quando se possui previamente todos os atributos e rótulos de treinamento, ou seja, o conjunto de treinamento.
- **Convergência garantida**, dado que o passo de aprendizagem tenha o tamanho apropriado.
- **Convergência pode ser bem lenta**, dado que o modelo é apresentado a todos os exemplos a cada época.

# Versões do Gradiente Descendente

- **Gradiente Descendente Estocástico (GDE)**: também conhecido como **online** ou **incremental** (exemplo-a-exemplo). Com esta versão, os pesos do modelo são atualizados a cada novo exemplo de treinamento.

$$a_k = a_k + \alpha [y(n) - (a_1 x_1(n) + a_2 x_2(n))] x_k(n), \quad k = 1, \dots, K$$

## Características:

- Aproxima o gradiente através de uma **estimativa estocástica**: aproximação através do gradiente calculado com um único exemplo de treinamento.
- Pode ser utilizado quando os atributos e rótulos são obtidos sequencialmente, ou seja, de forma online, exemplo a exemplo.
- Ou quando o conjunto de treinamento é muito grande. Nesse caso, escolhe-se aleatoriamente um par atributo/rótulo a cada iteração (i.e., atualização dos pesos).
- Computacionalmente mais rápido que o GD em batelada.
- **Convergência não é garantida** com um passo de aprendizagem fixo. O algoritmo pode oscilar em torno do mínimo sem nunca convergir para o valores ótimos.
- Esquemas de variação do passo de aprendizagem podem ajudar a garantir a convergência.
- O gradiente ruidoso, calculado com um único exemplo, ajuda o modelo a escapar de regiões com vários mínimos locais ou irregulares para uma região com o mínimo global.

# Versões do Gradiente Descendente

- **Mini-batch:** é um meio-termo entre as duas versões anteriores. O conjunto de treinamento é dividido em vários subconjuntos (mini-batches) com elementos aleatórios (i.e., par atributo/rótulo), onde os pesos do modelo são ajustados a cada mini-batch.

$$a_k = a_k + \alpha \sum_{n=0}^{MB-1} [y(n) - (a_1 x_1(n) + a_2 x_2(n))] x_k(n), \quad k = 1, \dots, K$$

onde  $MB$  é o tamanho do mini-batch.

## Características:

- Pode ser visto como uma generalização das 2 versões anteriores:
  - Caso  $MB = N$ , então se torna o GD em batelada.
  - Caso  $MB = 1$ , então se torna o GD estocástico.
- Computacionalmente mais rápido do que o GD em batelada, mas mais lento do que o GD estocástico.
- Convergência depende do tamanho do mini-batch.
- Pode usar esquemas de variação do passo de aprendizagem para melhorar a convergência.

# Implementação: GD em Batelada

[Exemplo: batch\\_gradient\\_descent\\_with\\_figures.ipynb](#)

```
import numpy as np
```

```
# Define the number of examples.
```

```
N = 1000
```

```
# Generate target function.
```

```
x1 = np.random.randn(N, 1)
```

```
x2 = np.random.randn(N, 1)
```

```
y = x1 + x2 + np.random.randn(N, 1)
```

```
# Concatenate both column vectors, x1 and x2.
```

```
X = np.c_[x1, x2]
```

```
# Constant learning rate.
```

```
eta = 0.1
```

```
# Number of iterations.
```

```
n_iterations = 1000
```

```
# Random initialization.
```

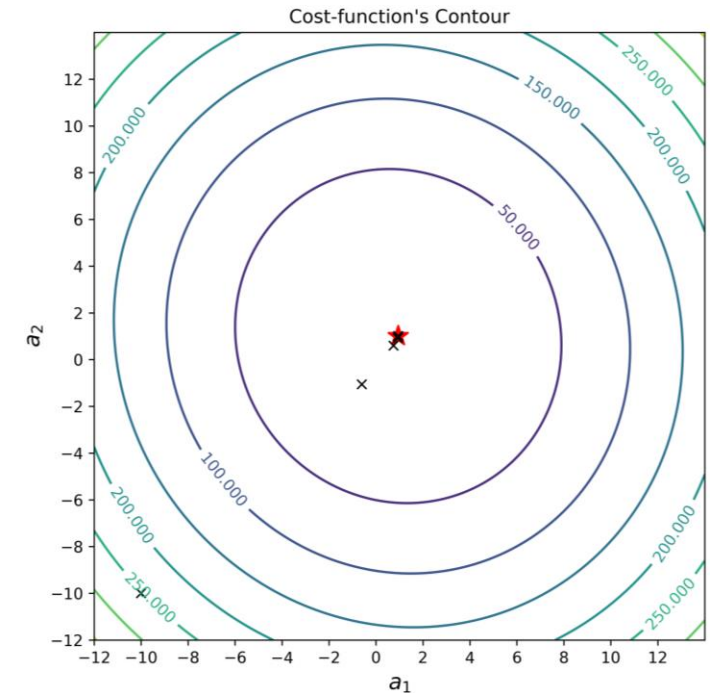
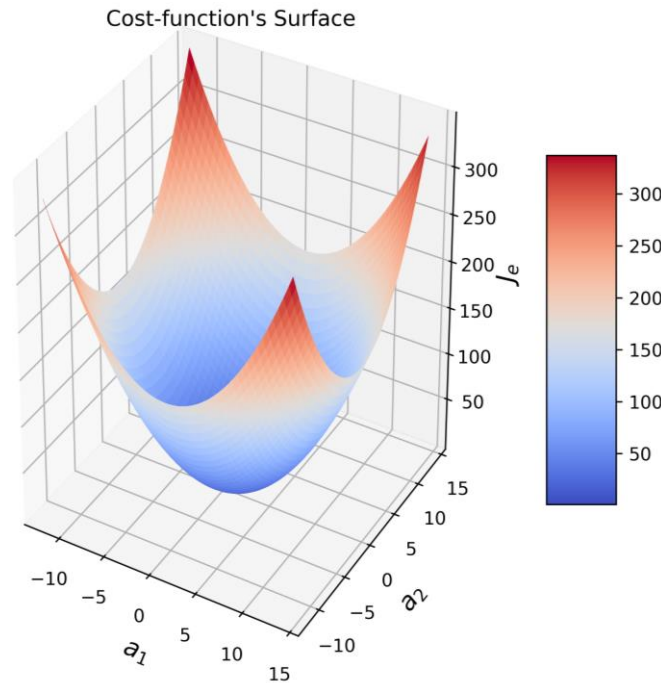
```
a = np.random.randn(2, 1)
```

```
# Batch gradient-descent loop.
```

```
for iteration in range(n_iterations):
```

```
    gradients = -2/N * X.T.dot(y - X.dot(a))
```

```
    a = a - eta * gradients
```



- Segue diretamente para o mínimo global.
- Atinge o mínimo global em 4 épocas.
- Nesse caso específico, segue linha reta entre  $a_0$  e  $a_1$  pois a taxa de decrescimento da superfície de erro é igual para os dois pesos (contornos são circulares).
- Não fica “oscilando” em torno do mínimo após alcançá-lo.
- Algoritmo para no mínimo pois o vetor gradiente no ponto ótimo é praticamente nulo.

# Implementação: GD Estocástico

```
import numpy as np
```

```
# Define the number of examples.
```

```
N = 1000
```

```
# Generate target function.
```

```
x1 = np.random.randn(N, 1)
```

```
x2 = np.random.randn(N, 1)
```

```
y = x1 + x2 + np.random.randn(N, 1)
```

```
# Concatenate both column vectors, x1 and x2.
```

```
X = np.c_[x1, x2]
```

```
# Number of epochs.
```

```
n_epochs = 1
```

```
# Constant learning rate.
```

```
alpha = 0.1
```

```
# Random initialization of parameters.
```

```
a = np.random.randn(2, 1)
```

```
# Stochastic gradient-descent loop.
```

```
for epoch in range(n_epochs):
```

```
    for i in range(N):
```

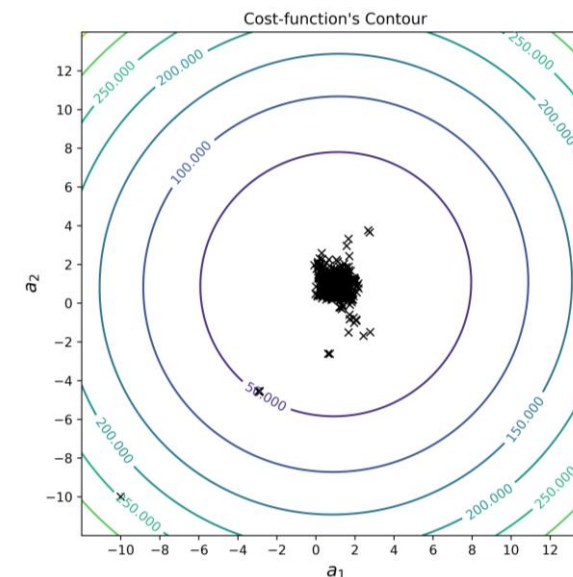
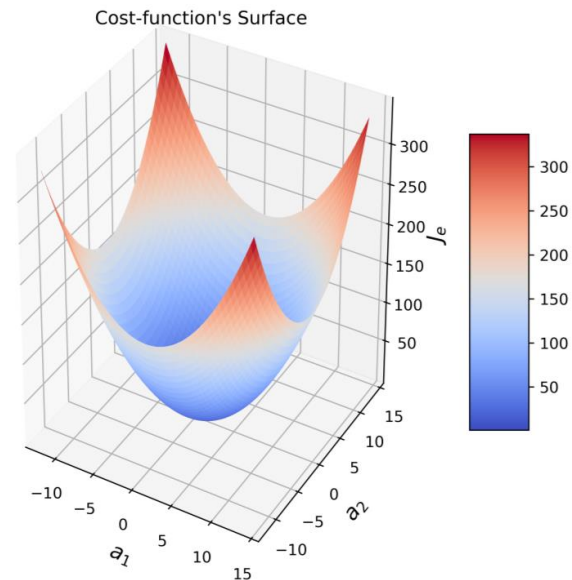
```
        random_index = np.random.randint(N)
```

```
        xi = X[random_index:random_index+1]
```

```
        yi = y[random_index:random_index+1]
```

```
        gradients = -2 * xi.T.dot(yi - xi.dot(a))
```

```
        a = a - alpha * gradients
```



[Exemplo: stochastic gradient descent with figures.ipynb](#)

- Devido à sua natureza estocástica, não apresenta um caminho regular/direto para o mínimo, mudando de direção várias vezes (gradiente ruidoso devido a aproximação do gradiente).
- Por aproximar o gradiente com apenas um exemplo, nem sempre irá na direção ideal, porque as derivadas parciais são "ruidosas".
- O algoritmo não converge suavemente para o mínimo, fica "*oscilando*" ou "*ricocheteando*" em torno dele.
- Quando o treinamento termina, os valores finais dos pesos são bons, mas não são ótimos.
- A convergência ocorre apenas na média.
- Tempo de treinamento é menor, nesse caso, com apenas uma época o algoritmo já se aproxima do ponto ótimo.
- Necessita de um esquema de ajuste do passo de aprendizagem,  $\alpha$ , para ficar mais "*comportado*". Por exemplo, pode-se diminuir o valor do passo conforme o algoritmo caminha em direção ao mínimo.



# Implementação: GD com Mini-Batch

```
import numpy as np
```

```
# Define the number of examples.
```

```
M = 1000
```

```
# Generate target function.
```

```
x1 = np.random.randn(M, 1)
```

```
x2 = np.random.randn(M, 1)
```

```
y = x1 + x2 + np.random.randn(M, 1)
```

```
# Concatenate both column vectors, x1 and x2.
```

```
X = np.c_[x1, x2]
```

```
# Constant learning rate.
```

```
alpha = 0.1
```

```
# Number of iterations.
```

```
n_iterations = 1000
```

```
# Random initialization.
```

```
a = np.random.randn(2,1)
```

```
# Mini-batch size.
```

```
mb_size = 10
```

```
# Mini-batch gradient-descent loop.
```

```
for epoch in range(n_epochs):
```

```
    sdi = random.sample(range(0, N), N)
```

```
    for i in range(0, N//mb_size):
```

```
        bi = sdi[i*mb_size:mb_size*(i+1)]
```

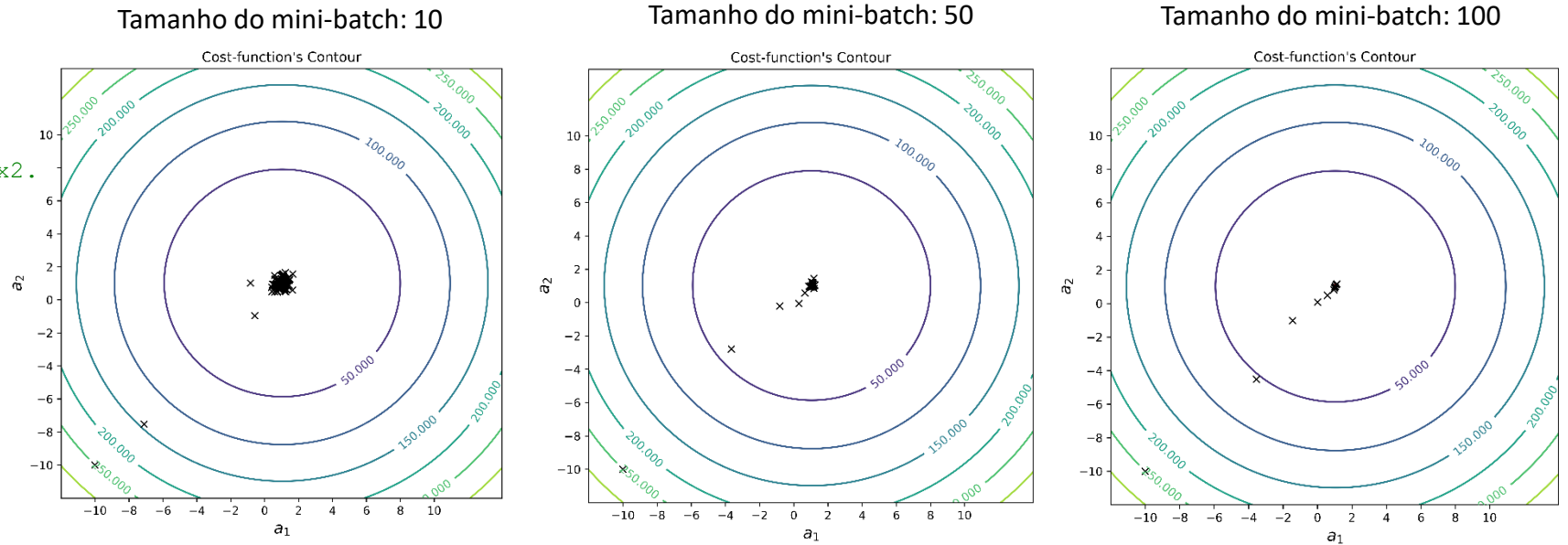
```
        xi = X[bi]
```

```
        yi = y_noisy[bi]
```

```
        gradients = -(2.0/mb_size)*xi.T.dot(yi - xi.dot(a))
```

```
        a = a - alpha*gradients
```

[Exemplo: mini batch gradient descent with figures.ipynb](#)



- O progresso do algoritmo é menos irregular do que com o GD estocástico, especialmente com mini-batches grandes o suficiente.
- Como resultado, o mini-batch oscila menos ao redor do mínimo global do que o GDE.
- Tem comportamento mais próximo do GD em batelada para mini-batches maiores.
- Oscilação em torno do mínimo diminui conforme o tamanho do mini-batch aumenta.
- Pode também ser usado com um esquema de variação do passo de aprendizagem.

# Comparação das versões de GD

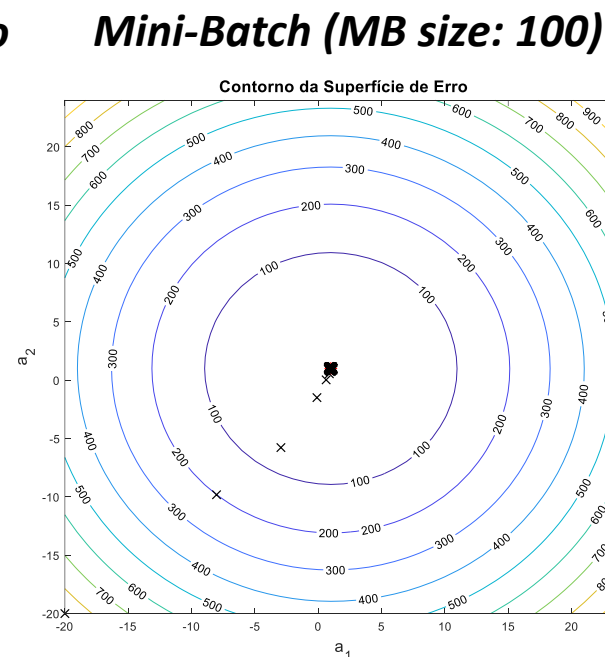
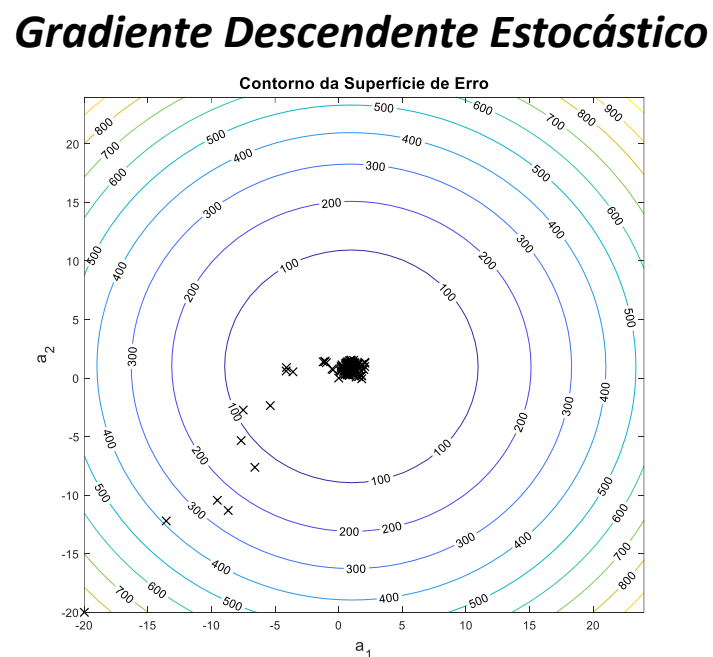
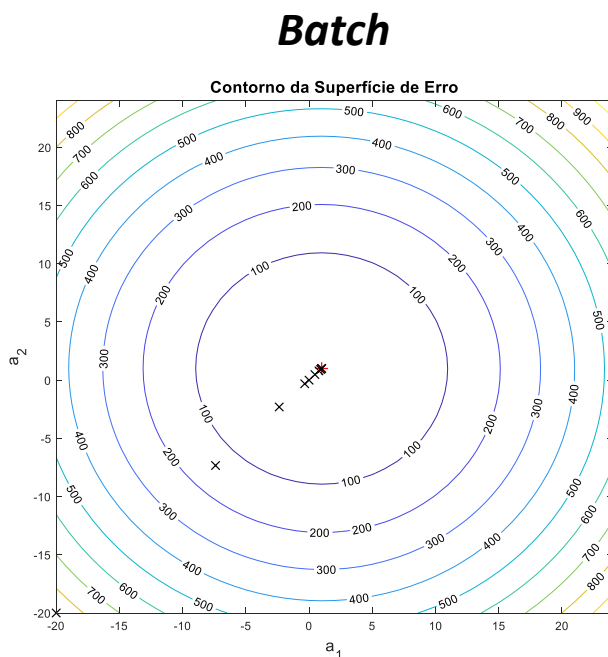
- Todos se aproximam do mínimo, mas o GD em **batch** caminha diretamente para lá.
- Enquanto **GDE** e o **mini-batch** continuam a caminhar (*dançar*) ao redor do mínimo (tangente diferente de zero, pois estimativa do gradiente é ruidosa).
- O progresso do **mini-batch** é menos irregular do que com o **GDE**, mas depende do tamanho do mini-batch.
- Mas não se esqueçam, o **batch** leva mais tempo para executar cada época do que o **GDE** e o **mini-batch**, os quais também podem alcançar o mínimo e se estabilizarem lá caso uma boa estratégia para ajuste do **passo de aprendizagem** seja usada.

Função observável

$x1 = \text{randn}(N, 1)$

$x2 = \text{randn}(N, 1)$

$y = x1 + x2 + \text{randn}(N, 1)$

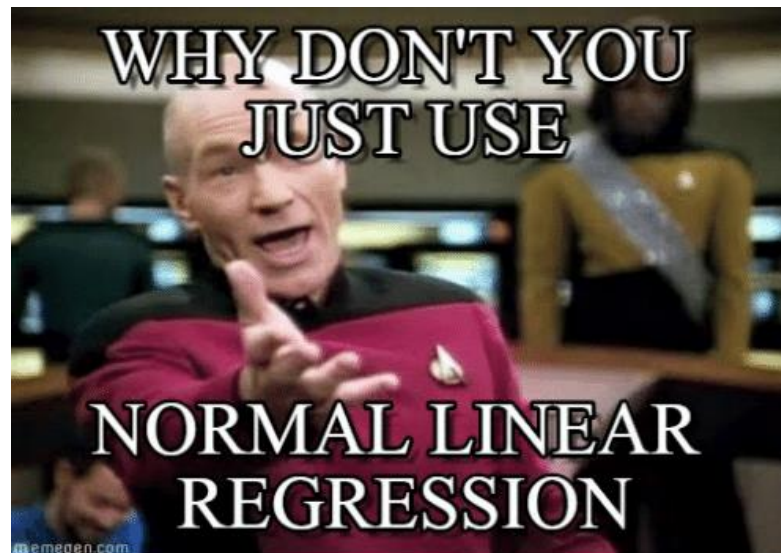
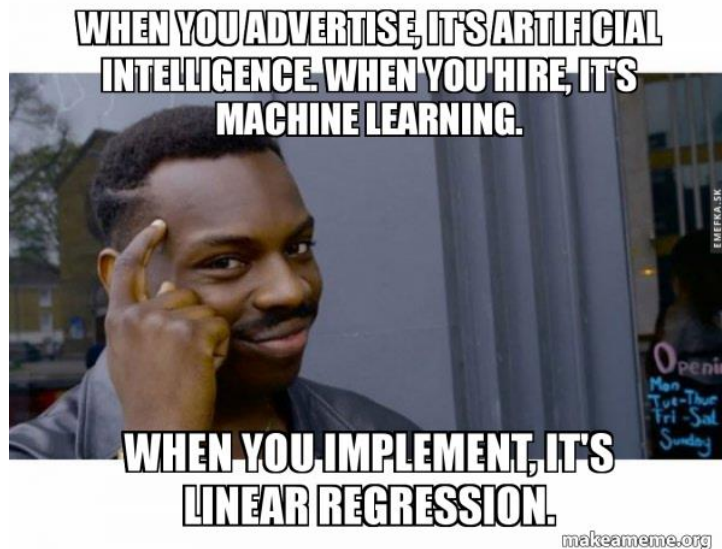
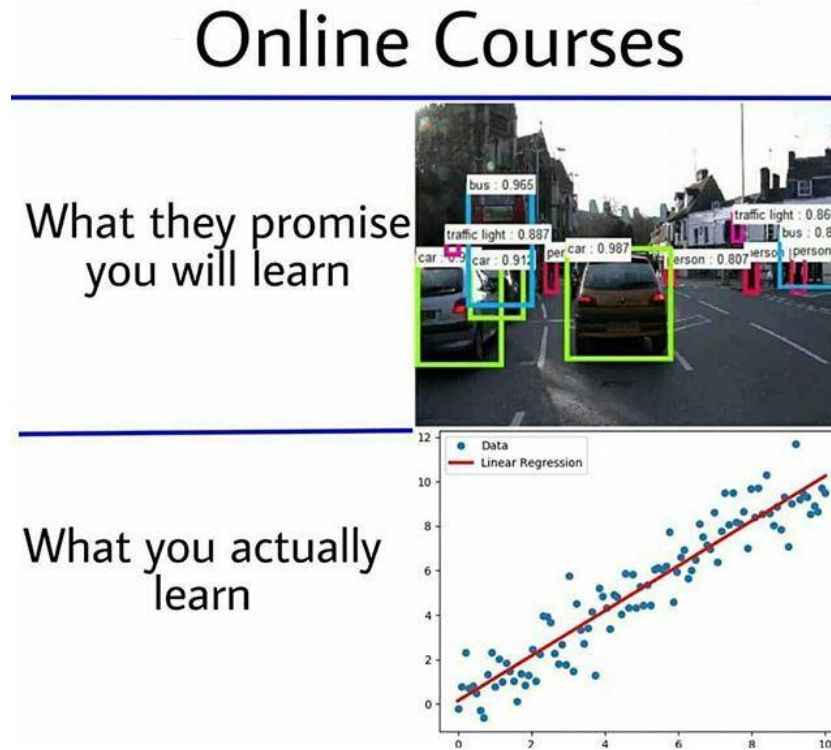


# Tarefas

- **Quiz:** “*T319 - Quiz - Regressão: Parte II (1S2021)*” que se encontra no MS Teams.
- **Exercício Prático:** [Laboratório #3](#).
  - Pode ser baixado do MS Teams ou do GitHub.
  - Pode ser respondido através do link acima (na nuvem) ou localmente.
  - [Instruções para resolução e entrega dos laboratórios](#).

Obrigado!





LEARNING ML/DL  
FROM UNIVERSITY

ONLINE COURSES

FROM YOUTUBE

FROM ARTICLES

FROM MEMES

