Universidade Federal
de Ouro Preto

Gradiente Descendente

Grupo 6

Gabriel Costa, Lucas Rodrigues,
Alexsandro, Jean Pierre

1 Visão geral:

Em matemática, gradiente descendente (também chamado de descida mais íngreme) é um algoritmo de otimização iterativa de primeira ordem para encontrar um mínimo local de uma função diferenciável. A ideia é dar passos repetidos na direção oposta do gradiente (ou gradiente aproximado) da função no ponto atual, pois essa é a direção de descida mais acentuada. Por outro lado, caminhar na direção do gradiente levará a um máximo local dessa função; o procedimento é então conhecido como subida de encosta.

A descida do gradiente é geralmente atribuída a Cauchy (1789 - 1857), que a sugeriu pela primeira vez em 1847. Hadamard (1865 - 1963) propôs independentemente um método semelhante em 1907. Suas propriedades de convergência para problemas de otimização não linear foram estudadas pela primeira vez por Haskell Curry (1900 - 1982) em 1944, com o método se tornando cada vez mais bem estudado e utilizado nas décadas seguintes.

A descida de gradiente é baseada na observação de que se uma função multivariável $F(x)$ é definida e diferenciável em uma vizinhança de um ponto a, então $F(x)$ diminui mais rápido se for de a na direção do gradiente negativo de F em a , $^{-}\nabla F(a)$. Segue-se que, se:

$$a_{n+1} = a_n - \gamma \nabla F(a_n)$$

para um tamanho de passo pequeno o suficiente ou **learning rate** $\gamma \in \mathbb{R}_+$, então $F(a_n) \geq F(a_{n+1})$. Em outras palavras, o termo $\gamma \nabla F(a)$ é subtraído de a porque queremos nos mover contra o gradiente, em direção ao mínimo local. Com esta observação em mente, começa-se com um palpite x_0 para um mínimo local de F, e considera-se a sequência x_0, x_1, x_2, ... tal que

$$x_{n+1} = x_n - \gamma_n \nabla F(x_n), n \geq 0.$$

Nós temos uma sequência monotônica

$$F(x_0) \geq F(x_1) \geq F(x_2) \geq ...,$$

então, esperamos que a sequência x_n convirja para o mínimo local desejado. Observe que o valor do tamanho do passo γ pode mudar a cada iteração. Com certas suposições sobre a função F (por exemplo, F **convexa** e ∇F Lipschitz) e escolhas particulares de γ. Quando a função F é convexa, todos os mínimos locais também são mínimos globais, então neste caso o gradiente descendente pode convergir para a solução global.

Este processo é ilustrado na imagem abaixo. Aqui, assume-se que F está definido no plano e que seu gráfico tem uma forma de tigela. As curvas azuis são as curvas de nível, ou seja, as regiões nas quais o valor de F é constante. Uma seta vermelha originada em um ponto mostra a direção do gradiente negativo naquele ponto. Observe que o gradiente em um ponto é ortogonal à linha de contorno que passa por esse ponto. Vemos que o gradiente descendente nos leva ao fundo da tigela, ou seja, ao ponto em que o valor da função F é mínimo.

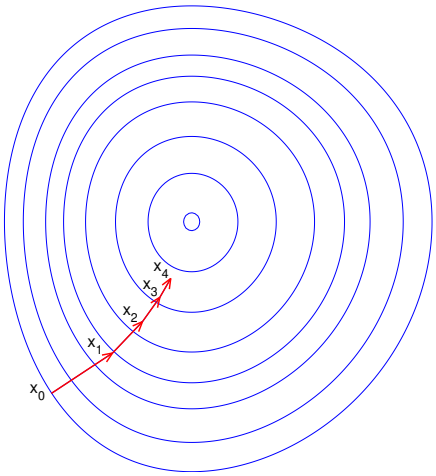


Figura 1: Ilustração de descida de gradiente em uma série de curvas de nível

2 Analogia:

A intuição básica por trás do gradiente descendente pode ser ilustrada por um cenário hipotético. Uma pessoa está presa nas montanhas e está tentando descer (ou seja, tentando encontrar o mínimo global). Há forte neblina de tal forma que a visibilidade é extremamente baixa. Portanto, o caminho para baixo da montanha não é visível, então eles devem usar informações locais para encontrar o mínimo. Eles podem usar o método de descida de gradiente, que envolve observar a inclinação da colina em sua posição atual e prosseguir na direção da descida mais íngreme (ou seja, descida). Se eles estivessem tentando encontrar o topo da montanha (ou seja, o máximo), então eles seguiriam na direção da subida mais íngreme (ou seja, para cima). Usando esse método, eles eventualmente encontrariam o caminho para baixo da montanha ou possivelmente ficariam presos em algum buraco (ou seja, ponto mínimo local ou ponto de sela), como um lago de montanha. No entanto, suponha também que a inclinação do morro não é imediatamente óbvia com a simples observação, mas requer um instrumento sofisticado para medir, que a pessoa tem no momento. Leva algum tempo para medir a inclinação da colina com o instrumento, portanto, eles devem minimizar o uso do instrumento se quiserem descer a montanha antes do pôr do sol. A dificuldade então é escolher a frequência com que devem medir a inclinação do morro para não sair da trilha.

Nessa analogia, a pessoa representa o algoritmo e o caminho percorrido pela montanha representa a sequência de configurações de parâmetros que o algoritmo explorará. A inclinação da colina representa a inclinação da superfície de erro naquele ponto. O instrumento usado para medir a inclinação é a diferenciação (a inclinação da superfície de erro pode ser calculada tomando a

derivada da função de erro ao quadrado naquele ponto). A direção que eles escolhem para viajar se alinha com o gradiente da superfície de erro naquele ponto. A quantidade de tempo que eles viajam antes de fazer outra medição é o tamanho do passo.

Figura 2: Nevoeiro nas montanhas

3 Regressão linear:

Um modelo de regressão linear tenta explicar a relação entre uma variável dependente e uma ou mais variáveis independentes usando uma reta. Essa reta é representada pela seguinte fórmula:

$$y = a_0 x + a_1$$

Onde,
y: variável dependente,
x: variável independente,
a_0: inclinação da reta,
a_1: intercepto em y.

O primeiro passo para encontrar uma equação de regressão linear é determinar se existe uma relação entre as duas variáveis. Podemos fazer isso usando o coeficiente de correlação e o gráfico de dispersão. Quando um coeficiente de correlação mostra que os dados provavelmente são capazes de prever resultados futuros e um gráfico de dispersão dos dados parece formar uma linha reta, podemos usar a regressão linear simples para encontrar uma função preditiva.

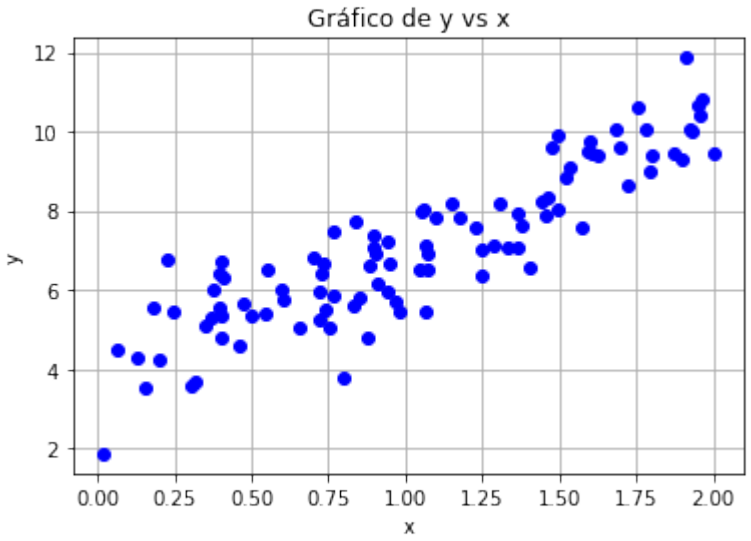
Agora consideremos um exemplo para nossa regressão linear: Vamos começar com alguns dados, melhor ainda vamos criar alguns dados lineares com algum ruído gaussiano aleatório.

```
x = 2 * np.random.rand(100, 1)
y = 4 + 3 * x + np.random.randn(100, 1)

plt.plot(x, y, 'bo')
plt.title('Grafico de y vs x')
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True)
plt.show()
```

Listing 1: Criação e plot dos dados

Em seguida, vamos visualizar os dados:



É evidente que y tem uma relação linear com x. Esses dados são muito simples e possuem apenas uma variável independente x.

Podemos resolver a equação da reta para a_0 e a_1 da seguinte maneira:

$$a_0 = \frac{n\sum_{i=1}^{n} x_i y_i - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2}$$

$$a_1 = \frac{(\sum_{i=1}^{n} x_i^2)(\sum_{i=1}^{n} y_i) - (\sum_{i=1}^{n} x_i y_i)(\sum_{i=1}^{n} x_i)}{n\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2}$$

Acima temos o método analítico de resolver a equação da reta. Nada de errado com isso, no entanto, lembre-se de que o machine learning é sobre programação em matrizes. Por causa disto, podemos expressar a equação para uma reta em termos do machine learning de uma maneira diferente. Denominamos

y como nossa hipótese e a representamos como $J(\theta)$ e chamamos a_1 de θ_0 e a_0 de θ_1. Podemos escrever a mesma equação da reta como:

$$J(\theta) = \theta_0 + \theta_1 x$$

Para resolver a forma analítica θ_0 e θ_1 teríamos que escrever o seguinte programa:

```
theta = np.linalg.inv(x.T.dot(X)).dot(x.T).dot(y)
```

Listing 2: "linalg"é uma função do NumPy que calcula o inverso (multiplicativo) de uma matriz

Lembre-se que adicionamos uma unidade de inclinação a x que é 1 para cada vetor em x. Isso facilita a multiplicação de matrizes para resolver θ.

Complexidade da Regressão Linear: Temos que resolver:

$$(X'X)^{-1}X'Y$$

Onde X é uma matriz np. Agora, em geral, a complexidade do produto de matriz AB é $O(abc)$ sempre que A é ab e B é bc. Portanto, podemos avaliar as seguintes complexidades:

a) o produto da matriz $X'X$ com complexidade $O(p^2n)$,

b) o produto matriz-vetor $X'Y$ com complexidade $O(pn)$,

c) o inverso $(X'X)^{-1}$ com complexidade $O(p^3)$,

Portanto, a complexidade é $\mathbf{O(np^2 + p^3)}$.

Podemos notar que se o número de recursos em x começar a aumentar, a carga na CPU / GPU para fazer a multiplicação da matriz começará a aumentar e se o número de recursos for realmente grande como um milhão de recursos, quase se tornará inviável para o seu computador resolver isso. É aqui que o gradiente descendente entra.

Usando nossa analogia da seção 2:

1. Tamanho dos passos dados em qualquer direção → learning rate
2. Informação da altura → função de custo
3. Direção dos passos → gradiente

Função de custo:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h(\theta)^i - y^i)^2$$

Gradiente:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h(\theta)^i - y^i)X_j^i$$

Onde m é o número de observações.

Nossa função de custo é comumente conhecida como **MSE** (mean squared error ou erro quadrático médio), uma função convexa e por isso o gradiente descendente pode ser aplicado sem quaisquer problemas.

Estamos prontos para escrever nosso próprio gradiente descendente. Quais são as coisas que precisamos?

1. Uma função de custo
2. Uma função de gradiente descendente que calcula o novo vetor θ

Código da função de custo:

```
def Custo(theta, x, y):
    m = float(len(y))
    Predicao = x.dot(theta)
    Custo = (1 / 2 * m) * np.sum(np.square(Predicao - y))
    return Custo
```

Código da função do Gradiente Descendente:

```
def GradienteDescendente(x, y, theta, LearningRate = 0.01, Itmax =
    100):
    '''
    x = Matriz de x com unidades de inclinacao adicionadas
    y = Vetor de y
    theta = Vetor de thetas np.random.randn(j, 1)
    Learning Rate
    Itmax = Nro maximo de iteracoes

    Retorna o vetor theta final e o vetor do historico de custos
    das iteracoes
    '''
    m = float(len(y))
    HistoricoCusto = np.zeros(Itmax)
    HistoricoTheta = []
    for i in range(Itmax):
        Predicao = np.dot(x, theta)
        theta = theta - (1 / m) * LearningRate * (x.T.dot((Predicao
     - y)))
        HistoricoTheta.append(theta.T)
        HistoricoCusto[i] = Custo(theta, x, y)
    return theta, HistoricoCusto, HistoricoTheta
```

Complexidade do Gradiente Descendente: O custo do modelo é avaliada pelo MSE. O passo básico é a atualização dos parâmetros θ_0 e $\theta 1$. Este passo é realizado duas vezes a cada iteração. Portanto, a complexidade é: $\mathbf{O(2k)}$, onde k é o número de iterações executadas para atingir o custo mínimo.

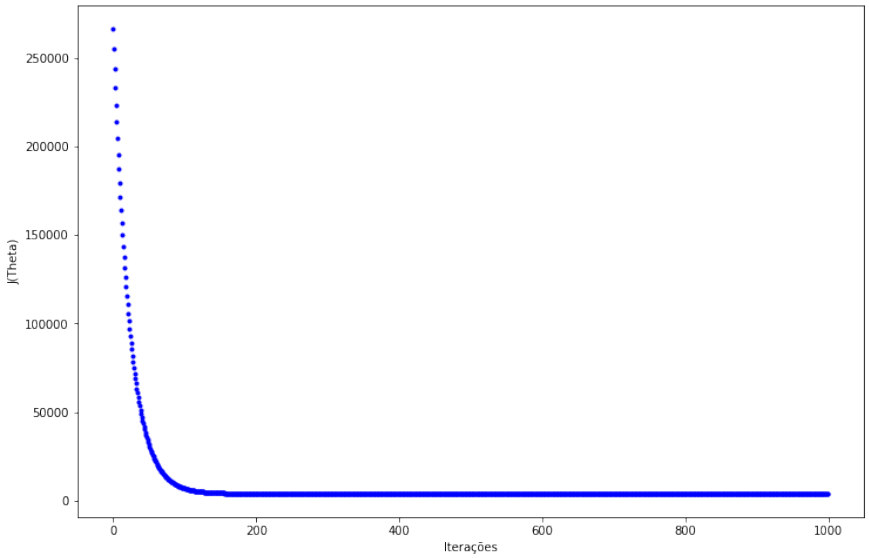
Começando com 1.000 iterações e um learning rate de 0,01 (além do θ de uma distribuição gaussiana) obtivemos a seguinte saída:

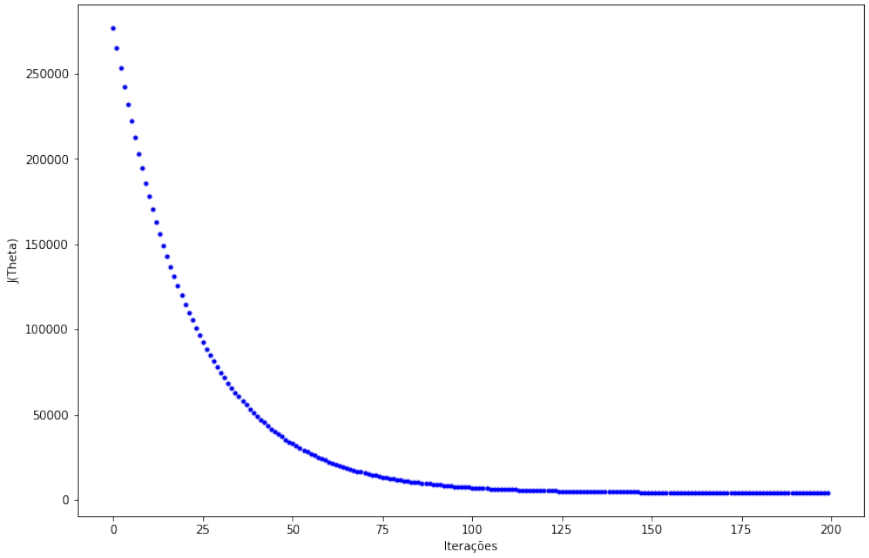
Theta0: 3.180,
Theta1: 3.689,
Custo Final/MSE: 4331.358

Plotando o histórico de custos versus iterações:

Olhando para o gráfico é evidente que o custo após cerca de 180 iterações não diminui, o que significa que podemos usar apenas 200 iterações. Se ampliarmos o gráfico, podemos notar isso.

Também podemos notar que o custo reduz mais rápido inicialmente e depois diminui.



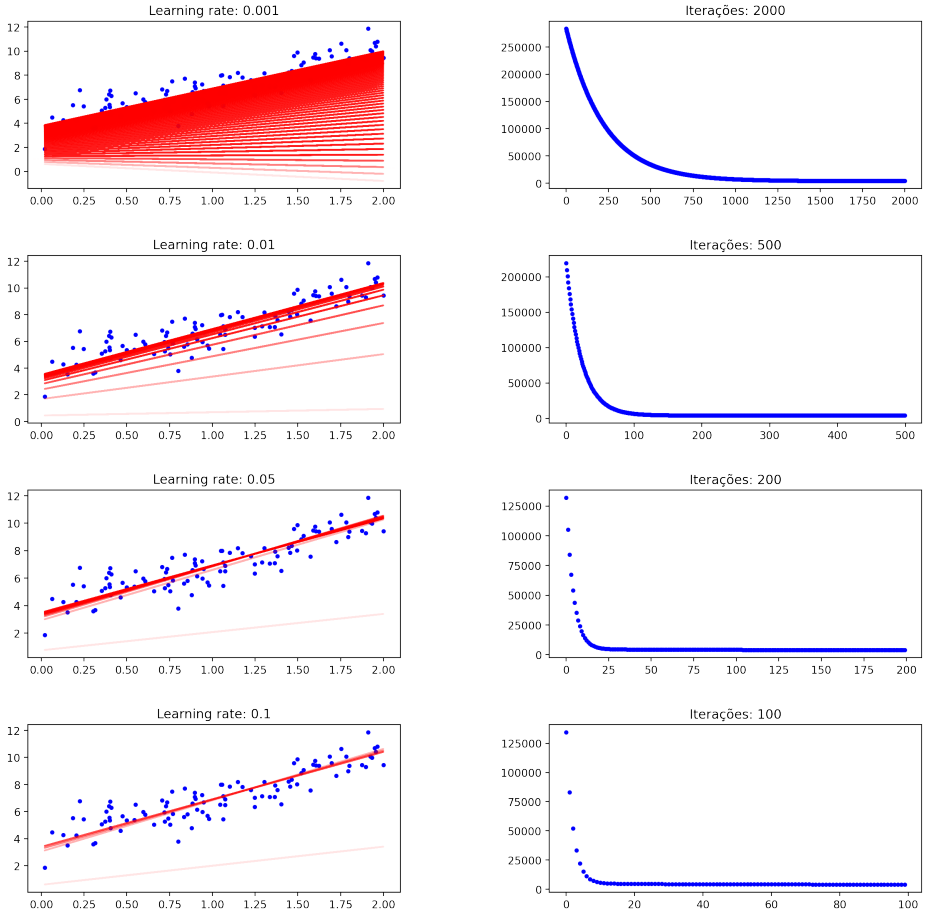


Seria ótimo ver como o gradiente descendente realmente converge para a solução com diferentes learning rates e iterações. E isso faria mais sentido se pudéssemos ver tudo de uma só vez.

Plotando os gráficos para convergência e custo versus iterações para as quatro combinações de iterações e learning rates:

Pares iterações e learning rates $\rightarrow (2000; 0,001), (500; 0,01), (200; 0,05), (100; 0,1)$

Verificamos que com learning rates pequenos demoramos muito para convergir para a solução, enquanto com learning rates maiores a convergência é mais



rápida.

Observação: o learning rate real para o seu problema depende dos dados e não há uma fórmula geral para defini-lo corretamente. No entanto, existem algoritmos de otimização sofisticados que começam com learning rates maiores e depois reduzem lentamente à medida que nos aproximamos da solução, por exemplo: otimizador ADAM.

4 Outros tipos de gradiente descendente:

4.1 Gradiente descendente estocástico:

Este tipo é muito aplicado na área das **redes neurais** e **deep learning**.

Quando estamos atualizando os pesos de uma rede neural, a partir de uma função de custo, o que esta função faz é comparar a previsão que a nossa rede

neural está fazendo – a classificação que ela está tendo – com o resultado real – o gabarito. Essa função nada mais é do que a soma de todos os erros; e o nosso objetivo é minimizar esses erros, de forma que a rede neural fique cada vez mais precisa.

Vamos supor que estejamos treinando uma rede neural com uma série de imagens, 10 mil, por exemplo. Nesse caso, nós poderíamos atualizar todos os pesos da rede neural com base somente em uma imagem. Esse cenário não seria o ideal pois faria com que a rede ficasse viciada: ela aprenderia muito bem como reconhecer aquela imagem específica mas talvez não fosse uma rede que generalizasse muito bem, se entrássemos com outra imagem, talvez ela não saberia classificá-la.

Portanto, o ideal seria atualizar os pesos numa iteração com base em todas as imagens de entrada. Ou seja, estaríamos somando os erros para cada uma das imagens – cada uma das amostras – e a partir dessa soma total é que tentaríamos reduzir a função de custo – não somente em relação a uma imagem. Isso gera, no entanto, um custo computacional elevado.

Nosso exemplo de 10 mil imagens já parece grande, mas se analisarmos casos de big data – que tem um conjunto de dados muito maior – esse custo computacional aumenta de forma brutal, pois quanto maior é o nosso conjunto de dados de treinamento, maior é o custo associado.

Dessa forma, o método de gradiente descendente pode ser muito prejudicado, o que faz com que as atualizações sejam muito lentas. Talvez até se consiga chegar em um modelo bem calibrado; mas levará muito tempo para que se consiga fazer a rede neural chegar a esse ponto. Também é possível que o nosso computador nem consiga fazer isso; ele pode simplesmente travar porque atingiu o limite de memória.

Como podemos, então, tentar contornar esse problema do custo computacional? É aí que entra o gradiente descendente estocástico. Em vez de pagar todo o conjunto de dados, ele pega somente uma parte desses dados para fazer cada atualização dos pesos. Obviamente não estaremos usando somente uma imagem, pois já comentamos que isso deixaria o modelo muito enviesado, mas por que não utilizar um número como 200 imagens para cada atualização? Se, em cada atualização dos pesos da rede neural, estivermos passando um grupo de imagens diferente e genérico o bastante, talvez o algoritmo esteja caminhando na direção certa em direção ao ponto ótimo da função de custo.

Sumarizando: é chamado de estocástico porque as amostras são selecionadas aleatoriamente (ou embaralhadas) em vez de como um único grupo (como no gradiente descendente padrão) ou na ordem em que aparecem no conjunto de treinamento.

Código do gradiente descendente estocástico:

```
1 def GradienteDescendenteEstocastico(x, y, theta, LearningRate =
    0.01, Itmax = 10):
2     m = len(y)
3     HistoricoCusto = np.zeros(Itmax)
4     for i in range(Itmax):
5         Custo_AT = 0
6         for j in range(m):
```

```
            rand_ind = np.random.randint(0, m)
            x_i = x[rand_ind, :].reshape(1, x.shape[1])
            y_i = y[rand_ind].reshape(1, 1)
            Predicao = np.dot(x_i, theta)
            theta = theta - (1 / float(m)) * LearningRate * (x_i.T.
    dot(Predicao - y_i))
            Custo_AT += Custo(theta, x_i, y_i)
        HistoricoCusto[i] = Custo_AT
    return theta, HistoricoCusto
```

Complexidade: A grande vantagem do gradiente descendente estocástico é sua eficiência, que é basicamente linear no número de exemplos de treinamento. Se X é uma matriz de tamanho (n, p), o treinamento tem um custo de $\mathbf{O(knp)}$, onde k é o número de iterações e p é o número médio de atributos diferentes de zero por amostra.

Vantagens:

1. As atualizações frequentes fornecem imediatamente uma visão sobre o desempenho do modelo e a taxa de melhoria.
2. Essa variante de gradiente descendente pode ser a mais simples de entender e implementar, especialmente para iniciantes.
3. O aumento da frequência de atualização do modelo pode resultar em um aprendizado mais rápido em alguns problemas.
4. O processo de atualização ruidoso pode permitir que o modelo evite mínimos locais (por exemplo, convergência prematura).

Desvantagens:

1. Atualizar o modelo com tanta frequência é computacionalmente mais caro do que outras configurações de gradiente descendente, levando muito mais tempo para treinar modelos em grandes conjuntos de dados.
2. As atualizações frequentes podem resultar em um sinal de gradiente ruidoso, o que pode fazer com que os parâmetros do modelo e, por sua vez, o erro do modelo salte (ter uma variação maior em relação ao conjunto de treinamento).
3. O processo de aprendizado ruidoso ao longo do gradiente de erro também pode tornar difícil para o algoritmo estabelecer um erro mínimo para o modelo.

4.2 Gradiente descendente em mini-lotes:

Essa abordagem usa amostras aleatórias, mas em lotes. O que isso significa é que não calculamos os gradientes para cada observação, mas para um grupo de observações que resulta em uma otimização mais rápida. Uma maneira simples de implementar é embaralhar as observações e criar lotes e prosseguir com a descida do gradiente usando lotes.

O gradiente descendente em mini-lotes procura encontrar um equilíbrio entre a robustez do gradiente descendente estocástico e a eficiência do gradiente descendente em lote. É a implementação mais comum usada no campo de deep learning.

Código do gradiente descendente em mini-lotes:

```
def GradienteDescendenteMiniLotes(x, y, theta, LearningRate = 0.01,
     Itmax = 10, TamLote = 20):
    m = len(y)
    HistoricoCusto = np.zeros(Itmax)
    n_lotes = int(m / TamLote)
    for i in range(Itmax):
        Custo_AT = 0
        Indices = np.random.permutation(m)
        X = x[Indices]
        y = y[Indices]
        for j in range(0, m, TamLote):
            X_i = x[j:j + TamLote]
            y_i = y[j:j + TamLote]
            X_i = np.c_[np.ones(len(X_i)), X_i]
            Predicao = np.dot(X_i, theta)
            theta = theta - (1/m) * LearningRate * (X_i.T.dot((
    Predicao - y_i)))
            Custo_AT += Custo(theta, X_i, y_i)
        HistoricoCusto[i] = Custo_AT

    return theta, HistoricoCusto
```

Complexidade: O passo básico é a atualização dos parâmetros θ_0 e θ_1. Esta etapa é executada para o número $\frac{2m}{p}$ de vezes a cada iteração.

Aqui p representa o tamanho do lote e $\frac{m}{p}$ representa o número de lotes presentes no conjunto de dados. Como a atualização acontece para cada lote em uma iteração, portanto, a complexidade é dada por: $\mathbf{O}(\frac{\mathbf{2m}}{\mathbf{p}}\mathbf{k})$, onde k é o número de iterações executadas para atingir o custo mínimo.

Vantagens:

1. A frequência de atualização do modelo é maior que o gradiente descendente em lote, o que permite uma convergência mais robusta, evitando mínimos locais.

2. As atualizações em lote fornecem um processo computacionalmente mais eficiente do que o gradiente descendente estocástico.

3. O loteamento permite tanto a eficiência de não ter todos os dados de treinamento na memória quanto as implementações de algoritmos.

Desvantagens:

1. O mini-lote requer a configuração de um hiperparâmetro adicional de "tamanho do mini-lote" para o algoritmo de aprendizado.

2. As informações de erro devem ser acumuladas em mini-lotes de exemplos de treinamento, como no gradiente descendente em lotes.

5 Aprofundando no learning rate:

O learning rate é definitivamente o mais importante de todos os hiperparâmetros, então vamos gastar um certo tempo discutindo como ele influencia no aprendizado e como ajustá-lo bem.

O mesmo define o tamanho dos passos que daremos em direção ao mínimo em cada iteração. Se esses passos forem muito pequenos, é quase garantido que

chegaremos ao ponto de mínimo da função, mas para isso talvez precisaremos de muitas iterações de treino, tornando o algoritmo desnecessariamente lento.

Por outro lado, se colocarmos um learning rate muito alto, pode acontecer de sermos catapultados para cima da função custo e irmos cada vez mais longe do mínimo, resultando em uma falha completa de aprendizado. Isso acontecerá quando o passo que dermos for tão grande que pulará o ponto de mínimo e chegará em um ponto na função de custo mais alto do que o de onde saímos. Nesse novo ponto, o gradiente será ainda maior, aumentando mais ainda o passo seguinte e nos arremessando ainda mais longe do ponto de mínimo a cada iteração.

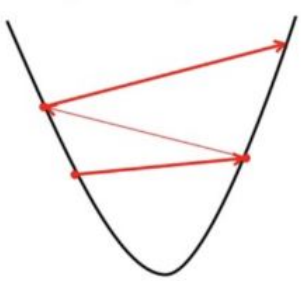


Figura 3: Se escolhermos um learning rate muito grande, o gradiente descendente pode ultrapassar o mínimo. Pode não convergir ou até divergir.

Figura 4: Se escolhermos um learning rate que seja muito pequeno, o gradiente descendente dará pequenos passos para atingir os mínimos locais e levará mais tempo para atingir os mesmos.

Podemos ver que o learning rate não deve ser nem tão grande, nem tão pequeno. Uma sugestão de ajustamento desse hiper-parâmetro é começar com 0.01 e explorar os pontos em volta dez vezes maior/menor (isto é, 0.1 e 0.001). Na maioria dos casos, um bom learning rate será algum dos seguintes valores: 1, 0.1, 0.01, 0.001, 0.0001, 0.00001.

6 Bibliografia:

- Livro → Python para análise de dados: Tratamento de dados com Pandas, NumPy e IPython - *Wes McKinney*
- GeeksForGeeks → https://www.geeksforgeeks.org/
- Towards Data Science → https://towardsdatascience.com/
- Livro → Otimização - volume 2. Métodos computacionais - *Alexey Izmailov, Mikhail Solodov*
- IBM Brasil → https://www.ibm.com/br-pt