1. Giới thiệu:

Trong Machine Learning nói riêng và Toán Tối Ưu nói chung, chúng ta thường xuyên phải tìm giá trị nhỏ nhất (hoặc đôi khi là 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 của các hàm mất mát trong Machine Learning là rất phức tạp, thậm chí là bất khả thi. Thay vào đó, người ta thường cố gắng tìm các điểm local minimum, và ở một mức độ nào đó, coi đó là nghiệm cần tìm của bài toán.

Các điểm local minimum là nghiệm của phương trình đạo hàm bằng 0. Nếu bằng một cách nào đó có thể tìm được toàn bộ (hữu hạn) các điểm cực tiểu, ta chỉ cần thay từng điểm local minimum đó vào hàm số rồi tìm điểm làm cho hàm có giá trị nhỏ nhất (*đoạn này nghe rất quen thuộc, đúng không?*). Tuy nhiên, trong hầu hết các trường hợp, việc giải phương trình đạo hàm bằng 0 là bất khả thi. Nguyên nhân có thể đến từ sự phức tạp của dạng của 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 nhất 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 đến khi đạo hàm gần với 0. Gradient Descent (viết gọn là GD) và các biến thể của nó là một trong những phương pháp được dùng nhiều nhất.

Vì kiến thức về GD khá rộng nên tôi xin phép được chia thành hai phần. Phần 1 này giới thiệu ý tưởng phía sau thuật toán GD và một vài ví dụ đơn giản giúp các bạn làm quen với thuật toán này và vài khái niệm mới. Phần 2 sẽ nói về các phương pháp cải tiến GD và các biến thể của GD trong các bài toán mà số chiều và số điểm dữ liệu lớn. Những bài toán như vậy được gọi là *large-scale*.

2. Gradient Descent cho hàm 1 biến

Quay trở lại hình vẽ ban đầu và một vài quan sát tôi đã nêu. Giả sử xt là điểm ta tìm được sau vòng lặp thứ t. Ta cần tìm một thuật toán để đưa xt về càng gần $x*$ càng tốt.

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

- Nếu đạo hàm của hàm số tại xt: f′(xt)>0 thì xt nằm về bên phải so với x∗ (và ngược lại). Để điểm tiếp theo xt+1 gần với x∗ hơn, chúng ta cần di chuyển xt về phía bên trái, tức về phía *âm*. Nói các khác, **chúng ta cần di chuyển ngược dấu với đạo hàm**:xt+1=xt+ΔTrong đó Δ là một đại lượng ngược dấu với đạo hàm f′(xt).
- xt càng xa x∗ về phía bên phải thì f′(xt) càng lớn hơn 0 (và ngược lại). Vậy, lượng di chuyển Δ, một cách trực quan nhất, là tỉ lệ thuận với −f′(xt).

Hai nhận xét phía trên cho chúng ta một cách cập nhật đơn giản là: xt+1=xt−ηf′(xt)

Trong đó η (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 (Đây cũng chính là lý do phương pháp này được gọi là Gradient Descent - *descent* nghĩa là *đi ngược*). 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.

Ví dụ đơn giản với Python

Xét hàm số f(x)=x2+5sin(x) với đạo hàm f′(x)=2x+5cos(x) (một lý do tôi chọn hàm này vì nó không dễ tìm nghiệm của đạo hàm bằng 0 như hàm phía trên). Giả sử bắt đầu từ một điểm x0 nào đó, tại vòng lặp thứ t, chúng ta sẽ cập nhật như sau:xt+1=xt−η(2xt+5cos(xt))

Như thường lệ, tôi khai báo vài thư viện quen thuộc

```
# 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, tôi viết các hàm số :

- grad để tính đạo hàm

- 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.
- 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)
```

Điểm khởi tạo khác nhau

Sau khi có các hàm cần thiết, tôi thử tìm nghiệm với các điểm khởi tạo khác nhau là x0=−5 và x0=5.

```
(x1, it1) = myGD1(.1, -5)
(x2, it2) = myGD1(.1, 5)
```

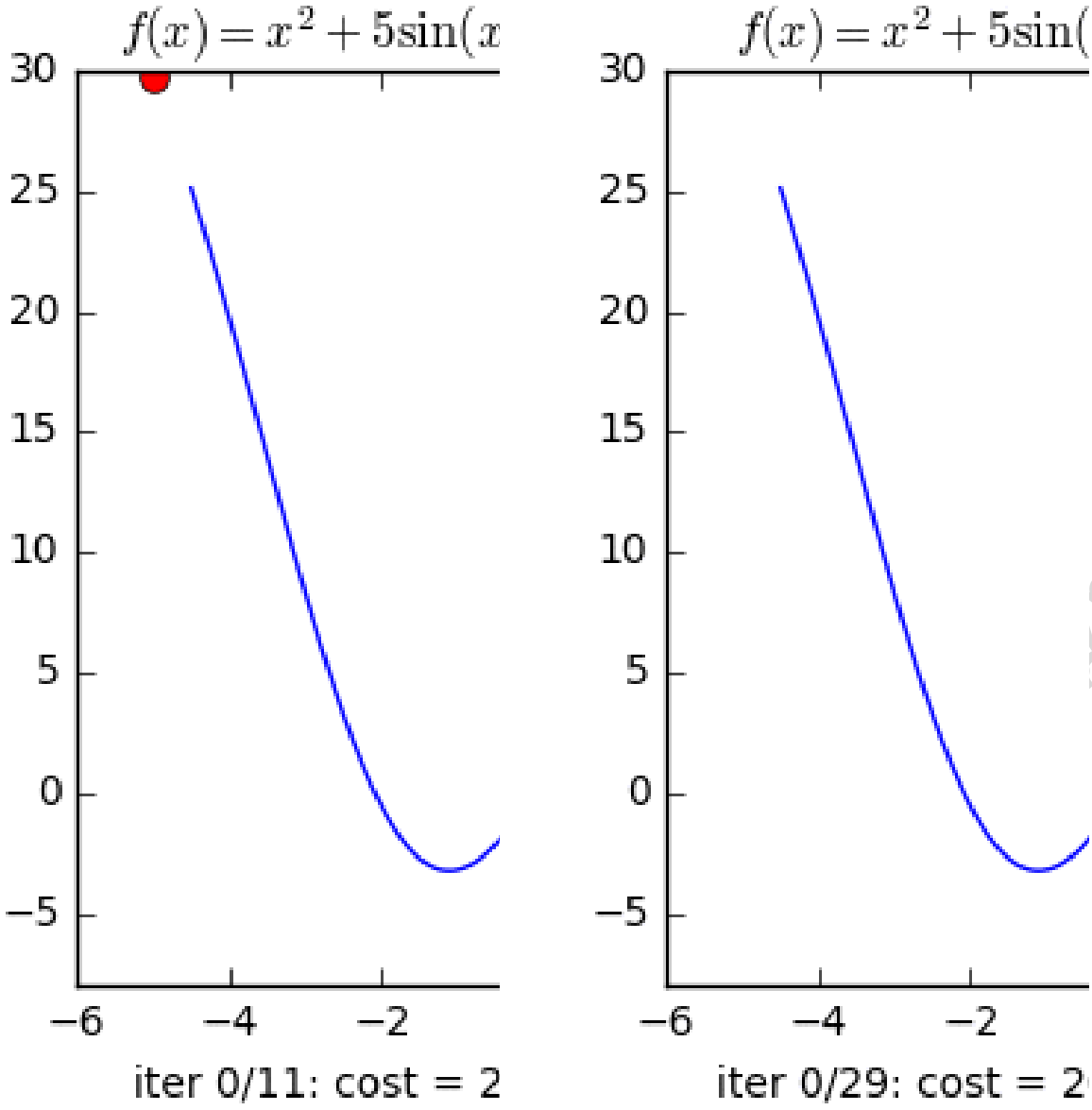
print('Solution x1 = %f, cost = %f, obtained after %d iterations'**%**(x1[-1], cost(x1[-1]), it1))

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

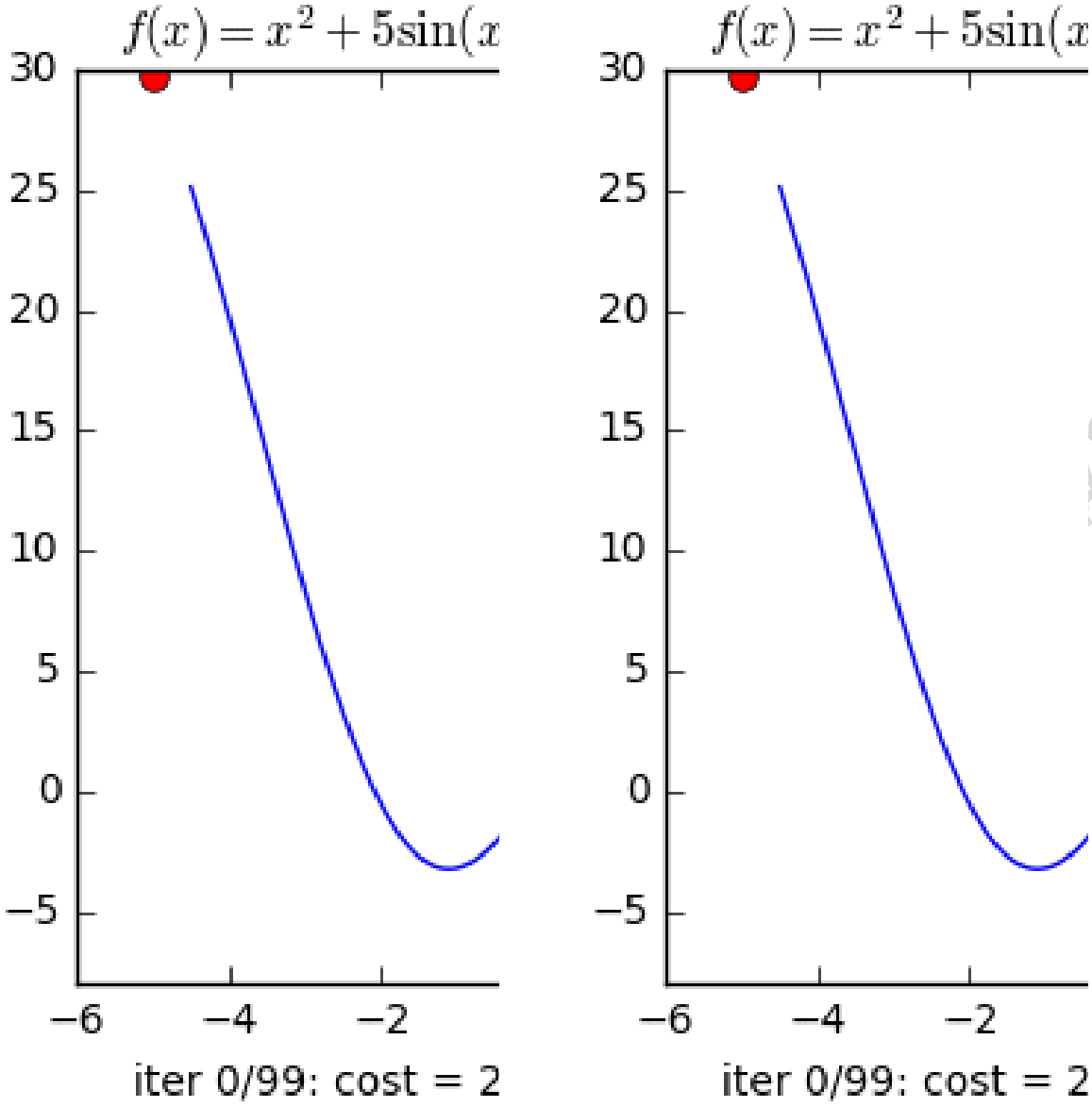
Vậy là với các điểm ban đầu khác nhau, thuật toán của chúng ta tìm được nghiệm gần giống nhau, mặc dù với tốc độ hội tụ khác nhau. Dưới đây là hình ảnh minh họa thuật toán GD cho bài toán này (*xem tốt trên Desktop ở chế độ full màn hình*).



Từ hình minh họa trên ta thấy rằng ở hình bên trái, tương ứng với x0=−5, nghiệm hội tụ nhanh hơn, vì điểm ban đầu x0 gần với nghiệm $x* \approx -1$ hơn. Hơn nữa, với x0=5 ở hình bên phải, *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.

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:



Ta quan sát thấy hai điều:

1. Với *learning rate* nhỏ η=0.01, tốc độ hội tụ rất chậm. Trong ví dụ này tôi chọn 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. Tôi sẽ quay lại vấn đề này ở phần 2.

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**.

Việc tính toán đạo hàm của các hàm nhiều biến là một kỹ năng cần thiết. Một vài đạo hàm đơn giản có thể được tìm thấy ở đây.

Quay lại với bài toán Linear Regression

Trong mục này, chúng ta quay lại với bài toán Linear Regression và thử tối ưu hàm mất mát của nó bằng thuật toán GD.

Hàm mất mát của Linear Regression là: $L(w) = \frac{1}{2N}||y - \bar{X}w||_2^2$

Chú ý: hàm này có khác một chút so với hàm tôi nêu trong bài Linear Regression. Mẫu số có thêm N là số lượng dữ liệu trong training set. Việc lấy trung bình cộng của lỗi này nhằm giúp tránh trường hợp hàm mất mát và đạo hàm có giá trị là một số rất lớn, ảnh hưởng tới độ chính xác của các phép toán khi thực hiện trên máy tính. Về mặt toán học, nghiệm của hai bài toán là như nhau.

Đạo hàm của hàm mất mát là:$\nabla_w L(w) = \frac{1}{N}\bar{X}^T(\bar{X}w - y)$ (1)

Sau đây là ví dụ trên Python và một vài lưu ý khi lập trình

Load thư viện

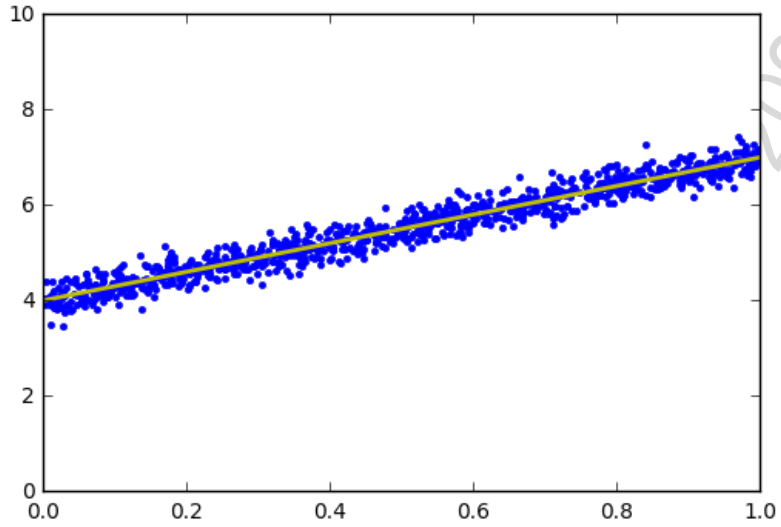
```
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
np.random.seed(2)
```

Tiếp theo, 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:

```
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 $y \approx 4 + 2.998x$.

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;
```

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

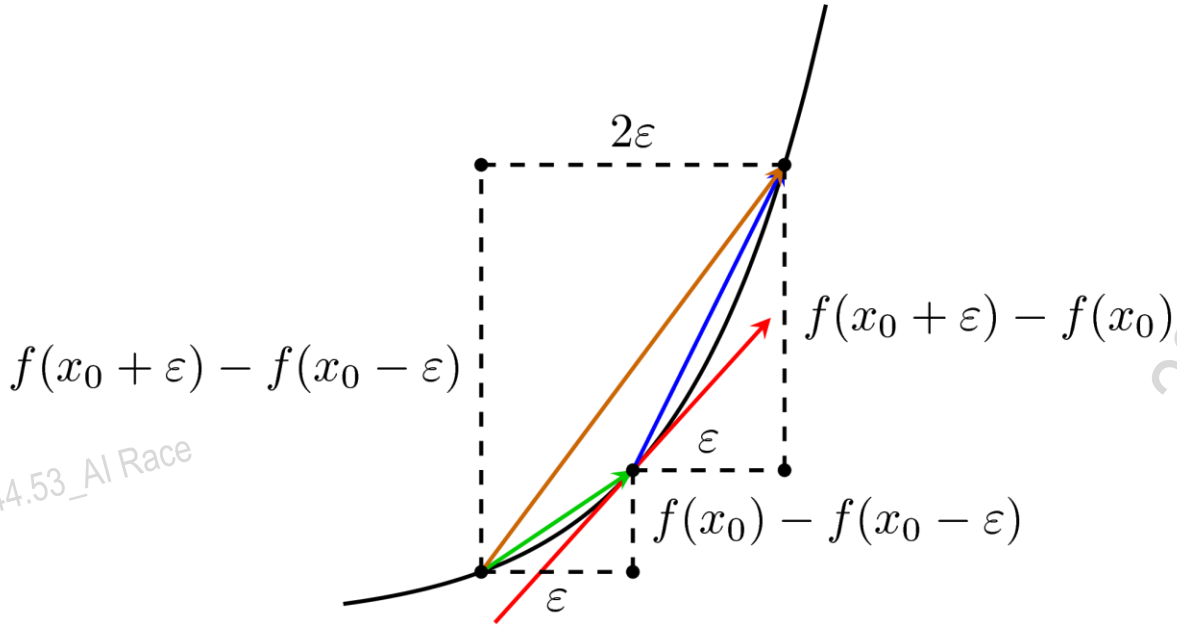
Cách tính này được gọi là *numerical gradient*.

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 x0. Vector màu xanh lam (có vẻ là hơi tím sau khi convert từ .pdf sang .png) 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

Chúng ta cùng quay lại một chút với Giải tích I năm thứ nhất đại học: Khai triển Taylor.

Với ε rất nhỏ, ta có hai xấp xỉ sau:

$f(x+\varepsilon) \approx f(x)+f'(x)\varepsilon+f''(x)2\varepsilon 2+\dots$

và: $f(x-\varepsilon) \approx f(x)-f'(x)\varepsilon+f''(x)2\varepsilon 2-\dots$

Từ đó ta có: $f(x+\varepsilon)-f(x)\varepsilon \approx f'(x)+f''(x)2\varepsilon+\dots=f'(x)+O(\varepsilon)$ (3)

$f(x+\varepsilon)-f(x-\varepsilon)2\varepsilon \approx f'(x)+f(3)(x)6\varepsilon 2+\dots=f'(x)+O(\varepsilon 2)$ (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 các hàm số khác, bạn đọc chỉ cần viết lại hàm grad và cost ở phần trên rồi áp dụng đoạn code này để kiểm tra đạo hàm. Nếu hàm số là hàm của một ma trận thì chúng ta thay đổi một chút trong hàm numerical_grad, tôi hy vọng không quá phức tạp*).

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)
```

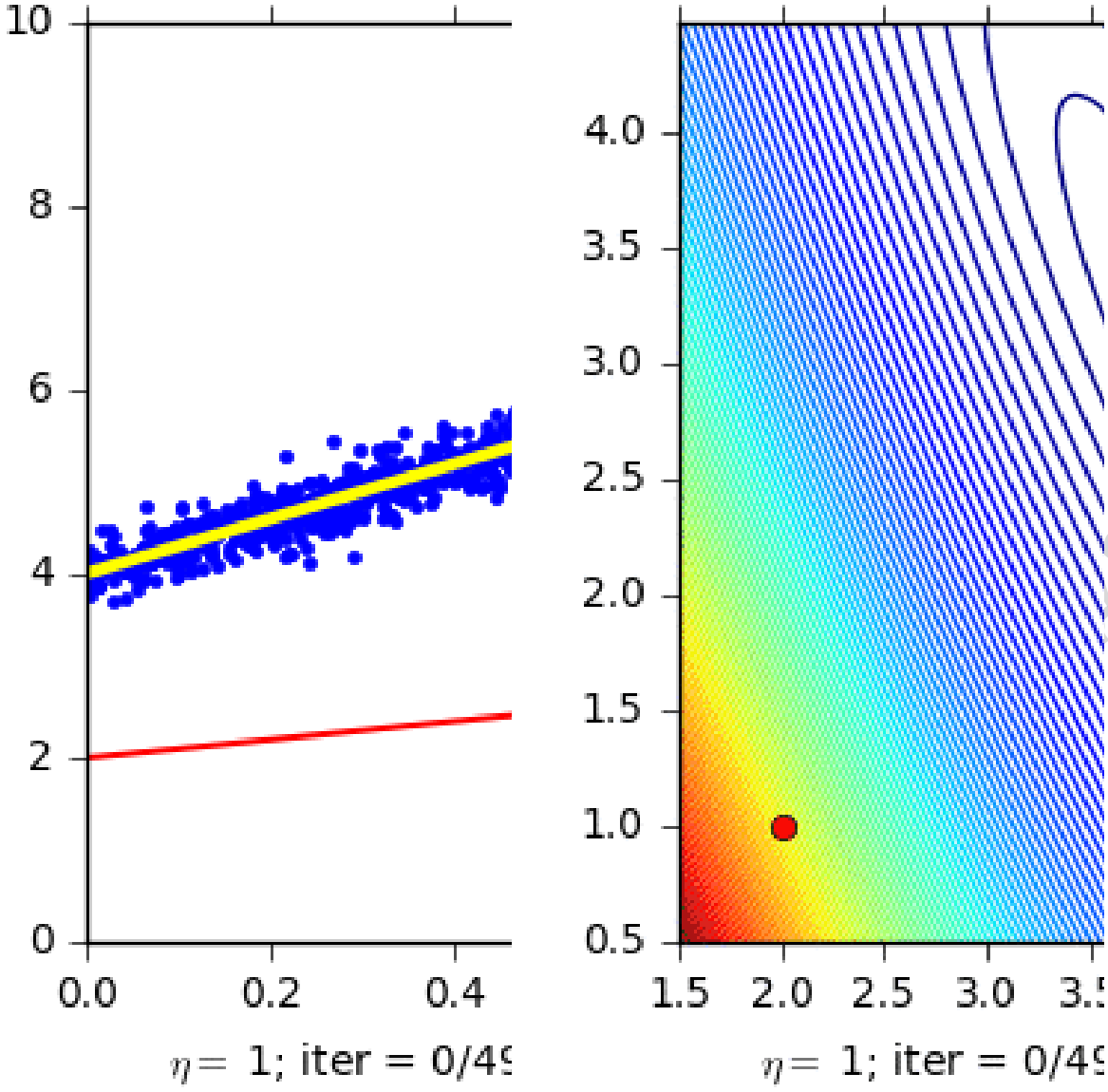
```
w_init = np.array([[2], [1]])
(w1, it1) = myGD(w_init, grad, 1)
print('Solution found by GD: w = ', w1[-1].T, ',\nafter %d iterations.' %(it1+1))
```

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.

Dưới đây là hình động minh họa thuật toán GD.



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*.

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

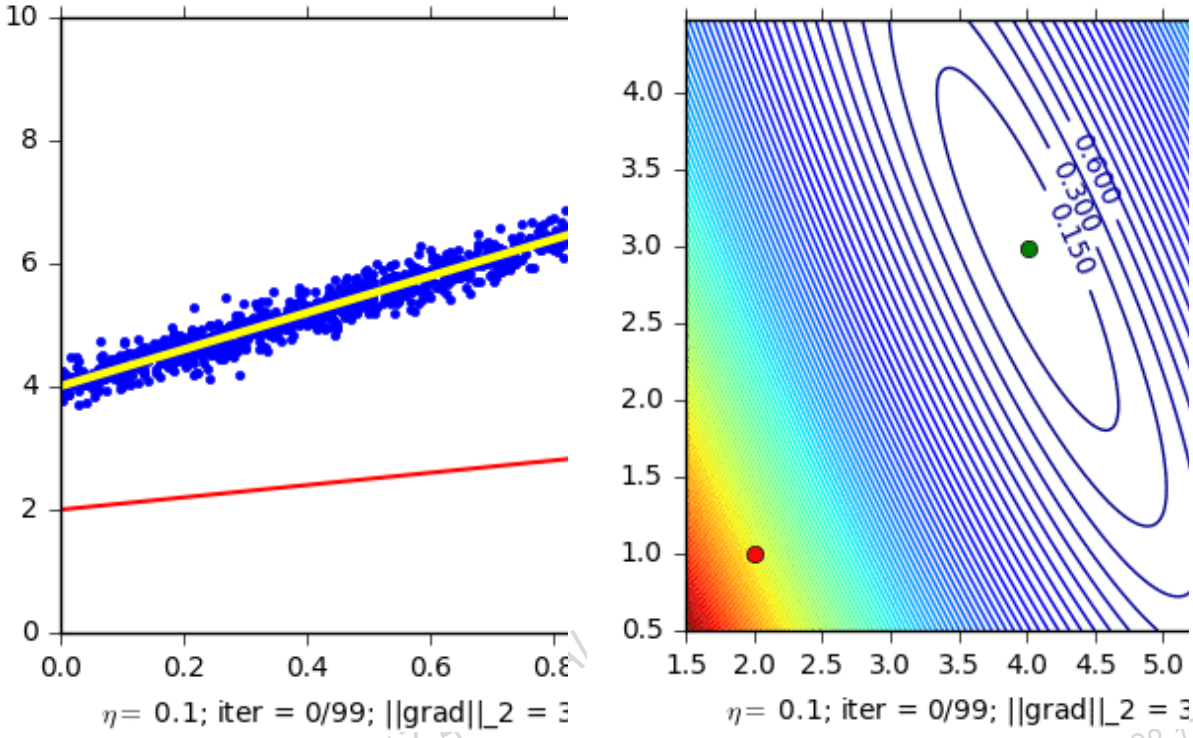
toán tối ưu, người ta thường dùng một cách vẽ sử dụng khái niệm *đường đồng mức* (level sets).

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, tôi hiển thị giá trị của hàm số 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).

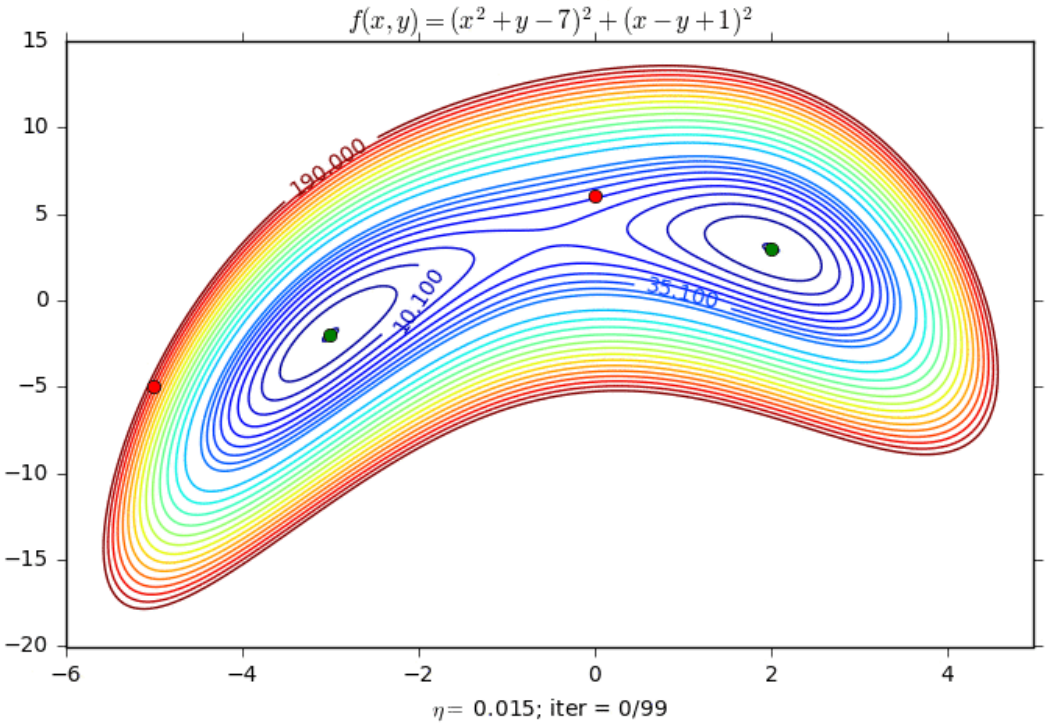
Tôi thử với *learning rate* nhỏ hơn, kết quả như sau:



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.

4. Một ví dụ khác

Để kết thúc phần 1 của Gradient Descent, tôi xin nêu thêm một ví dụ khác.



Hàm số f(x,y)=(x2+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. Thảo luận

Dựa trên GD, có rất nhiều thuật toán phức tạp và hiệu quả hơn được thiết kế cho những loại bài toán khác nhau. Vì bài này đã đủ dài, tôi xin phép dừng lại ở đây. Mời các bạn đón đọc bài Gradient Descent phần 2 với nhiều kỹ thuật nâng cao hơn.

6. Tài liệu tham khảo

1. An overview of gradient descent optimization algorithms
2. An Interactive Tutorial on Numerical Optimization
3. Gradient Descent by Andrew NG