SỞ GIÁO DỤC VÀ ĐÀO TẠO THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG THPT CHUYÊN TRẦN ĐẠI NGHĨA

ଌ★★★ଌ

ĐỀ TÀI: Gradient Descent

Giáo viên hướng dẫn: Hồ Ngọc Lâm

Lớp: 12CTin

Nhóm thực hiện:

05 – Phan Lê Tường Bách

11 – Hồ Thị Hồng Lạc

13 – Lê Hoàng Long

22 – Phan Đình Minh Quân

Tp. Hồ Chí Minh, ngày 09 tháng 01 năm 2023

Mục lục

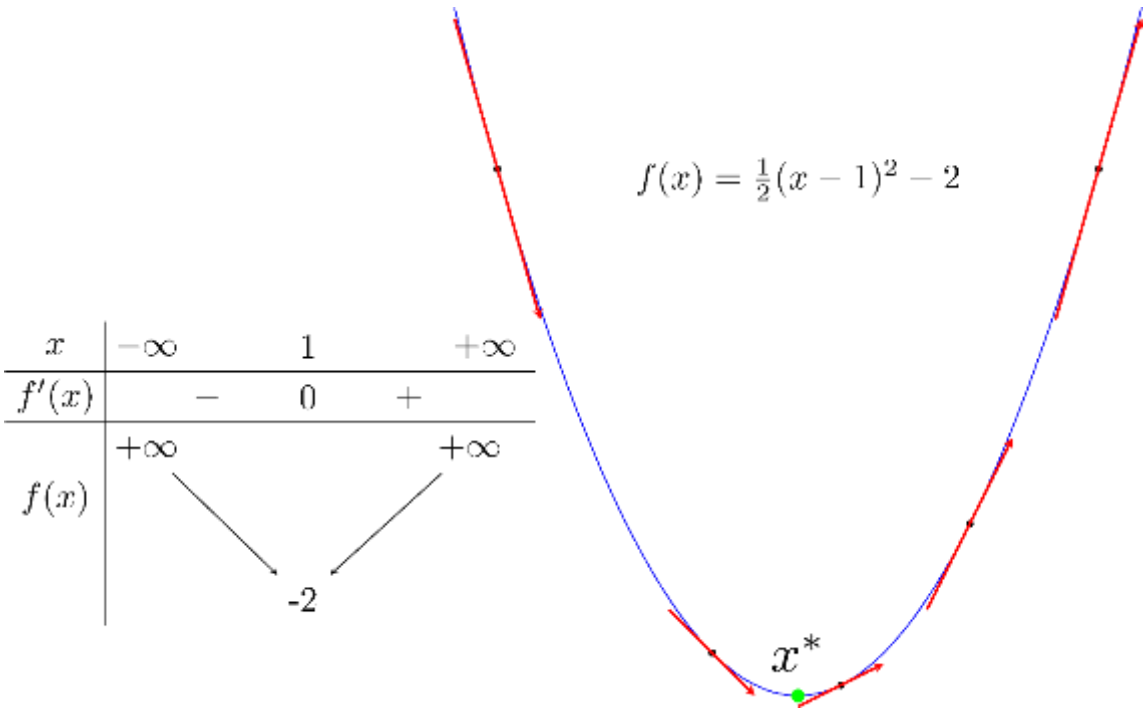
1 Giới thiệu Gradient Descent

Trong Machine Learning và Toán tối ưu nói chung, ta thường xuyên phải tìm giá trị nhỏ nhất (hoặc lớn nhất) của một hàm số nào đó (ví dụ như các hàm mất mát trong hai bài Linear Regression và K-means Clustering). Nhìn chung việc tìm global minimum trong Machine Learning rất phức tạp, thậm chí có thể là bất khả thi. Thay vào đó, ta thường cố gắng tìm các điểm local minimum, ở mức độ nào đó, là nghiệm cần tìm cho bài toán.

Các điểm local minimum là nghiệm của phương trình đạo hàm bằng 0. Nếu bằng cách nào đó tìm được toàn bộ (hữu hạn) các điểm local minimum, ta chỉ cần thay toàn bộ vào hàm số rồi tìm global minimum. Tuy vậy, hầu hết các trường hợp thì giải phương trình đạo hàm bằng 0 là bất khả thi. Nguyên nhân có thể đến từ sự phức tạp của dạng đạo hàm, từ việc các điểm dữ liệu có số chiều lớn, hoặc từ việc có quá nhiều điểm dữ liệu.

Hướng tiếp cận phổ biến là xuất phát từ một điểm mà chúng ta coi là gần với nghiệm của bài toán, sau đó dùng một phép toán lặp để tiến dần đến điểm cần tìm, tức là đến khi đạo hàm gần với 0. Gradient Descent và các biến thể của nó là một trong những phương pháp được dùng nhiều nhất.

2 Gradient Descent cho hàm 1 biến



Nhìn hình ta thấy, điểm màu xanh lục x^* là điểm local minimum (cực tiểu) vậy ta gọi global minimum là điểm mà tại đó hàm số đạt giá trị nhỏ nhất.

1. Điểm local minimum x^* của hàm số là điểm có đạo hàm $f'(x^*) = 0$. Hơn nữa, đạo hàm các điểm bên trái của x^* là không dương, ngược lại với các điểm bên phải là không âm. Như vậy, càng xa về phía bên trái đạo hàm càng âm, càng xa về phía bên phải đạo hàm càng dương.

2. Đường tiếp tuyến với đồ thị hàm số đó tại 1 điểm bất kì có hệ số góc chính bằng đạo hàm của hàm số tại điểm đó.

Trong hình trên, các điểm bên trái của điểm local minimum màu xanh lục có đạo hàm âm, các điểm bên phải có đạo hàm dương. Và đối với hàm số này, càng xa về phía trái của điểm local minimum thì đạo hàm càng âm, càng xa về phía phải thì đạo hàm càng dương.

Giả sử x_t là điểm ta tìm được sau vòng lặp thứ t. Ta cần tìm một thuật toán để đưa x_t về càng gần x^*

càng tốt.

Trong hình đầu tiên, chúng ta lại có thêm hai quan sát nữa:

1. Nếu đạo hàm của hàm số tại x_t: $f'(x_t) > 0$ thì x_t nằm về bên phải so với x^* (và ngược lại). Để điểm tiếp theo x_{t+1} gần với x^* hơn, chúng ta cần di chuyển x_t về phía bên trái, tức về phía *âm*. Nói cách khác, **chúng ta cần di chuyển ngược dấu với đạo hàm**:
$$x_{t+1} = x_t + \Delta$$
Trong đó Δ là một đại lượng ngược dấu với đạo hàm $f'(x_t)$
2. x_t càng xa x^* về phía bên phải thì $f'(x_t)$ càng lớn hơn 0 (và ngược lại). Vậy, lượng di chuyển Δ, một cách trực quan nhất, là tỉ lệ thuận với $-f'(x_t)$

Hai nhận xét phía trên cho chúng ta một cách cập nhật đơn giản là:

$$x_{t+1} = x_t - \eta f'(x_t)$$

Trong đó $\boldsymbol{\eta}$ (đọc là *eta*) là một số dương được gọi là *learning rate* (tốc độ học). Dấu trừ thể hiện việc chúng ta phải *đi ngược* với đạo hàm. Các quan sát đơn giản phía trên, mặc dù không phải đúng cho tất cả các bài toán, là nền tảng cho rất nhiều phương pháp tối ưu nói chung và thuật toán Machine Learning nói riêng.

2.1 Ví dụ đơn giản với Python

Xét hàm số $f(x) = x_2 + 5sin(x)$

với đạo hàm $f'(x) = 2x + 5cos(x)$ (hàm này được chọn vì nó không dễ tìm nghiệm của đạo hàm bằng 0 như hàm phía trên). Giả sử bắt đầu từ một điểm $\boldsymbol{x_0}$ nào đó, tại vòng lặp thứ t, ta sẽ cập nhật như sau:

$$x_{t+1} = x_t - \eta\big(2x_t + 5cos(x_t)\big)$$

Ta khai báo các thư viện:

```
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals
import math
import numpy as np
import matplotlib.pyplot as plt
```

Tiếp theo, ta viết các hàm số :

1. grad để tính đạo hàm

2. cost để tính giá trị của hàm số. Hàm này không sử dụng trong thuật toán nhưng thường được dùng để kiểm tra việc tính đạo hàm của đúng không hoặc để xem giá trị của hàm số có giảm theo mỗi vòng lặp hay không.
3. myGD1 là phần chính thực hiện thuật toán Gradient Desent nêu phía trên. Đầu vào của hàm số này là learning rate và điểm bắt đầu. Thuật toán dừng lại khi đạo hàm có độ lớn đủ nhỏ.

```
def grad(x):
    return 2*x+ 5*np.cos(x)

def cost(x):
    return x**2 + 5*np.sin(x)

def myGD1(eta, x0):
    x = [x0]
    for it in range(100):
        x_new = x[-1] - eta*grad(x[-1])
        if abs(grad(x_new)) < 1e-3:
            break
        x.append(x_new)
    return (x, it)
```

2.2 Điểm khởi tạo khác nhau

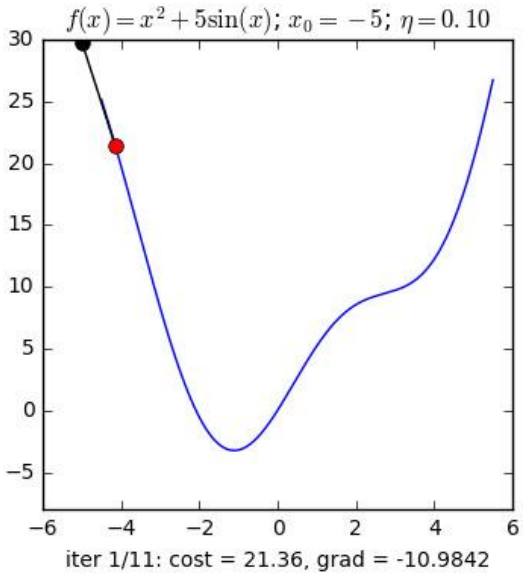
Sau khi có các hàm cần thiết, ta thử tìm nghiệm với các điểm khởi tạo khác nhau là $x_0 = -5$ và $x_0 = 5$.

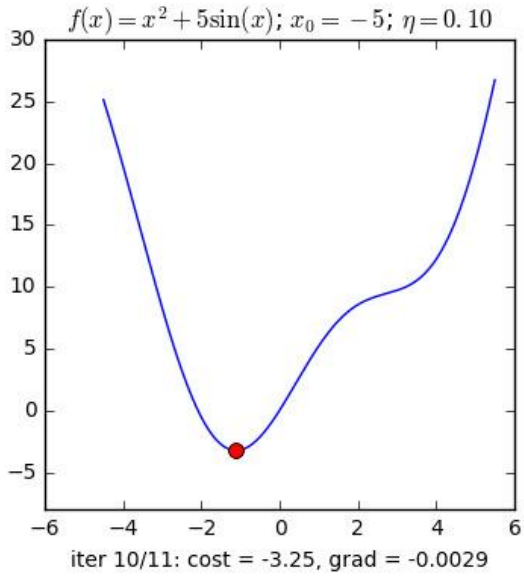
```
(x1, it1) = myGD1(.1, -5)
(x2, it2) = myGD1(.1, 5)
print('Solution x1 = %f, cost = %f, obtained after %d iterations'%(x1[-1], cost(x1[-1]), it1))
print('Solution x2 = %f, cost = %f, obtained after %d iterations'%(x2[-1], cost(x2[-1]), it2))
```

```
Solution x1 = -1.110667, cost = -3.246394, obtained after 11 iterations
Solution x2 = -1.110341, cost = -3.246394, obtained after 29 iterations
```

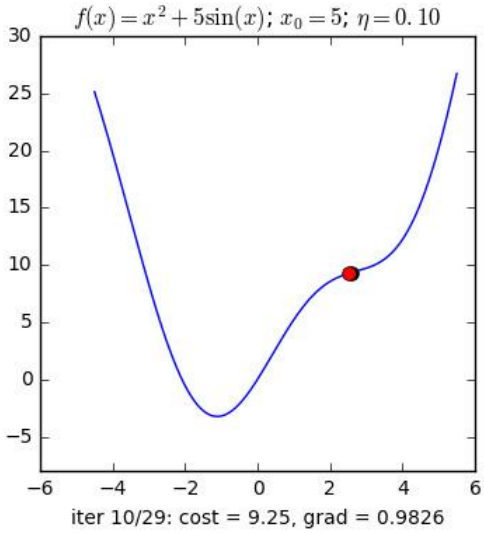
Vậy là với các điểm ban đầu khác nhau, thuật toán của chúng ta tìm được nghiệm gần giống nhau, mặc dù với tốc độ hội tụ khác nhau. Dưới đây là hình ảnh minh họa thuật toán GD cho bài toán này:

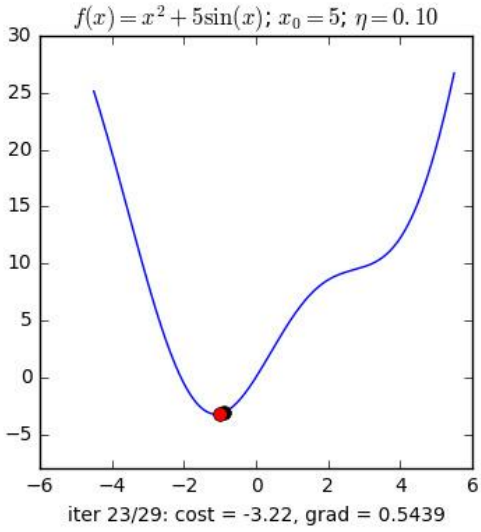
- Với $x_0 = -5$





- Với $x_0 = 5$



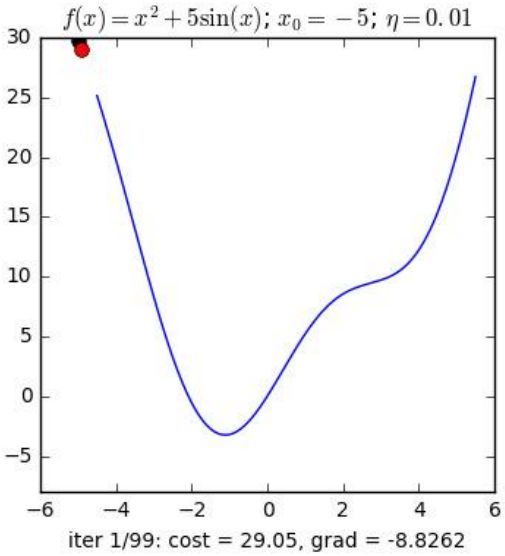


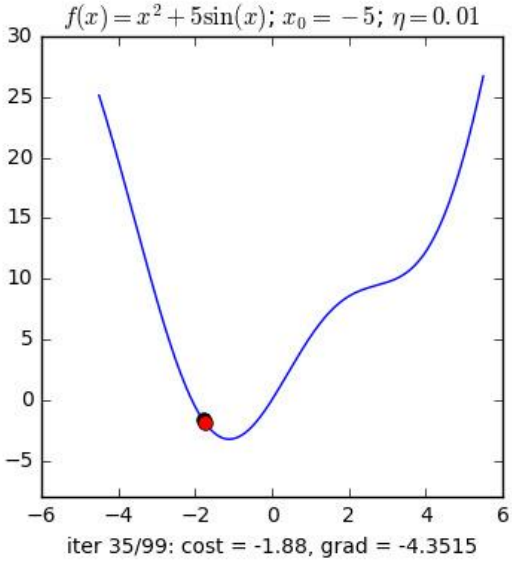
Từ hình minh họa trên ta thấy rằng ở hình trên, tương ứng với $x_0 = -5$, nghiệm hội tụ nhanh hơn, vì điểm ban đầu x0 gần với nghiệm $x^* \approx -1$ hơn. Hơn nữa, với $x_0 = 5$ ở hình dưới, *đường đi* của nghiệm có chứa một khu vực có đạo hàm khá nhỏ gần điểm có hoành độ bằng 2. Điều này khiến cho thuật toán *la cà* ở đây khá lâu. Khi vượt qua được điểm này thì mọi việc diễn ra rất tốt đẹp.

2.3 Learning rate khác nhau

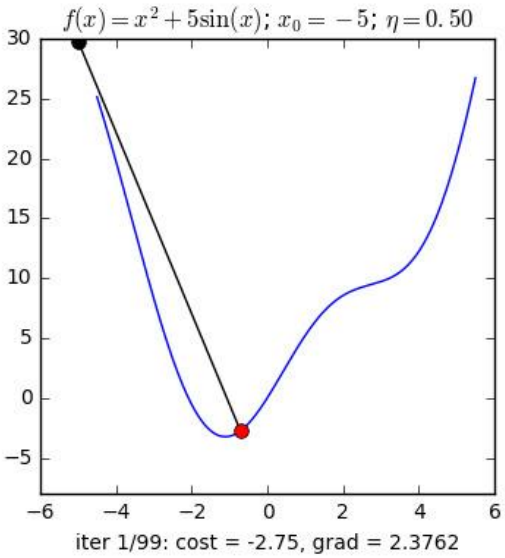
Tốc độ hội tụ của GD không những phụ thuộc vào điểm khởi tạo ban đầu mà còn phụ thuộc vào *learning rate*. Dưới đây là một ví dụ với cùng điểm khởi tạo x0=−5 nhưng learning rate khác nhau:

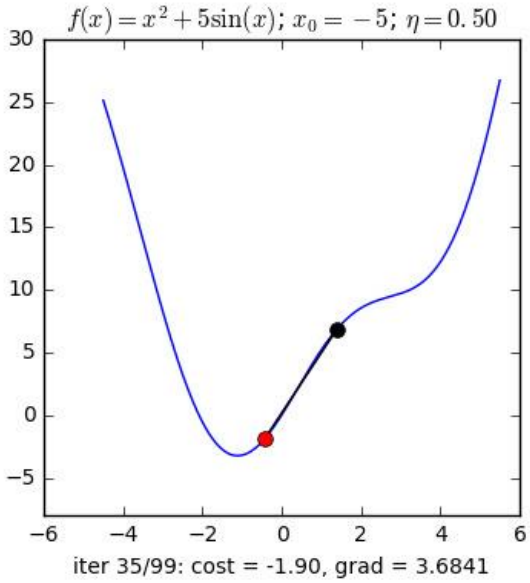
- Với $\eta = 0.01$





- Với $\eta = 0.50$





Ta quan sát thấy hai điều:

1. Với *learning rate* nhỏ $\eta = 0.01$ tốc độ hội tụ rất chậm. Trong ví dụ này có tối đa 100 vòng lặp nên thuật toán dừng lại trước khi tới *đích*, mặc dù đã rất gần. Trong thực tế, khi việc tính toán trở nên phức tạp, *learning rate* quá thấp sẽ ảnh hưởng tới tốc độ của thuật toán rất nhiều, thậm chí không bao giờ tới được đích.
2. Với *learning rate* lớn $\eta = 0.5$, thuật toán tiến rất nhanh tới *gần đích* sau vài vòng lặp. Tuy nhiên, thuật toán không hội tụ được vì *bước nhảy* quá lớn, khiến nó cứ *quẩn quanh* ở đích.

Việc lựa chọn *learning rate* rất quan trọng trong các bài toán thực tế. Việc lựa chọn giá trị này phụ thuộc nhiều vào từng bài toán và phải làm một vài thí nghiệm để chọn ra giá trị tốt nhất.

Ngoài ra, tùy vào một số bài toán, GD có thể làm việc hiệu quả hơn bằng cách chọn ra *learning rate* phù hợp hoặc chọn *learning rate* khác nhau ở mỗi vòng lặp.

3 Gradient Descent cho hàm nhiều biến

Giả sử ta cần tìm global minimum cho hàm $f(\theta)$trong đó θ (*theta*) là một vector, thường được dùng để ký hiệu tập hợp các tham số của một mô hình cần tối ưu (trong Linear Regression thì các tham số chính là hệ số w). Đạo hàm của hàm số đó tại một điểm θ bất kỳ được ký hiệu là $\nabla_\theta f(\theta)$(hình tam giác ngược đọc là *nabla*). Tương tự như hàm 1 biến, thuật toán GD cho hàm nhiều biến cũng bắt đầu bằng một điểm dự đoán θ_0, sau đó, ở vòng lặp thứ t, quy tắc cập nhật là:

$$\theta_{t+1} = \theta t - \eta \nabla_\theta f(\theta t)$$

Hoặc viết dưới dạng đơn giản hơn: $\theta = \theta - \eta \nabla_\theta f(\theta)$

Quy tắc cần nhớ: **luôn luôn đi ngược hướng với đạo hàm**.

3.1 Áp dụng vào bài toán Linear Regression

Ta thử tối ưu hàm mất mát của thuật toán Linear Regression bằng thuật toán GD.
Hàm mất mát của Linear Regression là:

$$\mathcal{L}(w) = \frac{1}{2N} ||y - \bar{X}w \|_2^2$$

Đạo hàm của hàm mất mát là:

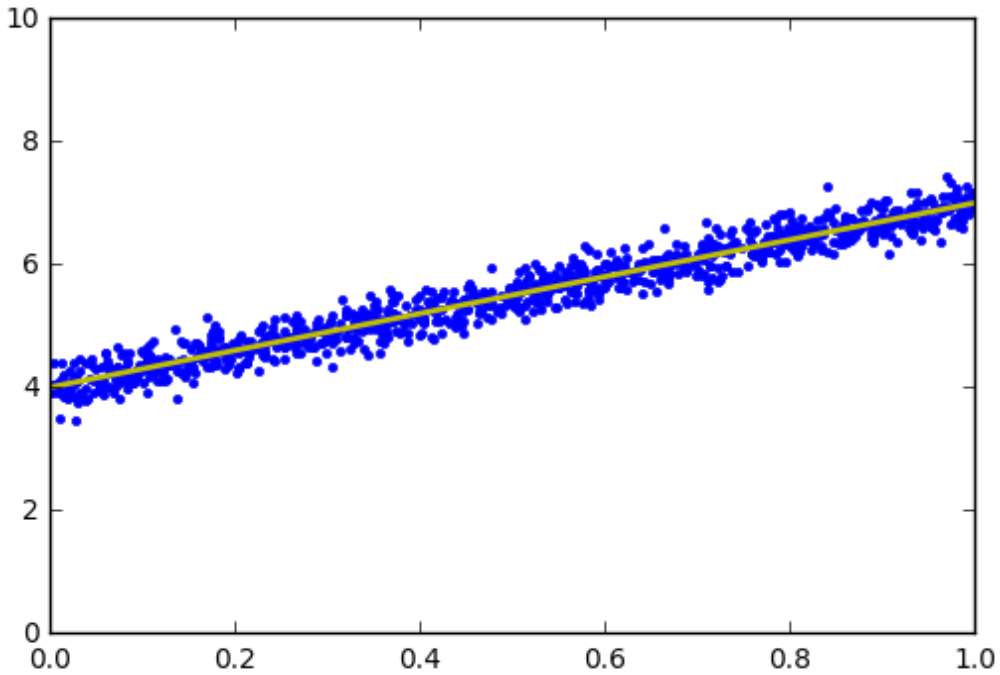
$$\nabla_w \mathcal{L}(w) = \frac{1}{N} \bar{X}^T \left(\overline{X}w - y\right) \quad (1)$$

3.2 Ví dụ trên Python

Khai báo thư viện, sau đó chúng ta tạo 1000 điểm dữ liệu được chọn *gần* với đường thẳng $y = 4 + 3x$, hiển thị chúng và tìm nghiệm theo công thức:

```python
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
np.random.seed(2)

X = np.random.rand(1000, 1)
y = 4 + 3 * X + .2*np.random.randn(1000, 1) # noise added

# Building Xbar
one = np.ones((X.shape[0],1))
Xbar = np.concatenate((one, X), axis = 1)

A = np.dot(Xbar.T, Xbar)
b = np.dot(Xbar.T, y)
w_lr = np.dot(np.linalg.pinv(A), b)
print('Solution found by formula: w = ',w_lr.T)

# Display result
w = w_lr
w_0 = w[0][0]
w_1 = w[1][0]
x0 = np.linspace(0, 1, 2, endpoint=True)
y0 = w_0 + w_1*x0

# Draw the fitting line
plt.plot(X.T, y.T, 'b.')     # data
plt.plot(x0, y0, 'y', linewidth = 2)   # the fitting line
plt.axis([0, 1, 0, 10])
plt.show()
```

```
Solution found by formula: w =  [[ 4.00305242  2.99862665]]
```



Đường thẳng tìm được là đường có màu vàng có phương trình $\boldsymbol{y \approx 4 + 2.998x}$
Tiếp theo ta viết đạo hàm và hàm mất mát:

```
def grad(w):
    N = Xbar.shape[0]
    return 1/N * Xbar.T.dot(Xbar.dot(w) - y)

def cost(w):
    N = Xbar.shape[0]
    return .5/N*np.linalg.norm(y - Xbar.dot(w), 2)**2;
```

3.3 Kiểm tra đạo hàm

Việc tính đạo hàm của hàm nhiều biến thông thường khá phức tạp và rất dễ mắc lỗi, nếu chúng ta tính sai đạo hàm thì thuật toán GD không thể chạy đúng được. Trong thực nghiệm, có một cách để kiểm tra liệu đạo hàm tính được có chính xác không. Cách này dựa trên định nghĩa của đạo hàm (cho hàm 1 biến):

$$f'(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

Một cách thường được sử dụng là lấy một giá trị ε rất nhỏ, ví dụ 10−6, và sử dụng công thức:

$$f'(x) \approx \frac{f(x+\varepsilon) - f(x-\varepsilon)}{2\varepsilon} \quad (2)$$

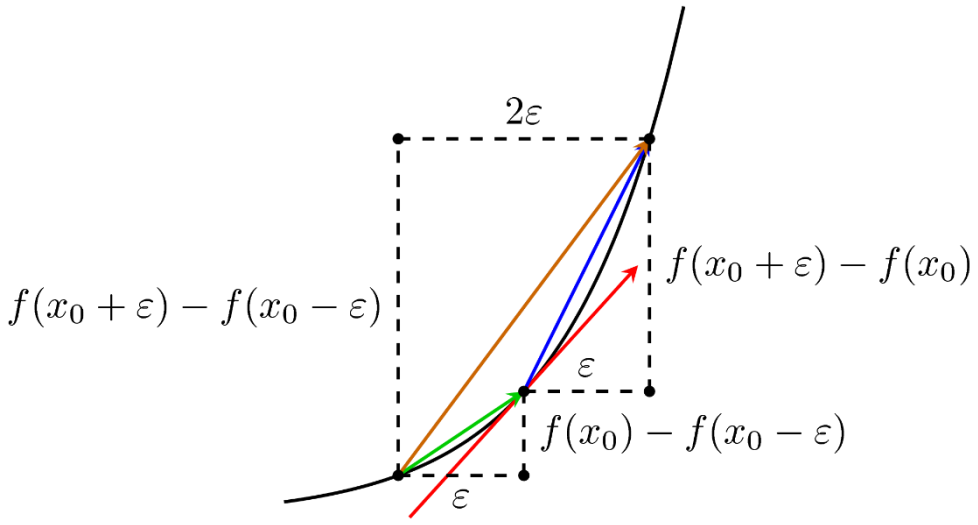
Cách tính này được gọi là *numerical gradient*.

Câu hỏi: Tại sao công thức xấp xỉ hai phía trên đây lại được sử dụng rộng rãi, sao không sử dụng công thức xấp xỉ đạo hàm bên phải hoặc bên trái?

Có hai các giải thích cho vấn đề này, một bằng hình học, một bằng giải tích.

- **Giải thích bằng hình học**

Quan sát hình dưới đây:



Trong hình, vector màu đỏ là đạo hàm *chính xác* của hàm số tại điểm có hoành độ bằng x_0. Vector màu xanh lam thể hiện cách xấp xỉ đạo hàm phía phải. Vector màu xanh lục thể hiện cách xấp xỉ đạo hàm phía trái. Vector màu nâu thể hiện cách xấp xỉ đạo hàm hai phía. Trong ba vector xấp xỉ đó, vector xấp xỉ hai phía màu nâu là gần với vector đỏ nhất nếu xét theo hướng.

Sự khác biệt giữa các cách xấp xỉ còn lớn hơn nữa nếu tại điểm x, hàm số bị *bẻ cong* mạnh hơn. Khi đó, xấp xỉ trái và phải sẽ khác nhau rất nhiều. Xấp xỉ hai bên sẽ *ổn định* hơn.

❖ **Giải thích bằng giải tích**

Sử dụng khai triển Taylor, với ε rất nhỏ, ta có hai xấp xỉ sau:

$$f(x+\varepsilon) \approx f(x) + f'(x)\varepsilon + \frac{f''(x)}{2}\varepsilon^2 + \cdots$$

và

$$f(x-\varepsilon) \approx f(x) - f'(x)\varepsilon - \frac{f''(x)}{2}\varepsilon^2 + \cdots$$

Từ đó ta có:

$$\frac{f(x+\varepsilon) - f(x)}{\varepsilon} \approx \frac{f'^{(x)} + f''(x)}{2}\varepsilon + \cdots = f'(x) + O(\varepsilon) \quad (3)$$

$$\frac{f(x+\varepsilon) - f(x-\varepsilon)}{2\varepsilon} \approx \frac{f'(x) + f^{(3)}(x)}{6}\varepsilon^2 + \cdots = f'(x) + O(\varepsilon^2) \quad (4)$$

Từ đó, nếu xấp xỉ đạo hàm bằng công thức (3) (xấp xỉ đạo hàm phải), sai số sẽ là $O(\varepsilon)$. Trong khi đó, nếu xấp xỉ đạo hàm bằng công thức (4) (xấp xỉ đạo hàm hai phía), sai số sẽ là $O(\varepsilon_2) \ll O(\varepsilon)$ nếu ε nhỏ.

Cả hai cách giải thích trên đây đều cho chúng ta thấy rằng, xấp xỉ đạo hàm hai phía là xấp xỉ tốt hơn.

❖ **Với hàm nhiều biến**

Với hàm nhiều biến, công thức (2) được áp dụng cho từng biến khi các biến khác cố định. Cách tính này thường cho giá trị khá chính xác. Tuy nhiên, cách này không được sử dụng để tính đạo hàm vì độ phức tạp quá cao so với cách tính trực tiếp. Khi so sánh đạo hàm này với đạo hàm chính xác tính theo công thức, người ta thường giảm số chiều dữ liệu và giảm số điểm dữ liệu để thuận tiện cho tính toán. Một khi đạo hàm tính được rất gần với *numerical gradient*, chúng ta có thể tự tin rằng đạo hàm tính được là chính xác.

Dưới đây là một đoạn code đơn giản để kiểm tra đạo hàm và có thể áp dụng với một hàm số (của một vector) bất kỳ với cost và grad đã tính ở phía trên.

```
def numerical_grad(w, cost):
    eps = 1e-4
    g = np.zeros_like(w)
    for i in range(len(w)):
        w_p = w.copy()
        w_n = w.copy()
        w_p[i] += eps
        w_n[i] -= eps
        g[i] = (cost(w_p) - cost(w_n))/(2*eps)
    return g

def check_grad(w, cost, grad):
    w = np.random.rand(w.shape[0], w.shape[1])
    grad1 = grad(w)
    grad2 = numerical_grad(w, cost)
    return True if np.linalg.norm(grad1 - grad2) < 1e-6 else False

print( 'Checking gradient...', check_grad(np.random.rand(2, 1), cost, grad))
```

```
Checking gradient... True
```

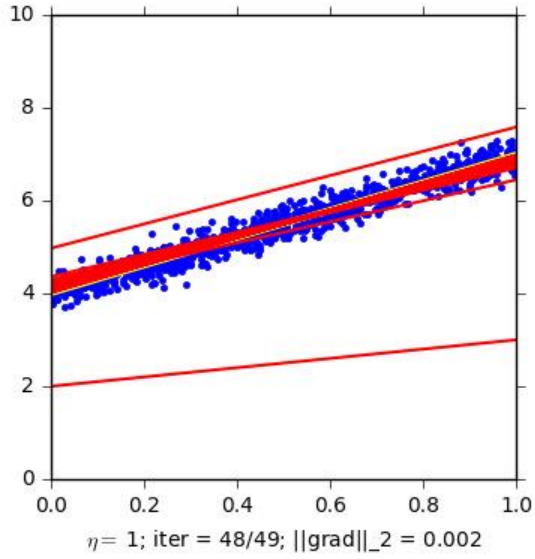
Với bài toán Linear Regression, cách tính đạo hàm như trong (1) phía trên được coi là đúng vì sai số giữa hai cách tính là rất nhỏ (nhỏ hơn 10^{-6}). Sau khi có được đạo hàm chính xác, chúng ta viết hàm cho GD:

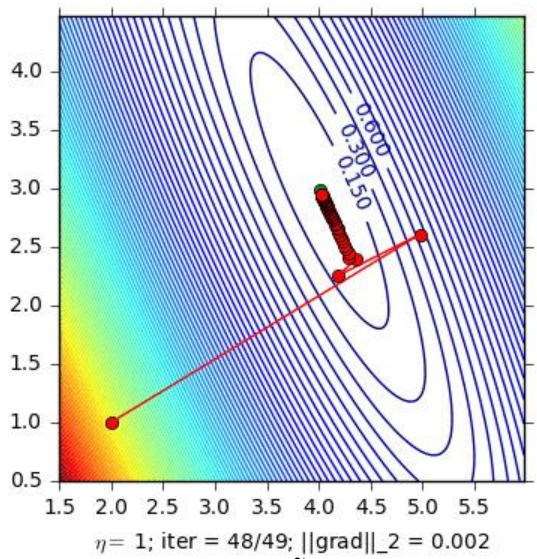
```
def myGD(w_init, grad, eta):
    w = [w_init]
    for it in range(100):
        w_new = w[-1] - eta*grad(w[-1])
        if np.linalg.norm(grad(w_new))/len(w_new) < 1e-3:
            break
        w.append(w_new)
    return (w, it)

w_init = np.array([[2], [1]])
(w1, it1) = myGD(w_init, grad, 1)
print('Solution found by GD: w = ', w1[-1].T, ',\nafter %d iterations.' %(it1+1))
```

```
Solution found by GD: w =  [[ 4.01780793  2.97133693]] ,
after 49 iterations.
```

Sau 49 vòng lặp, thuật toán đã hội tụ với một nghiệm khá gần với nghiệm tìm được theo công thức.

Sau đây là hình minh hoạ:





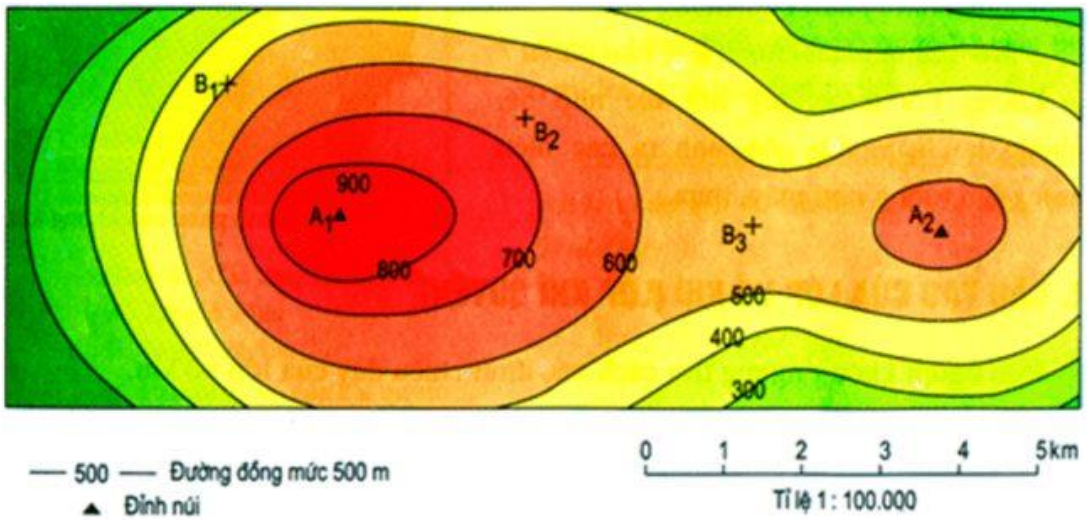
Trong hình bên trái, các đường thẳng màu đỏ là nghiệm tìm được sau mỗi vòng lặp.

Trong hình bên phải, tôi xin giới thiệu một thuật ngữ mới: *đường đồng mức*.

4 Đường đồng mức (level sets)

Với đồ thị của một hàm số với hai biến đầu vào cần được vẽ trong không gian ba chiều, nhều khi chúng ta khó nhìn được nghiệm có khoảng tọa độ bao nhiêu. Trong toán tối ưu, người ta thường dùng một cách vẽ sử dụng khái niệm *đường đồng mức* (level sets).

Trong các bản độ tự nhiên, để miêu tả độ cao của các dãy núi, người ta dùng nhiều đường cong kín bao quanh nhau như sau:



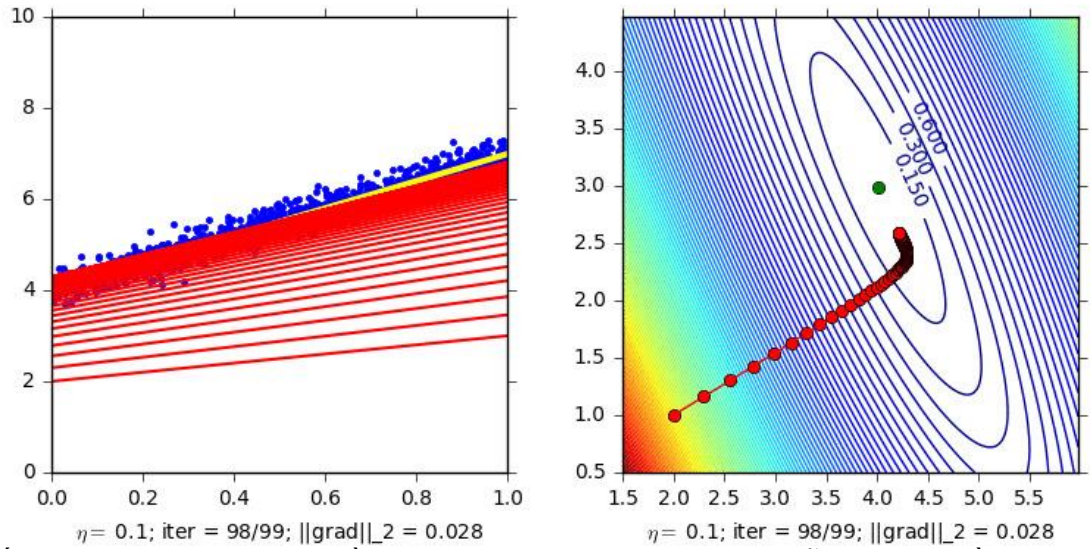
Ví dụ về đường đồng mức trong các bản đồ tự nhiên. (Nguồn: Địa lý 6: Đường đồng mức là những đường như thế nào?)

Các vòng nhỏ màu đỏ hơn thể hiện các điểm ở trên cao hơn.

Trong toán tối ưu, người ta cũng dùng phương pháp này để thể hiện các bề mặt trong không gian hai chiều.

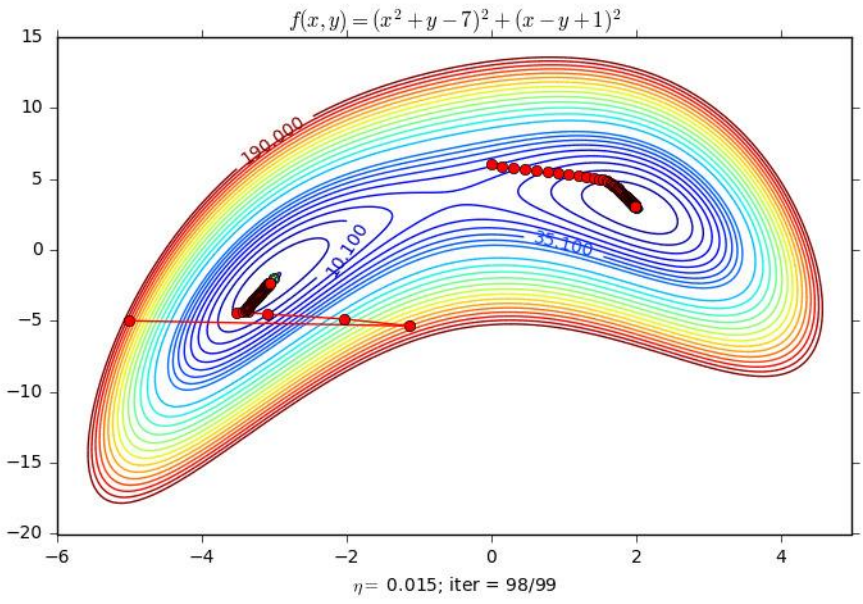
Quay trở lại với hình minh họa thuật toán GD cho bài toán Liner Regression bên trên, hình bên phải là hình biểu diễn các level sets. Tức là tại các điểm trên cùng một vòng, hàm mất mát có giá trị như nhau. Trong ví dụ này, giá trị của hàm số được hiển thị tại một số vòng. Các vòng màu xanh có giá trị thấp, các vòng tròn màu đỏ phía ngoài có giá trị cao hơn. Điểm này khác một chút so với đường đồng mức trong tự nhiên là các vòng bên trong thường thể hiện một thung lũng hơn là một đỉnh núi (vì chúng ta đang đi tìm giá trị nhỏ nhất).

Với *learning rate* nhỏ hơn, kết quả như sau:



Tốc độ hội tụ đã chậm đi nhiều, thậm chí sau 99 vòng lặp, GD vẫn chưa tới gần được nghiệm tốt nhất. Trong các bài toán thực tế, chúng ta cần nhiều vòng lặp hơn 99 rất nhiều, vì số chiều và số điểm dữ liệu thường là rất lớn.

Một ví dụ khác:



Hàm số $f(x,y) = (x^2 + y - 7)^2 + (x - y + 1)^2$ có hai điểm local minimum màu xanh lục tại (2,3) và (−3,−2), và chúng cũng là hai điểm global minimum. Trong ví dụ này, tùy vào điểm khởi tạo mà chúng ta thu được các nghiệm cuối cùng khác nhau.

5 Các thuật toán tối ưu Gradient Descent

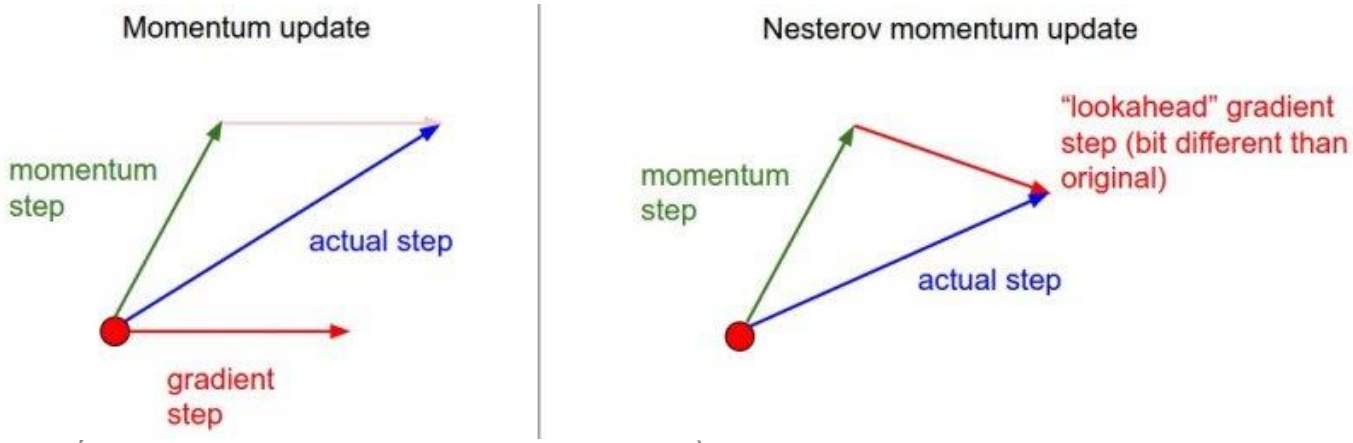
5.1 Momentum

5.2 Nesterov accelerated gradient (NAG)

Momentum giúp *hòn bi* vượt qua được *dốc local minimum* (cực tiểu), tuy nhiên, có một hạn chế chúng ta có thể thấy trong ví dụ trên: Khi tới gần *đích*, momemtum vẫn mất khá nhiều thời gian trước khi dừng lại. Lý do lại cũng chính là vì có *đà*. Có một phương pháp khác tiếp tục giúp khắc phục điều này, phương pháp đó mang tên Nesterov accelerated gradient (NAG), giúp cho thuật toán hội tụ nhanh hơn.

❖ **Ý tưởng chính**

Ý tưởng cơ bản là *dự đoán hướng đi trong tương lai*, tức nhìn trước một bước. Cụ thể, nếu sử dụng số hạng *momentum* γv_{t-1} để cập nhật thì ta có thể *xấp xỉ* được vị trí tiếp theo của hòn bi là $\theta - \gamma v_{t-1}$ (chúng ta không đính kèm phần gradient ở đây vì sẽ sử dụng nó trong bước cuối cùng). Vậy, thay vì sử dụng gradient của điểm hiện tại, NAG *đi trước một bước*, sử dụng gradient của điểm tiếp theo. Theo dõi hình dưới đây:



Ý tưởng của Nesterov accelerated gradient. Nguồn: CS231n Stanford: Convolutional Neural Networks for Visual Recognition)

- Với momentum thông thường: *lượng thay đổi* là tổng của hai vector: momentum vector và gradient ở thời điểm hiện tại.
- Với Nesterove momentum: *lượng thay đổi* là tổng của hai vector: momentum vector và gradient ở thời điểm được xấp xỉ là điểm tiếp theo.

❖ **Công thức cập nhật**

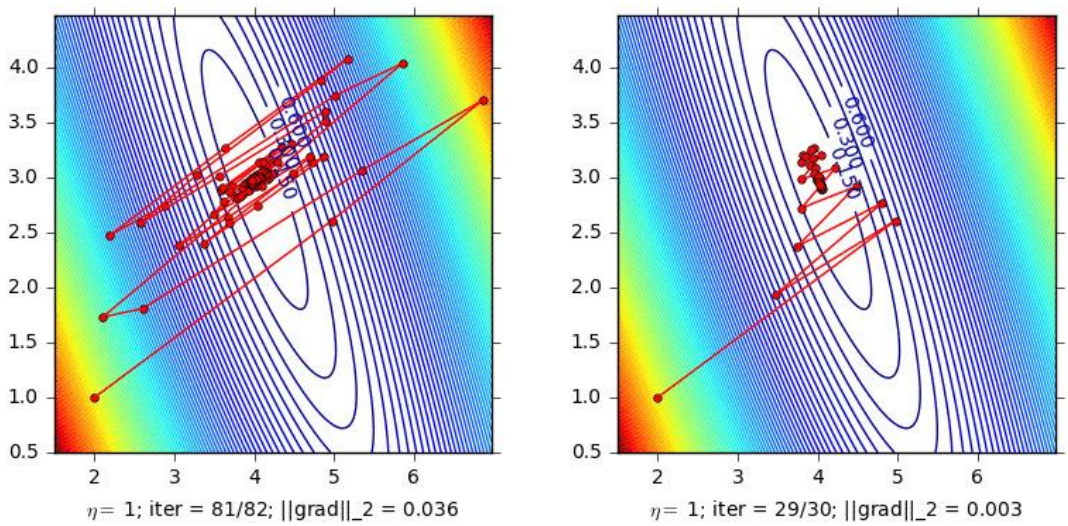
Công thức cập nhật của NAG được cho như sau:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})\ \theta = \theta - v_t$$

Ta có thể thấy điểm được tính đạo hàm đã thay đổi.

❖ **Ví dụ minh hoạ**

Dưới đây là ví dụ so sánh Momentum và NAG cho bài toán Linear Regression:



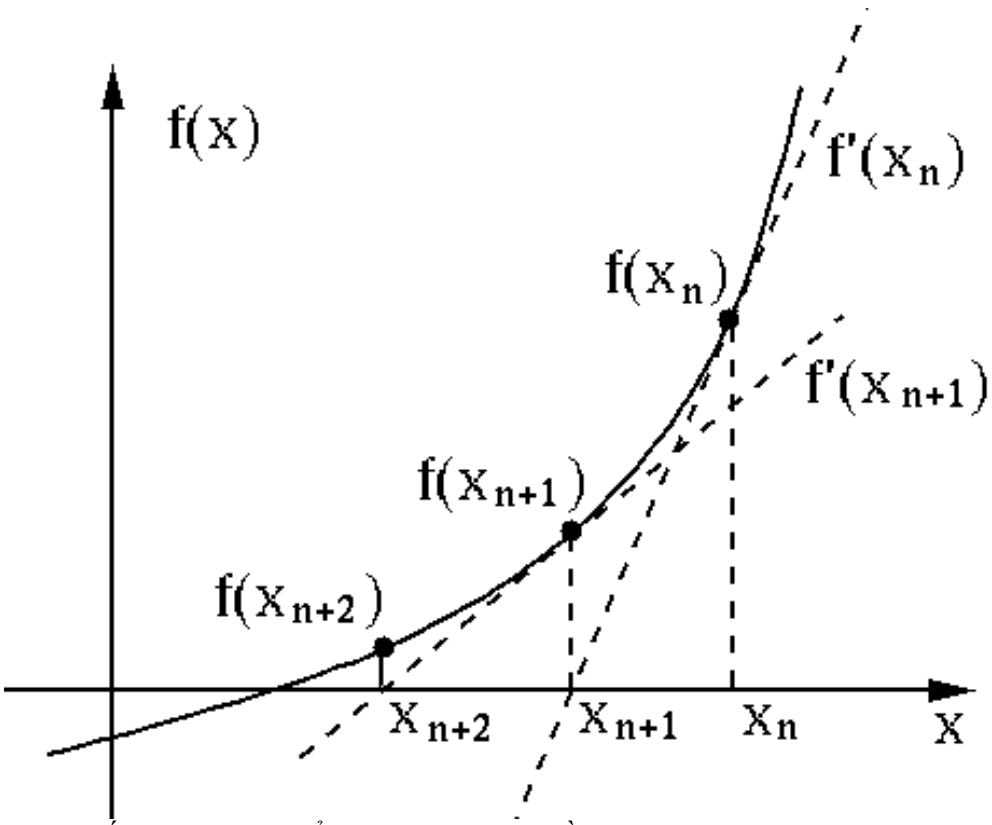
Hình bên trái là đường đi của nghiệm với phương pháp Momentum. nghiệm đi *zigzag* và mất nhiều vòng lặp hơn. Hình bên phải là đường đi của nghiệm với phương pháp NAG, nghiệm hội tụ nhanh hơn, và đường đi ít *zigzag* hơn.

5.3 Phương pháp tối ưu Newton

Các phương pháp GD đã trình bày còn được gọi là first-order method, vì lời giải tìm được dựa trên đạo hàm bậc nhất của hàm số. Phương pháp của Newton là một second-order method, tức lời giải yêu cầu tính đến đạo hàm bậc hai.

Nhắc lại, ta luôn giải phương trình đạo hàm của hàm mất mát bằng 0 để tìm các điểm local minimum. (Và trong nhiều trường hợp, coi nghiệm tìm được là nghiệm của bài toán tìm giá trị nhỏ nhất của hàm mất mát).

Phương pháp Newton là thuật toán giúp giải bài toán $f(x) = 0$.



Ý tưởng: Xuất phát từ một điểm x_0 được cho là gần với nghiệm x^*. Sau đó vẽ đường tiếp tuyến (mặt tiếp tuyến trong không gian nhiều chiều) với đồ thị hàm số $y = f(x)$ tại điểm trên đồ thị có hoành độ x0. Giao điểm x1 của đường tiếp tuyến này với trục hoành được xem là gần với nghiệm x^* hơn. Thuật toán lặp lại với điểm mới x_1 và cứ như vậy đến khi ta được $f(x_t) \approx 0$. Đó là ý nghĩa hình học của Newton's method, ta dựa vào đó lập phương trình tiếp tuyến với đồ thị của hàm f(x) tại điểm có hoành độ xt là:

$$y = f'(x_t)x - x_t + f(x_t)$$

Giao điểm của đường thẳng này với trục x tìm được bằng cách giải phương trình vế phải của biểu thức trên bằng 0, tức là:

$$x = x_t - \frac{f(x_t)}{f'(x_t)} \triangleq x_{t+1}$$

❖ **Newton's method trong bài toán tìm local minimum**

Áp dụng phương pháp này cho việc giải phương trình $f'(x) = 0$ ta có:

$$x_{t+1} = x_t - (f''(x_t) - 1f'(x_t))$$

Và trong không gian nhiều chiều với θ là biến:

$$\theta = \theta - \boldsymbol{H}\big(J(\theta)\big)^{-1}\nabla_\theta J(\theta)$$

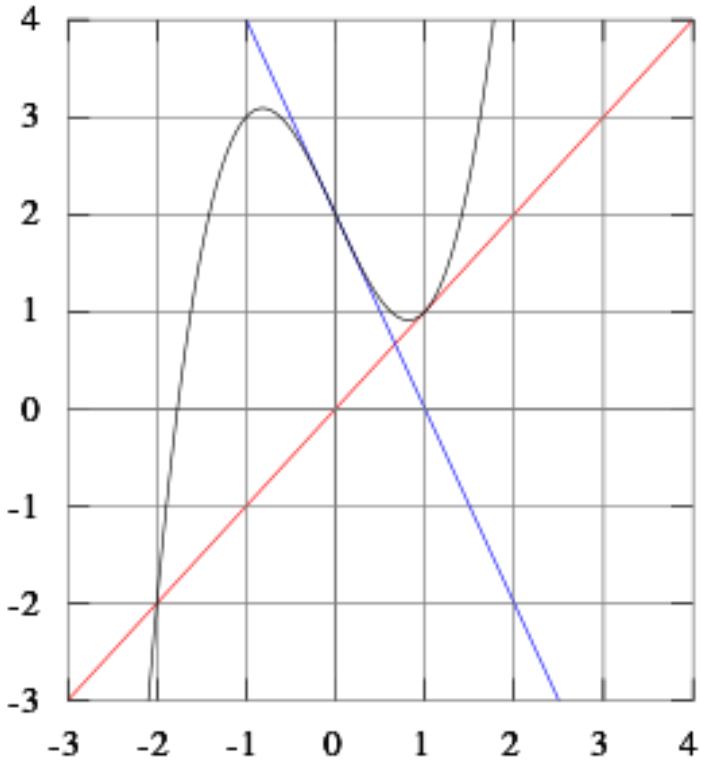
Trong đó $\boldsymbol{H}\big(J(\theta)\big)$ là đạo hàm bậc hai của hàm mất mát (còn gọi là Hessian matrix). Biểu thức này là một ma trận nếu θ là một vector. Và $\boldsymbol{H}\big(J(\theta)\big)^{-1}$ chính là nghịch đảo của ma trận đó.

❖ **Hạn chế của Newton's method**

Điểm khởi tạo phải *rất* gần với nghiệm x^*. Ý tưởng sâu xa hơn của Newton's method là dựa trên khai triển Taylor của hàm số $f(x)$ tới đạo hàm thứ nhất:

$$0 = f(x^*) \approx f(x_t) + f'(x_t)(x_t - x^*)$$

Từ đó suy ra: $x^* \approx x_t - \frac{f(x_t)}{f'(x_t)}$. Một điểm rất quan trọng, khai triển Taylor chỉ đúng nếu xt rất gần với x^*! Dưới đây là một ví dụ kinh điển về việc Newton's method cho một dãy số phân kỳ (divergent).



Nguồn: Wikipedia

Nghiệm là 1 điểm gần -2. Tiếp tuyến của đồ thị hàm số tại điểm có hoành độ bằng 0 cắt trục hoành tại 1 và ngược lại. Trong trường hợp này, Newton's Method không bao giờ hội tụ.

- Nhận thấy rằng trong việc giải phương trình $f(x) = 0$, chúng ta có đạo hàm ở mẫu số. Khi đạo hàm này gần với 0, ta sẽ được một đường thẳng song song hoặc gần song song với trục hoành. Ta sẽ hoặc không tìm được giao điểm, hoặc được một giao điểm ở vô cùng. Đặc biệt, khi nghiệm chính là điểm có đạo hàm bằng 0, thuật toán gần như sẽ không tìm được nghiệm.
- Khi áp dụng Newton's method cho bài toán tối ưu trong không gian nhiều chiều, chúng ta cần tính nghịch đảo của Hessian matrix. Khi số chiều và số điểm dữ liệu lớn, đạo hàm bậc hai của hàm mất mát sẽ là một ma trận rất lớn, ảnh hưởng tới cả memory và tốc độ tính toán của hệ thống.

Khi áp dụng Newton's method cho bài toán tối ưu trong không gian nhiều chiều, chúng ta cần tính nghịch đảo của Hessian matrix. Khi số chiều và số điểm dữ liệu lớn, đạo hàm bậc hai của hàm mất mát sẽ là một ma trận rất lớn, ảnh hưởng tới cả memory và tốc độ tính toán của hệ thống.

6 Biến thể của Gradient Descent

Lấy bài toán Linear Regression làm ví dụ. Hàm mất mát và đạo hàm của nó cho bài toán này lần lượt là

$$J(w) = \frac{1}{2N} \ ||Xw - y \ ||_2^2$$
$$= \frac{1}{2N} \sum_{i=1}^{N} (x_i w - y_i)^2$$

và

$$\nabla_w J(w) = \frac{1}{N} \sum_{i=1}^{N} x_i^T \ (x_i w - y_i)$$

6.1 Batch Gradient Descent

Thuật toán Gradient Descent được đề cập trong bài còn được gọi là Batch Gradient Descent. Batch ở đây được hiểu là tất cả, tức khi cập nhật $\theta = w$, chúng ta sử dụng tất cả các điểm dữ liệu x_i.

Cách làm này có một vài hạn chế đối với cơ sở dữ liệu có vô cùng nhiều điểm (hơn 1 tỉ người dùng của Facebook chẳng hạn). Việc phải tính toán lại đạo hàm với tất cả các điểm này sau mỗi vòng lặp trở nên cồng kềnh và không hiệu quả. Thêm nữa, thuật toán này không hiệu quả với online learning.

Online learning là khi cơ sở dữ liệu được cập nhật liên tục (chẳng hạn thêm người dùng đăng ký hàng ngày), mỗi lần thêm vài điểm dữ liệu mới. Kéo theo đó là mô hình của chúng ta cũng phải thay đổi một chút để phù hợp với các dữ liệu mới này. Nếu làm theo Batch Gradient Descent, tức tính lại đạo hàm của hàm mất mát tại tất cả các điểm dữ liệu, thì thời gian tính toán sẽ rất lâu, và thuật toán của chúng ta coi như không online nữa do mất quá nhiều thời gian tính toán.

Trên thực tế, có một thuật toán đơn giản hơn và tỏ ra rất hiệu quả, có tên gọi là Stochastic Gradient Descent (SGD).

6.2 Stochastic Gradient Descent

Trong thuật toán này, tại 1 thời điểm, ta chỉ tính đạo hàm của hàm mất mát dựa trên *chỉ một* điểm dữ liệu x_i rồi cập nhật θ dựa trên đạo hàm này. Việc này được thực hiện với từng điểm trên toàn bộ dữ liệu, sau đó lặp lại quá trình trên. Thuật toán rất đơn giản này trên thực tế lại làm việc rất hiệu quả.

Mỗi lần duyệt một lượt qua *tất cả* các điểm trên toàn bộ dữ liệu được gọi là một epoch. Với GD thông thường thì mỗi epoch ứng với 1 lần cập nhật θ, với SGD thì mỗi epoch ứng với N lần cập nhật θ với N là số điểm dữ liệu. Nhìn vào một mặt, việc cập nhật từng điểm một như thế này có thể làm giảm đi tốc độ thực hiện 1 epoch. Nhưng mặt khác, SGD chỉ yêu cầu một

lượng epoch rất nhỏ (thường là 10 cho lần đầu tiên, sau đó khi có dữ liệu mới thì chỉ cần chạy dưới một epoch là đã có nghiệm tốt). Vì vậy SGD phù hợp với các bài toán có lượng cơ sở dữ liệu lớn (chủ yếu là Deep Learning mà chúng ta sẽ thấy trong phần sau của blog) và các bài toán yêu cầu mô hình thay đổi liên tục, tức online learning.

Thứ tự lựa chọn điểm dữ liệu

Một điểm cần lưu ý đó là: sau mỗi epoch, chúng ta cần shuffle (xáo trộn) thứ tự của các dữ liệu để đảm bảo tính ngẫu nhiên. Việc này cũng ảnh hưởng tới hiệu năng của SGD.

Một cách toán học, quy tắc cập nhật của SGD là:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x_i; y_i)$$

trong đó $J(\theta; x_i; y_i)$ là hàm mất mát với chỉ 1 cặp điểm dữ liệu (input, label) là $(x_i; y_i)$

Chú ý: chúng ta hoàn toàn có thể áp dụng các thuật toán tăng tốc GD như Momentum, AdaGrad,… vào SGD.

Ví dụ với bài toán Linear Regression

Với bài toán Linear Regression, $\theta = w$, hàm mất mát tại một điểm dữ liệu là:

$$J(w; x_i; y_i) = \frac{1}{2}(x_i w - y_i)^2$$

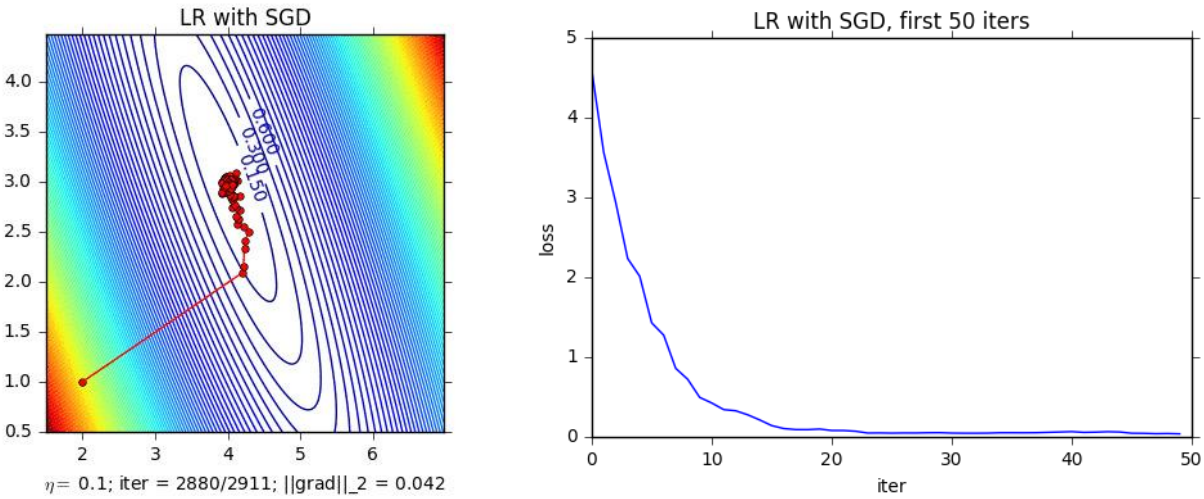
Đạo hàm theo w tương ứng là:

$$\nabla_w J(w; x_i; y_i) = x_i^T (x_i w - y_i)$$

Dưới đây là hàm số trong python để giải Linear Regression theo SGD:

```python
# single point gradient
def sgrad(w, i, rd_id):
    true_i = rd_id[i]
    xi = Xbar[true_i, :]
    yi = y[true_i]
    a = np.dot(xi, w) - yi
    return (xi*a).reshape(2, 1)

def SGD(w_init, grad, eta):
    w = [w_init]
    w_last_check = w_init
    iter_check_w = 10
    N = X.shape[0]
    count = 0
    for it in range(10):
        # shuffle data
        rd_id = np.random.permutation(N)
        for i in range(N):
            count += 1
            g = sgrad(w[-1], i, rd_id)
            w_new = w[-1] - eta*g
            w.append(w_new)
            if count%iter_check_w == 0:
                w_this_check = w_new
                if np.linalg.norm(w_this_check - w_last_check)/len(w_init) < 1e-3:
                    return w
                w_last_check = w_this_check
    return w
```

Kết quả được cho như hình:



Trái: đường đi của nghiệm với SGD. Phải: giá trị của loss function tại 50 vòng lặp đầu tiên.

Hình bên trái mô tả đường đi của nghiệm. Chúng ta thấy rằng đường đi khá là *zigzag* chứ không *mượt* như khi sử dụng GD. Điều này là dễ hiểu vì một điểm dữ liệu không thể đại diện cho toàn bộ dữ liệu được. Tuy nhiên, chúng ta cũng thấy rằng thuật toán hội tụ khá nhanh đến vùng lân cận của nghiệm. Với 1000 điểm dữ liệu, SGD chỉ cần gần 3 epoches (2911 tương ứng với 2911 lần cập nhật, mỗi lần lấy 1 điểm). Nếu so với con số 49 vòng lặp (epoches) như kết quả tốt nhất có được bằng GD, thì kết quả này lợi hơn rất nhiều.

Hình bên phải mô tả hàm mất mát cho toàn bộ dữ liệu sau khi *chỉ* sử dụng 50 điểm dữ liệu đầu tiên. Mặc dù không *mượt*, tốc độ hội tụ vẫn rất nhanh.

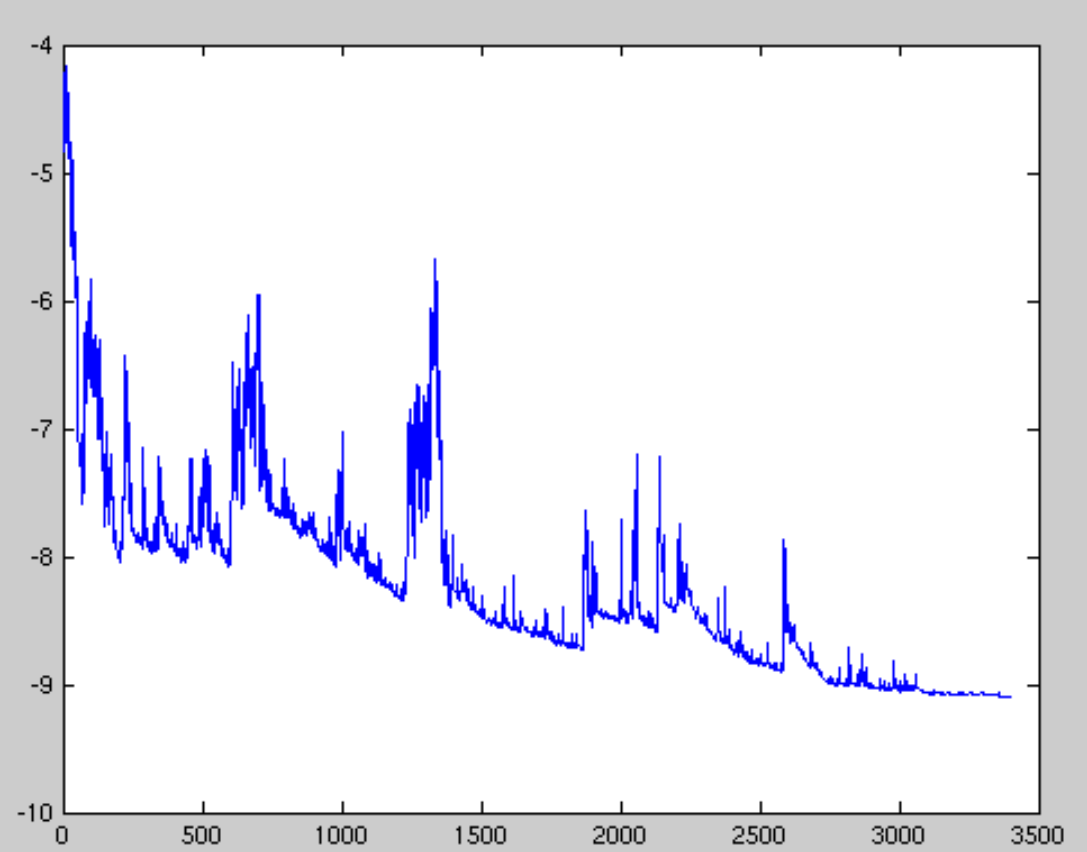
Thực tế cho thấy chỉ lấy khoảng 10 điểm là ta đã có thể xác định được gần đúng phương trình đường thẳng cần tìm rồi. Đây chính là ưu điểm của SGD - hội tụ rất nhanh.

6.3 Mini-batch Gradient Descent

Khác với SGD, mini-batch sử dụng một số lượng n lớn hơn 1 (nhưng vẫn nhỏ hơn tổng số dữ liệu Nrất nhiều). Giống với SGD, Mini-batch Gradient Descent bắt đầu mỗi epoch bằng việc xáo trộn ngẫu nhiên dữ liệu rồi chia toàn bộ dữ liệu thành các *mini-batch*, mỗi *mini-batch* có n điểm dữ liệu (trừ mini-batch cuối có thể có ít hơn nếu N không chia hết cho n). Mỗi lần cập nhật, thuật toán này lấy ra một mini-batch để tính toán đạo hàm rồi cập nhật. Công thức có thể viết dưới dạng:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x_{i:i+n}; y_{i:i+n})$$

Với $x_{i:i+n}$ được hiểu là dữ liệu từ thứ i tới thứ $i+n-1$ (theo ký hiệu của Python). Dữ liệu này sau mỗi epoch là khác nhau vì chúng cần được xáo trộn. Một lần nữa, các thuật toán khác cho GD như Momentum, Adagrad, Adadelta, … cũng có thể được áp dụng vào đây.
Mini-batch GD được sử dụng trong hầu hết các thuật toán Machine Learning, đặc biệt là trong Deep Learning. Giá trị n thường được chọn là khoảng từ 50 đến 100.
Dưới đây là ví dụ về giá trị của hàm mất mát mỗi khi cập nhật tham số θ
của một bài toán khác phức tạp hơn.



Hàm mất mát dao động (fluctuate) sau mỗi lần cập nhật nhưng nhìn chung giảm dần và có xu hướng hội tụ về cuối. (Nguồn: Wikipedia).

7 Stopping Criteria (điều kiện dừng)

Khi nào thì chúng ta biết thuật toán đã hội tụ và dừng lại?

Trong thực nghiệm, có một số phương pháp để làm thuật toán dừng lại như sau:

1. **Giới hạn số vòng lặp**: đây là phương pháp phổ biến nhất và cũng để đảm bảo rằng chương trình chạy không quá lâu. **Nhược điểm**: thuật toán có thể dừng lại trước khi đủ gần với nghiệm.
2. **So sánh gradient của nghiệm tại hai lần cập nhật liên tiếp**, khi nào giá trị này đủ nhỏ thì dừng lại. **Nhược điểm**: việc tính đạo hàm đôi khi trở nên quá phức tạp (ví dụ như khi có quá nhiều dữ liệu), nếu áp dụng phương pháp này thì ta không được lợi khi sử dụng SGD và mini-batch GD.
3. **So sánh giá trị của hàm mất mát của nghiệm tại hai lần cập nhật liên tiếp**, khi nào giá trị này đủ nhỏ thì dừng lại. Nhược điểm: nếu tại một thời điểm, đồ thị hàm số có dạng *bằng phẳng* tại một khu vực nhưng khu vực đó không chứa điểm local minimum (khu vực này thường được gọi là saddle points), thuật toán cũng dừng lại trước khi đạt giá trị mong muốn.
4. Trong SGD và mini-batch GD, cách thường dùng là **so sánh nghiệm sau một vài lần cập nhật**. Trong đoạn code Python phía trên về SGD đã áp dụng việc so sánh này mỗi khi nghiệm được cập nhật 10 lần. Việc làm này cũng tỏ ra khá hiệu quả.

8 Tài liệu tham khảo

[1] *Lemaréchal, C. (2012). "Cauchy and the Gradient Method" (PDF). Doc Math Extra: 251–254.*

[2] *Curry, Haskell B. (1944). "The Method of Steepest Descent for Non-linear Minimization Problems". Quart. Appl. Math.* ***2*** *(3): 258–261. doi:10.1090/qam/10667.*

[3] *Barzilai, Jonathan; Borwein, Jonathan M. (1988). "Two-Point Step Size Gradient Methods". IMA Journal of Numerical Analysis.* ***8*** *(1): 141–148. doi:10.1093/imanum/8.1.141.*

[4] *Fletcher, R. (2005). "On the Barzilai–Borwein Method". Trong Qi, L.; Teo, K.; Yang, X. (biên tập). Optimization and Control with Applications. Applied Optimization.* ***96****. Boston: Springer. 235–256. ISBN 0-387-24254-6.*

[5] *Boyd, Stephen; Vandenberghe, Lieven (2004). "Unconstrained Minimization" (PDF). Convex Optimization. New York: Cambridge University Press. 457–520. ISBN 0-521-83378-7.*

[6] *Chong, Edwin K. P.; Żak, Stanislaw H. (2013). "Gradient Methods". An Introduction to Optimization (Fourth ed.). Hoboken: Wiley. pp. 131–160. ISBN 978-1-118-27901-4.*

[7] *Himmelblau, David M. (1972). "Unconstrained Minimization Procedures Using Derivatives". Applied Nonlinear Programming. New York: McGraw-Hill. pp. 63–132. ISBN 0-07-028921-2.*

[8] *Ruder, S. (n.d.). "An overview of gradient descent optimization algorithms". arXiv.org. https://arxiv.org/abs/1609.04747*

[9] Nesterov, Y. (1983). "A method for unconstrained convex minimization problem with the rate of convergence o(1/k2)". Doklady ANSSSR (translated as Soviet.Math.Docl.), vol. 269, pp. 543– 547.

[10] Ketkar, N. (2017). "Stochastic gradient descent". In Apress eBooks (pp. 113–132). https://doi.org/10.1007/978-1-4842-2766-4_8

Các nguồn trên Internet

[1] http://sebastianruder.com/optimizing-gradient-descent/

[2] http://www.benfrederickson.com/numerical-optimization/

[3] https://www.youtube.com/watch?v=eikJboPQDT0

[4] https://machinelearningcoban.com/2017/01/12/gradientdescent/

[5] https://en.wikipedia.org/wiki/Gradient_descent

[6] https://en.wikipedia.org/wiki/Newton's_method

[7] http://sebastianruder.com/optimizing-gradientdescent/index.html#stochasticgradientdescent

[8] https://en.wikipedia.org/wiki/Stochastic_gradient_descent

[9] https://www.youtube.com/watch?v=UfNU3Vhv5CA