

Vũ Hữu Tiệp

# Machine Learning cơ bản

Pre-order ebook tại <https://machinelearningcoban.com/ebook/>

Blog: <https://machinelearningcoban.com>

Facebook Page: <https://www.facebook.com/machinelearningbasicvn/>

Facebook Group: <https://www.facebook.com/groups/machinelearningcoban/>

Interactive Learning: <https://fundaml.com>

Last update:

December 17, 2017

---

# Contents

---

## Phần IV Neural Networks

---

<b>1</b>	<b>Gradient Descent</b>	4
1.1	Giới thiệu	4
1.2	Gradient Descent cho hàm một biến	5
1.3	Gradient Descent cho hàm nhiều biến	10
1.4	Gradient descent với momentum	13
1.5	Nesterov accelerated gradient	16
1.6	Stochastic Gradient Descent	18
1.7	Điều kiện dừng thuật toán	19
1.8	Đọc thêm	20
<b>2</b>	<b>Perceptron Learning Algorithm</b>	21
2.1	Giới thiệu	21
2.2	Thuật toán perceptron	22
2.3	Ví dụ và minh họa trên Python	25
2.4	Mô hình Neural Network đầu tiên	27
2.5	Thảo Luận	28

Contents	VI
<b>3 Logistic regression . . . . .</b>	30
3.1 Giới thiệu . . . . .	30
3.2 Hàm mất mát và phương pháp tối ưu . . . . .	32
3.3 Triển khai thuật toán trên Python . . . . .	35
3.4 Tính chất của logistic regression . . . . .	38
3.5 Bài toán phân biệt hai chữ số viết tay . . . . .	39
3.6 Bộ phân lớp nhị phân cho bài toán phân lớp đa lớp . . . . .	41
3.7 Thảo luận . . . . .	43
<b>4 Softmax Regression . . . . .</b>	45
4.1 Giới thiệu . . . . .	45
4.2 Softmax function . . . . .	46
4.3 Hàm mất mát và phương pháp tối ưu . . . . .	49
4.4 Ví dụ trên Python . . . . .	55
4.5 Thảo luận . . . . .	57
<b>5 Multilayer neural network và Backpropagation . . . . .</b>	58
5.1 Giới thiệu . . . . .	58
5.2 Các ký hiệu và khái niệm . . . . .	61
5.3 Activation function – Hàm kích hoạt . . . . .	63
5.4 Backpropagation . . . . .	65
5.5 Ví dụ trên Python . . . . .	69
5.6 Tránh overfitting cho neural network bằng weight decay . . . . .	74
5.7 Đọc thêm . . . . .	76
<b>References . . . . .</b>	77

<b>Index</b> .....	78
--------------------	----



Bảng 0.1: Bảng các ký hiệu

Ký hiệu	Ý nghĩa
$x, y, N, k$	in nghiêng, thường hoặc hoa, là các số vô hướng
$\mathbf{x}, \mathbf{y}$	in đậm, chữ thường, là các vector
$\mathbf{X}, \mathbf{Y}$	in đậm, chữ hoa, là các ma trận
$\mathbb{R}$	tập hợp các số thực
$\in$	phần tử thuộc tập hợp
$\exists$	tồn tại
$\forall$	mọi
$\triangleq$	ký hiệu là. Ví dụ $a \triangleq f(x)$ nghĩa là “ký hiệu $f(x)$ bởi $a$ ”.
$x_i$	phần tử thứ $i$ (tính từ 1) của vector $\mathbf{x}$
$\text{sgn}(x)$	hàm xác định dấu. Bằng 1 nếu $x \geq 0$ , bằng -1 nếu $x < 0$ .
$\exp(x)$	$e^x$
$a_{ij}$	phần tử hàng thứ $i$ , cột thứ $j$ của ma trận $\mathbf{A}$
$\mathbb{N}$	tập hợp các số tự nhiên
$\mathbb{C}$	tập hợp các số phức
$\mathbb{R}^m$	tập hợp các vector thực có $m$ phần tử
$\mathbb{R}^{m \times n}$	tập hợp các ma trận thực có $m$ hàng, $n$ cột
$\mathbf{A}^T$	chuyển vị của ma trận $\mathbf{A}$
$\mathbf{A}^H$	chuyển vị liên hợp (Hermitian) của ma trận phức $\mathbf{A}$
$\mathbf{A}^{-1}$	nghịch đảo của ma trận vuông $\mathbf{A}$ , nếu tồn tại
$\mathbf{A}^\dagger$	giả nghịch đảo của ma trận không nhất thiết vuông $\mathbf{A}$
$\mathbf{A}^{-T}$	nghịch đảo rồi chuyển vị của ma trận $\mathbf{A}$
$\ \mathbf{x}\ _p$	norm $p$ của vector $\mathbf{x}$
$\ \mathbf{A}\ _F$	Frobenius norm của ma trận $\mathbf{A}$
$\text{diag}(\mathbf{A})$	đường chéo chính của ma trận $\mathbf{A}$
$\text{trace}(\mathbf{A})$	trace của ma trận $\mathbf{A}$
$\det(\mathbf{A})$	định thức của ma trận vuông $\mathbf{A}$
$\text{rank}(\mathbf{A})$	hạng của ma trận $\mathbf{A}$
$\mathbb{S}^n$	tập hợp các ma trận vuông đối xứng bậc $n$
$\mathbb{S}_+^n$	tập hợp các ma trận nửa xác định dương bậc $n$
$\mathbb{S}_{++}^n$	tập hợp các ma trận xác định dương bậc $n$
o.w	otherwise – trong các trường hợp còn lại



## **Phần IV**

---

### **Neural Networks**

# Gradient Descent

---

## 1.1 Giới thiệu

Hình 1.2 mô tả sự biến thiên của hàm số  $f(x) = \frac{1}{2}(x - 1)^2 - 2$ .

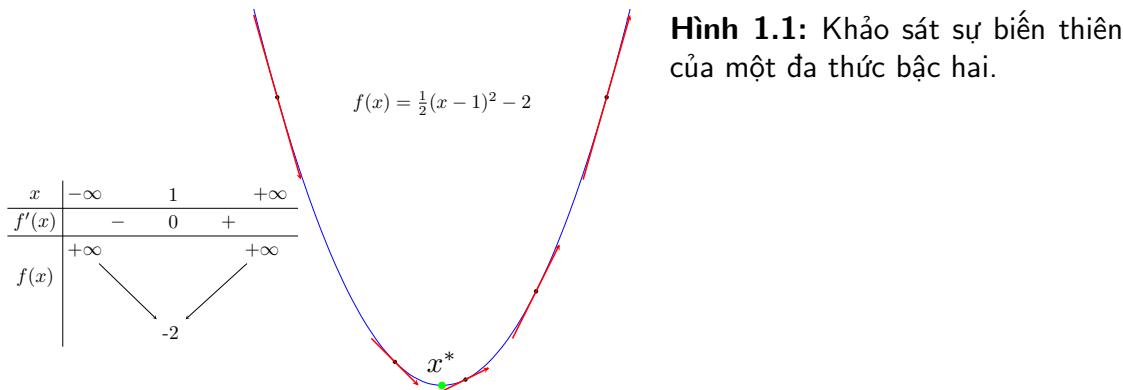
Điểm màu xanh lục là một điểm *cực tiểu* (*local minimum*), và cũng là điểm làm cho hàm số đạt giá trị nhỏ nhất (*global minimum*). Global minimum là một trường hợp đặc biệt của local minimum.

Giả sử chúng ta đang quan tâm đến một hàm số một biến có đạo hàm mọi nơi. Chúng ta cùng ôn lại một vài điểm cơ bản:

1. Điểm local minimum  $x^*$  của hàm số là điểm có đạo hàm  $f'(x^*)$  bằng không. Hơn thế nữa, trong lân cận của nó, đạo hàm của các điểm phía bên trái  $x^*$  là không dương, đạo hàm của các điểm phía bên phải  $x^*$  là không âm.
2. Đường tiếp tuyến với đồ thị hàm số đó tại một điểm bất kỳ có hệ số góc chính bằng đạo hàm của hàm số tại điểm đó.

Trong Hình 1.1, 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.

Trong machine learning nói riêng và toán tối ưu nói chung, chúng ta thường xuyên phải tìm các giá trị lớn nhất hoặc nhỏ nhất của một hàm số. Nếu chỉ xét riêng các hàm khả vi liên tục, việc giải phương trình đạo hàm bằng không thường rất phức tạp hoặc có thể ra vô số nghiệm. Thay vào đó, người ta thường cố gắng tìm các điểm local minimum, và ở một mức độ nào đó, coi đó là một nghiệm cần tìm của bài toán.



**Hình 1.1:** Khảo sát sự biến thiên của một đa thức bậc hai.

Các điểm local minimum là nghiệm của phương trình đạo hàm bằng không (vẫn đang giả sử rằng các hàm này liên tục và khả vi). 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. Tuy nhiên, trong hầu hết các trường hợp, việc giải phương trình đạo hàm bằng không 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.

Thực tế cho thấy, trong nhiều bài toán machine learning, các nghiệm local minimum thường đã cho kết quả tốt, đặc biệt là trong deep learning.

Hướng tiếp cận phổ biến nhất để giải quyết các bài toán tối ưu là xuất phát từ một điểm được 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 không. Gradient Descent (GD) và các biến thể của nó là một trong những phương pháp được dùng nhiều nhất.

## 1.2 Gradient Descent cho hàm một biến

Xét các hàm số một biến  $f : \mathbb{R} \rightarrow \mathbb{R}$ . Quay trở lại Hình 1.1 và một vài quan sát đã nêu. 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.

Ta lại có thêm hai quan sát rút ra từ Hình 1.1:

- 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, **ta cần di chuyển ngược dấu với đạo hàm**

$$x_{t+1} = x_t + \Delta \quad (1.1)$$

Trong đó  $\Delta$  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  $\Delta$ , 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

$$x_{t+1} = x_t - \eta f'(x_t) \quad (1.2)$$

Trong đó  $\eta$  là một số dương được gọi là *tốc độ học* (*learning rate*). 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 nên tảng cho rất nhiều phương pháp tối ưu.

### 1.2.1 Ví dụ đơn giản với Python

Xét hàm số  $f(x) = x^2 + 5 \sin(x)$  với đạo hàm  $f'(x) = 2x + 5 \cos(x)$ . Giả sử bắt đầu từ một điểm  $x_0$  nào đó, tại vòng lặp thứ  $t$ , chúng ta sẽ cập nhật như sau:

$$x_{t+1} = x_t - \eta(2x_t + 5 \cos(x_t)) \quad (1.3)$$

Khi thực hiện trên python, ta cần viết các hàm số<sup>1</sup>:

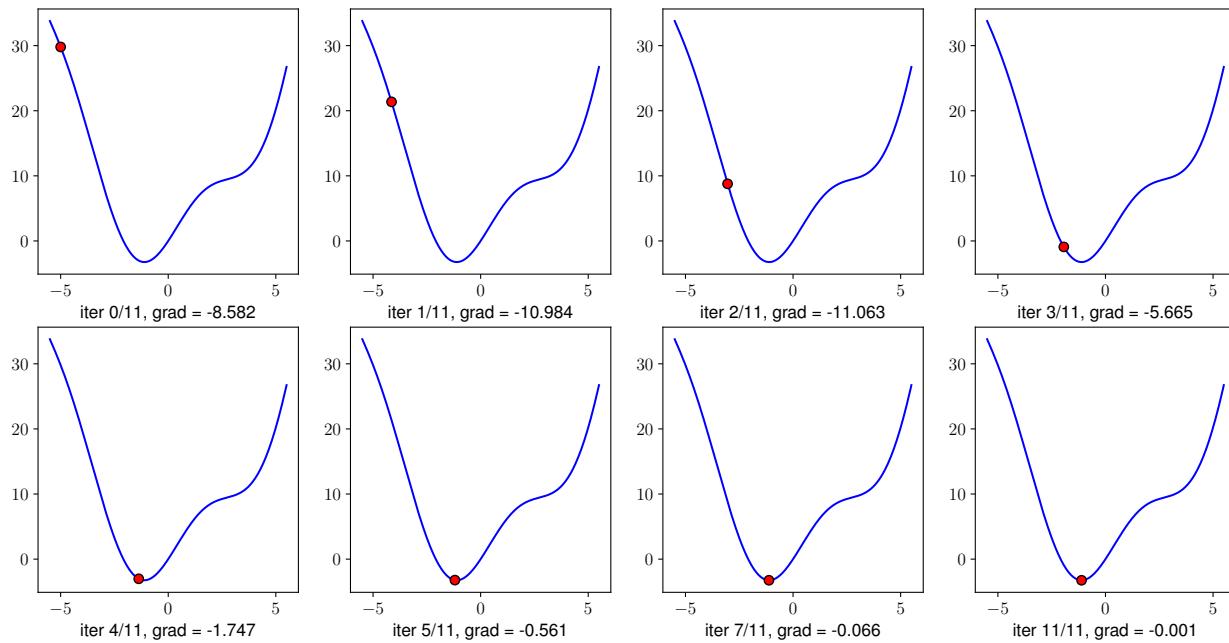
1. **grad** để tính đạo hàm.
2. **cost** để tính giá trị của hàm số. Hàm này không sử dụng trong thuật toán nhưng thường được dùng để kiểm tra việc tính đạo hàm của đúng không hoặc để xem giá trị của hàm số có giảm theo mỗi vòng lặp hay không.
3. **myGD1** là phần chính thực hiện thuật toán GD nêu phía 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(x0, eta):
    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)
```

<sup>1</sup> Giả sử rằng các thư viện đã được khai báo đầy đủ



**Hình 1.2:** Nghiệm tìm được qua các vòng lặp với  $x_0 = 5, \eta = 0.1$

### Điểm khởi tạo khác nhau

Sau khi đã có các hàm cần thiết, chúng ta thử tìm nghiệm với các điểm khởi tạo khác nhau là  $x_0 = -5$  và  $x_0 = 5$ , với cùng learning rate  $\eta = 0.1$ .

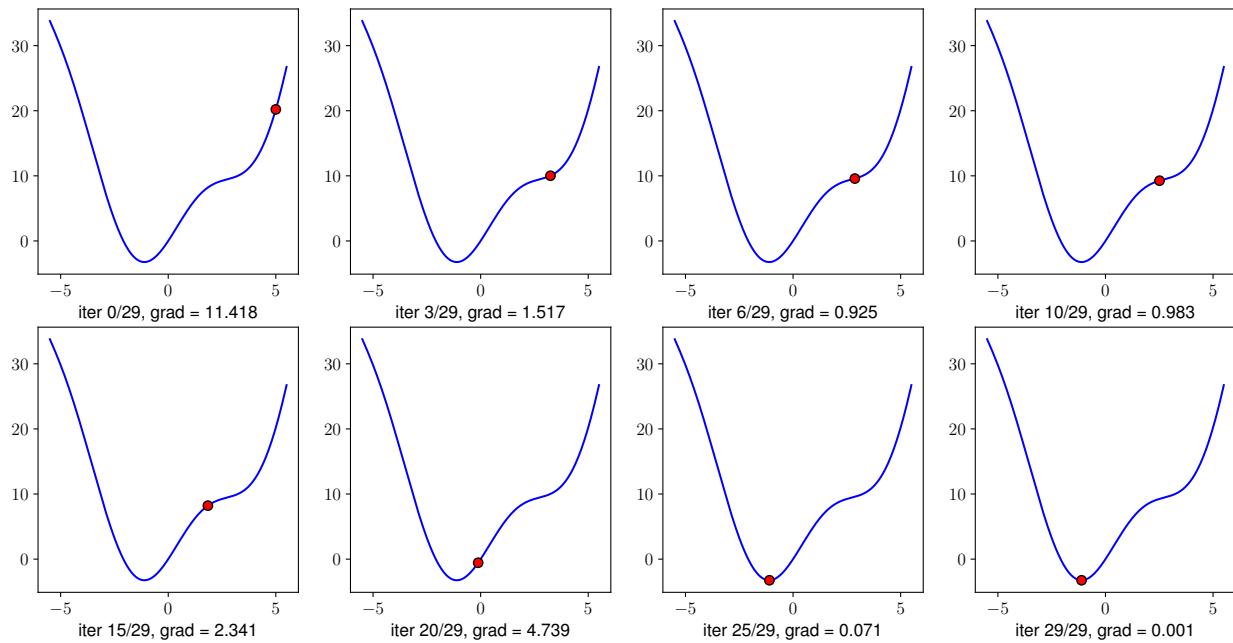
```
(x1, it1) = myGD1(-5, .1)
(x2, it2) = myGD1(5, 1)
print('Solution x1 = %f, cost = %f, after %d iterations' % (x1[-1], cost(x1[-1]), it1))
print('Solution x2 = %f, cost = %f, after %d iterations' % (x2[-1], cost(x2[-1]), it2))
```

Kết quả:

```
Solution x1 = -1.110667, cost = -3.246394, after 11 iterations
Solution x2 = -1.110341, cost = -3.246394, 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. Hình 1.2 và Hình 1.3 thể hiện vị trí của nghiệm và đạo hàm qua các vòng lặp với cùng learning rate  $\eta = .1$  nhưng điểm khởi tạo khác nhau tại  $-5$  và  $5$ .

Hình 1.2 tương ứng với  $x_0 = -5$ , cho thấy nghiệm hội tụ nhanh hơn, vì điểm ban đầu  $x_0$  gần với nghiệm  $x^* \approx -1$  hơn. Hơn nữa, *đường đi* tới nghiệm khá suôn sẻ với đạo hàm luôn âm và càng gần nghiệm thì đạo hàm càng nhỏ.



**Hình 1.3:** Nghiệm tìm được qua các vòng lặp với  $x_0 = -5$ ,  $\eta = 0.1$

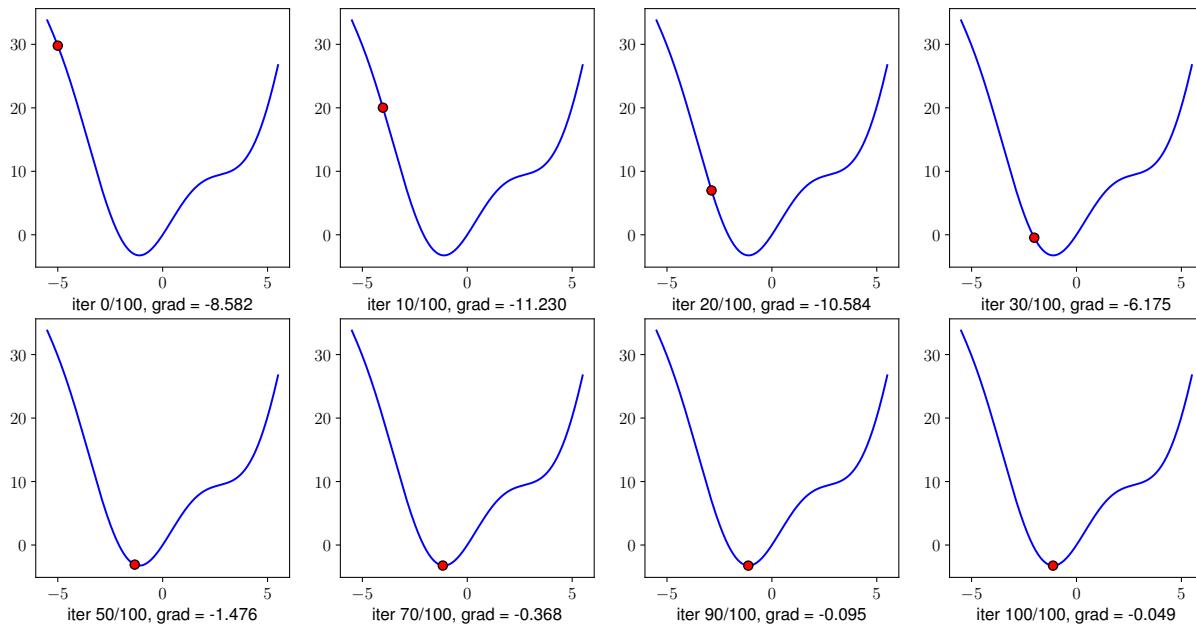
Trong Hình 1.3 tương ứng với  $x_0 = 5$ , *dường đi* của nghiệm có chứa một khu vực có đạo hàm khá nhỏ gần điểm có hoành độ bằng 2.5. Điều này khiến cho thuật toán *la cà* ở đây khá lâu. Khi vượt qua được điểm này thì mọi việc diễn ra rất tốt đẹp. Các điểm không phải là điểm cực tiểu nhưng có đạo hàm gần bằng không rất dễ gây ra hiện tượng nghiệm bị *bẫy* (*trapped*) tại đây vì đạo hàm nhỏ khiến nó không thay đổi nhiều ở vòng lặp tiếp theo. Chúng ta sẽ thấy một kỹ thuật khác giúp *thoát* được những chiếc bẫy này.

### Learning rate khác nhau

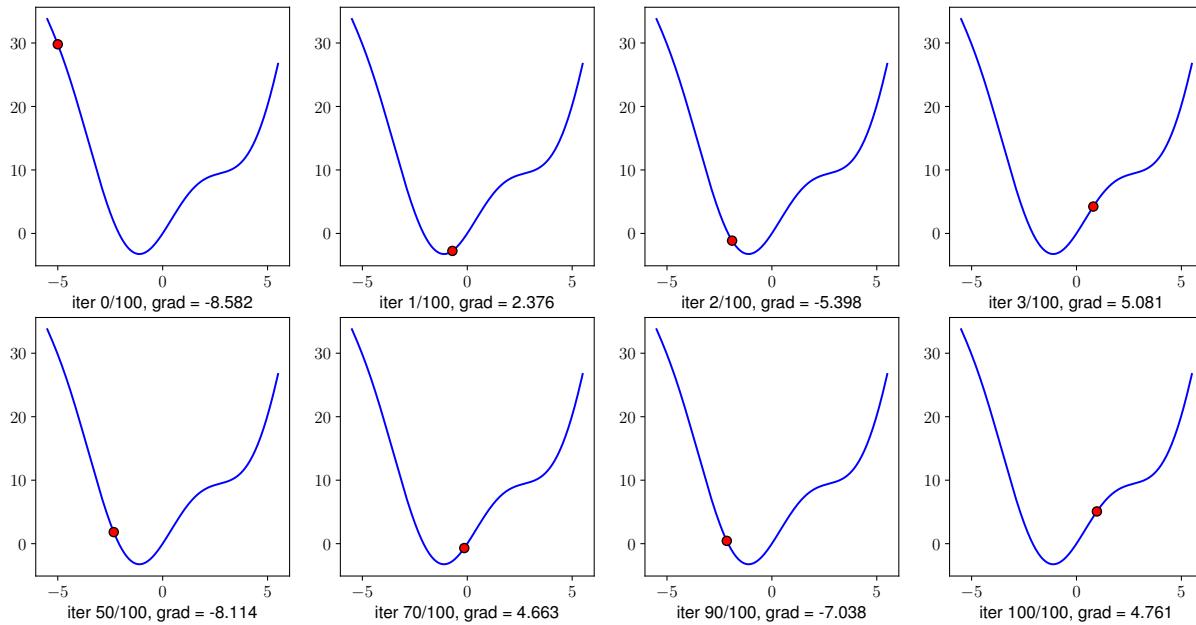
Tốc độ hội tụ của GD không những phụ thuộc vào điểm khởi tạo ban đầu mà còn phụ thuộc vào *learning rate*. Hình 1.4 và Hình 1.5 thể hiện vị trí của nghiệm qua các vòng lặp với cùng điểm khởi tạo  $x_0 = -5$  nhưng learning rate khác nhau.

Ta quan sát thấy hai điều:

1. Với *learning rate* nhỏ  $\eta = 0.01$  (Hình 1.4), tốc độ hội tụ rất chậm. Trong ví dụ này ta 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  $\eta = 0.5$  (Hình 1.5), thuật toán tiến rất nhanh tới *gần đích* sau vài vòng lặp. Tuy nhiên, thuật toán không hội tụ được vì sự thay đổi vị trí củ nghiệm sau mỗi vòng lặp là quá lớn, khiến nó cứ *quẩn quanh* ở đích mà vẫn không tới được đích.



**Hình 1.4:** Nghiệm tìm được qua các vòng lặp với  $x_0 = -5, \eta = 0.01$



**Hình 1.5:** Nghiệm tìm được qua các vòng lặp với  $x_0 = -5, \eta = 0.5$

Việc lựa chọn *learning rate* rất quan trọng. Việc 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.

### 1.3 Gradient Descent cho hàm nhiều biến

Giả sử ta cần tìm global minimum cho hàm  $f(\theta)$  trong đó  $\theta$  là tập hợp các tham số cần tối ưu. Đạo hàm của hàm số đó tại một điểm  $\theta$  bất kỳ được ký hiệu là  $\nabla_{\theta}f(\theta)$ . Tương tự như hàm một biến, thuật toán GD cho hàm nhiều biến cũng bắt đầu bằng một điểm dự đoán  $\theta_0$ , sau đó, ở vòng lặp thứ  $t$ , quy tắc cập nhật là

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta}f(\theta_t) \quad (1.4)$$

Hoặc viết dưới dạng đơn giản hơn:  $\theta \leftarrow \theta - \eta \nabla_{\theta}f(\theta)$ .

#### 1.3.1 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.

Nhắc lại hàm mất mát của Linear Regression và đạo hàm theo  $\mathbf{w}$  lần lượt là:

$$\begin{aligned} \mathcal{L}(\mathbf{w}) &= \frac{1}{2N} \|\mathbf{y} - \mathbf{X}^T \mathbf{w}\|_2^2 \\ \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) &= \frac{1}{N} \mathbf{X} (\mathbf{X}^T \mathbf{w} - \mathbf{y}) \end{aligned} \quad (1.5)$$

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

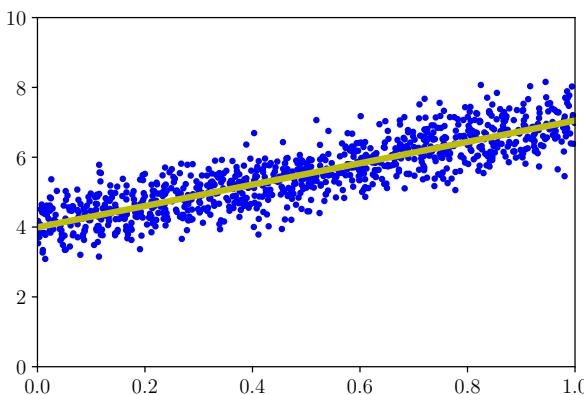
Trước hết, chúng ta tạo 1000 điểm dữ liệu được chọn gần với đường thẳng  $y = 4 + 3x$ , hiển thị chúng và tìm nghiệm theo công thức:

```
from sklearn.linear_model import LinearRegression
X = np.random.rand(1000)
y = 4 + 3 * X + .5*np.random.randn(1000) # noise added

model = LinearRegression()
model.fit(X.reshape(-1, 1), y.reshape(-1, 1))

w, b = model.coef_[0][0], model.intercept_[0]
sol_sklearn = np.array([b, w])
print(sol_sklearn)

# Draw the fitting line
x0 = np.linspace(0, 1, 2, endpoint=True)
y0 = w*x0 + b
plt.plot(X.T, y.T, 'b.')      # data
plt.plot(x0, y0, 'y', linewidth = 4) # the fitting line
plt.axis([0, 1, 0, 10])
plt.show()
```



**Hình 1.6:** Nghiệm của bài toán linear regression (đường thẳng màu vàng) tìm được bằng thư viện scikit-learn.

Kết quả:

```
Solution found by sklearn: [ 3.94323245  3.12067542]
```

Đường thẳng tìm được là đường có màu vàng có phương trình  $y \approx 3.97 + 3.01x$  (Xem Hình 1.6). Nghiệm tìm được rất gần với kỳ vọng.

Tiếp theo, ta sẽ thực hiện tìm nghiệm của linear regression sử dụng GD. Ta cần viết hàm mất mát và đạo hàm theo  $w$ . Chú ý rằng ở đây  $w$  đã bao gồm cả bias.

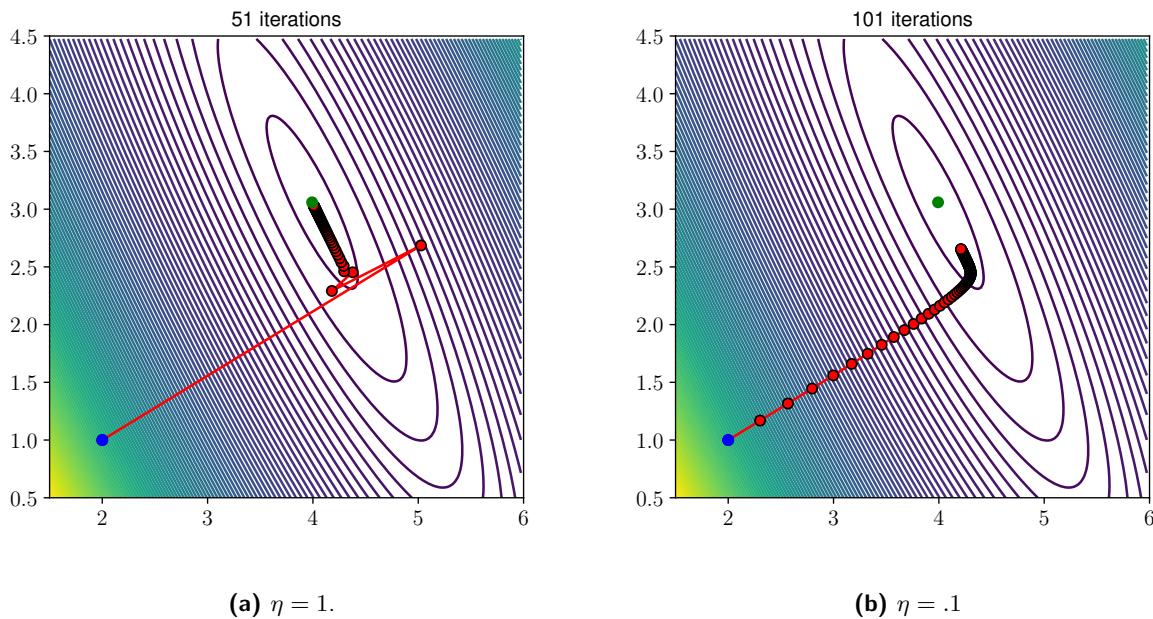
```
def grad(w):
    N = Xbar.shape[0]
    return 1/N * Xbar.T.dot(Xbar.dot(w) - y)

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

Với các hàm phức tạp, khi tính xong đạo hàm chúng ta cần kiểm tra đạo hàm thông qua numerical gradient (xem Mục ??). Trường hợp này tương đối đơn giản, việc kiểm tra đạo hàm xin giàn lại cho bạn đọc. Dưới đây là thuật toán GD cho bài toán.

```
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)

one = np.ones((X.shape[0],1))
Xbar = np.concatenate((one, X.reshape(-1, 1)), axis = 1)
w_init = np.array([[2], [1]])
(w1, it1) = myGD(w_init, grad, 1)
print('Sol found by GD: w = ', w1[-1].T, ',\nafter %d iterations.' % (it1+1))
```



**Hình 1.7:** Đường đi nghiệm của linear regression với các learning rate khác nhau.

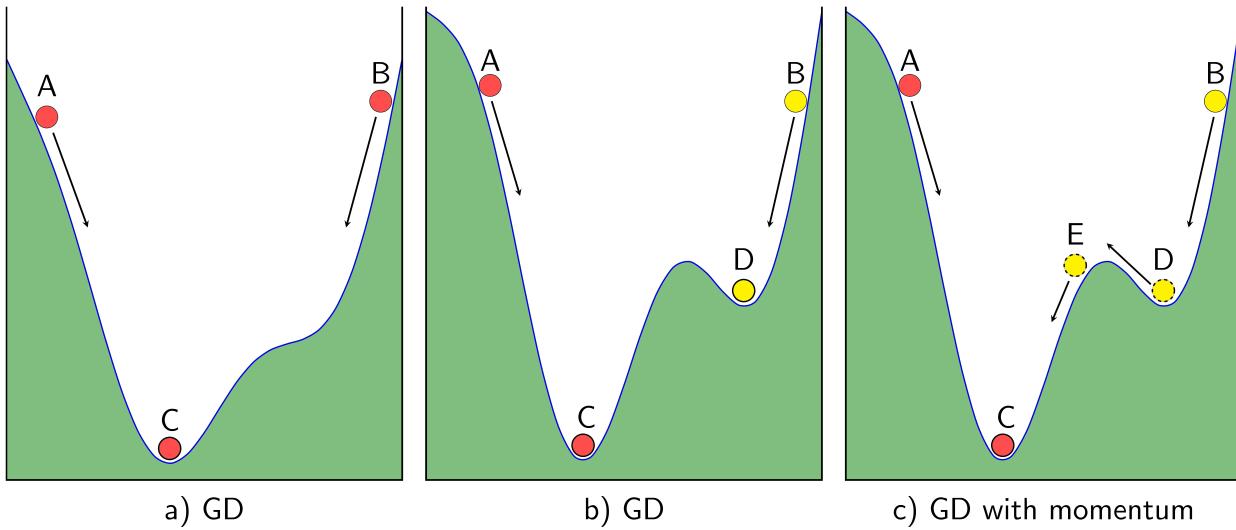
Kết quả:

```
Sol found by GD: w = [ 3.99026984  2.98702942] ,
after 49 iterations.
```

Sau 49 vòng lặp, thuật toán đã hội tụ với một nghiệm khá gần với nghiệm tìm được theo sklearn. Hình 1.7 mô tả đường đi của nghiệm với cùng điểm khởi tạo nhưng với learning rate khác nhau. Các điểm màu lam là các điểm khởi tạo. Các điểm màu lục là nghiệm tìm được bằng thư viện scikit-learn. Các điểm màu đỏ là nghiệm qua các vòng lặp trung gian. Ta thấy rằng khi  $\eta = 1$ , thuật toán hội tụ tới nghiệm theo thư viện sau 49 vòng lặp. Với learning rate nhỏ hơn,  $\eta = 0.1$ , sau hơn 100 vòng lặp, nghiệm vẫn còn cách xa nghiệm theo thư viện. Như vậy, việc chọn learning rate hợp lý là rất quan trọng.

Ở đây, chúng ta cùng làm quen với một khái niệm quan trọng: *đường đồng mức (level sets)*.

Ta thường gặp khái niệm *đường đồng mức* trong các bản đồ tự nhiên. Các điểm có cùng độ cao so với mực nước biển thường được nối với nhau. Với các ngọn núi, đường đồng mức thường là các đường kín bao quanh đỉnh núi. Tối ưu cũng có khái niệm tương tự. *Đường đồng mức* hay *level sets* của một hàm số là tập hợp các điểm làm cho hàm số có cùng giá trị. Tưởng tượng một hàm số với hai biến, đồ thị của nó là một *bề mặt (surface)* trong không gian ba chiều. Đường đồng mức có thể được xác định bằng cách *cắt* bề mặt này bằng một mặt phẳng song song với đáy và lấy giao điểm của chúng. Với dữ liệu hai chiều, hàm mất mát của linear regression là một hàm bậc hai của hai thành phần trong vector hệ số  $w$ . Đồ thị của nó là một bề mặt parabolic. Vì vậy, các đường đồng mức của hàm này là các đường



**Hình 1.8:** So sánh Gradient Descent với các hiện tượng vật lý.

ellipse có cùng tâm như trên Hình 1.7. Tâm này chính là đáy của parabolic và là giá trị nhỏ nhất của hàm mất mát. Các đường đồng mức được biểu diễn bởi các màu khác nhau với màu từ lam đậm đến lục, vàng, cam, đỏ, đỏ đậm thể hiện giá trị tăng dần.

## 1.4 Gradient descent với momentum

Trước hết, cùng nhắc lại thuật toán GD để tối ưu hàm mất mát  $J(\theta)$ :

1. Dự đoán một điểm khởi tạo  $\theta = \theta_0$ .
2. Cập nhật  $\theta$  đến khi đạt được kết quả chấp nhận được:

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \quad (1.6)$$

với  $\nabla_{\theta} J(\theta)$  là đạo hàm của hàm mất mát tại  $\theta$ .

### Gradient dưới góc nhìn vật lý

Thuật toán GD thường được ví với tác dụng của trọng lực lên một hòn bi đặt trên một mặt có dạng một thung lũng như Hình 1.8a. Bất kể ta đặt hòn bi ở A hay B thì cuối cùng hòn bi cũng sẽ lăn xuống và kết thúc ở vị trí C.

Tuy nhiên, nếu như bề mặt có hai đáy thung lũng như Hình 1.8b thì tùy vào việc đặt bi ở A hay B, vị trí cuối cùng của bi sẽ ở C hoặc D. Điểm D là một điểm local minimum ta không mong muốn.

Nếu suy nghĩ một cách vật lý hơn, vẫn trong Hình 1.8b, nếu vận tốc ban đầu của bi khi ở điểm B đủ lớn, khi bi lăn đến điểm D, theo *đà*, bi có thể tiếp tục di chuyển lên dốc phía bên trái của D. Và nếu giả sử vận tốc ban đầu lớn hơn nữa, bi có thể vượt dốc tới điểm E rồi lăn xuống C như trong Hình 1.8c. Dựa trên quan sát này, một thuật toán được ra đời nhằm giúp GD thoát được các local minimum. Thuật toán đó có tên là *momentum* (tức *theo đà*).

### Gradient Descent với momentum

Làm thế nào để biểu diễn *momentum* dưới dạng toán học.

Trong GD, chúng ta cần tính lượng thay đổi ở thời điểm  $t$  để cập nhật vị trí mới cho nghiệm (tức *hòn bi*). Nếu chúng ta coi đại lượng này như vận tốc  $v_t$  trong vật lý, vị trí mới của *hòn bi* sẽ là  $\theta_{t+1} = \theta_t - v_t$ , với giả sử rằng mỗi vòng lặp là một đơn vị thời gian. Dấu trừ thể hiện việc phải di chuyển ngược với đạo hàm. Việc tiếp theo là tính đại lượng  $v_t$  sao cho nó vừa mang thông tin của *độ dốc* (tức đạo hàm), vừa mang thông tin của *đà*, tức vận tốc trước đó  $v_{t-1}$  (với giả sử rằng vận tốc ban đầu  $v_0 = 0$ ). Một cách đơn giản nhất, ta có thể lấy tổng có trọng số của chúng:

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta) \quad (1.7)$$

Trong đó  $\gamma$  thường được chọn là một giá trị nhỏ hơn gần bằng một, thường là khoảng 0.9,  $v_t$  là vận tốc tại thời điểm trước đó,  $\nabla_{\theta} J(\theta)$  chính là độ dốc của điểm trước đó. Sau đó vị trí mới của *hòn bi* được xác định bởi

$$\theta \leftarrow \theta - v_t = \theta - \eta \nabla_{\theta} J(\theta) - \gamma v_{t-1} \quad (1.8)$$

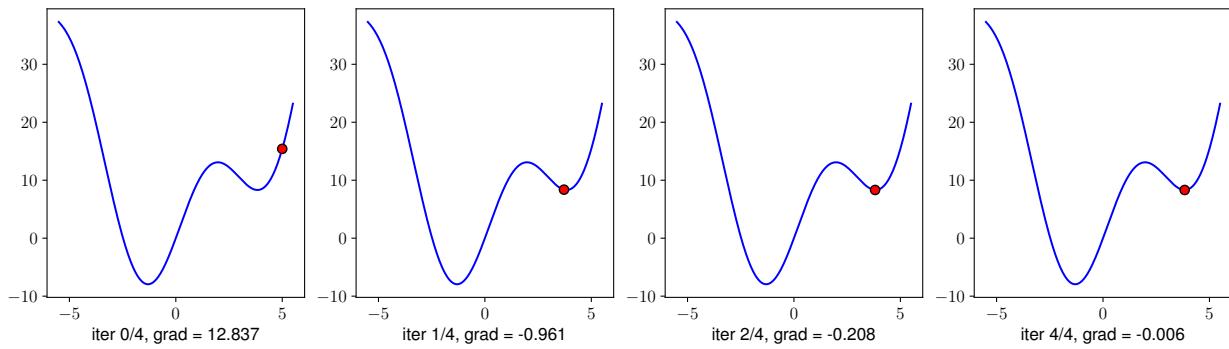
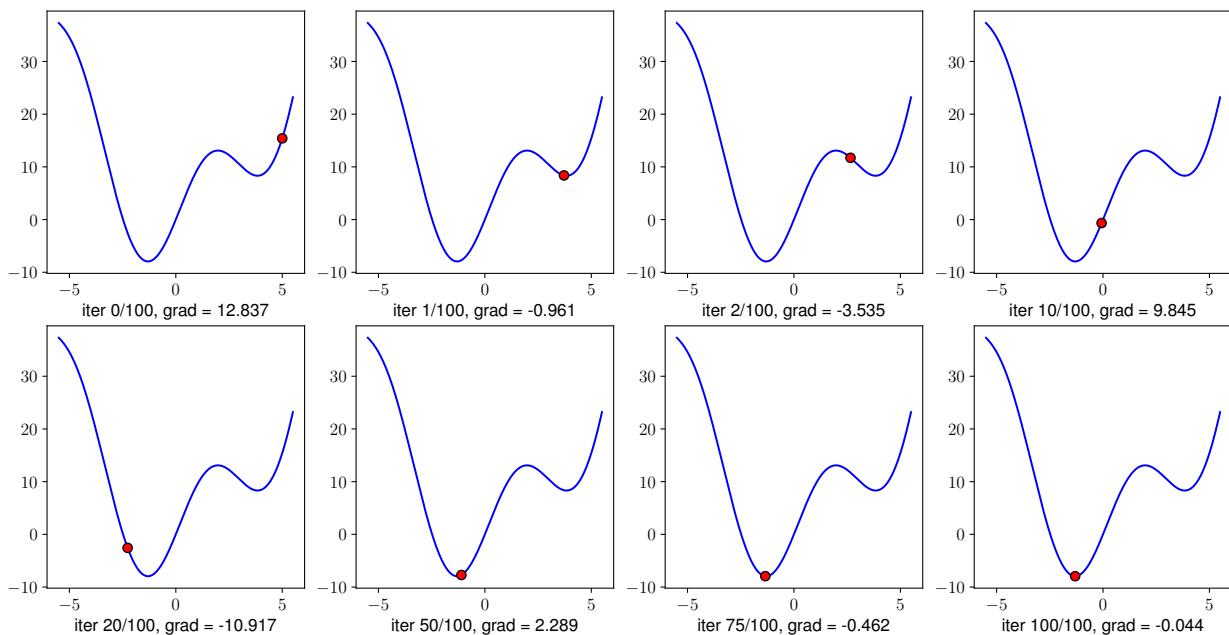
Thuật toán đơn giản này tỏ ra rất hiệu quả trong các bài toán thực tế (trong không gian nhiều chiều, cách tính toán cũng hoàn toàn tương tự). Dưới đây là một ví dụ trong không gian một chiều.

Chúng ta xem xét một hàm đơn giản có hai điểm local minimum, trong đó một điểm là global minimum

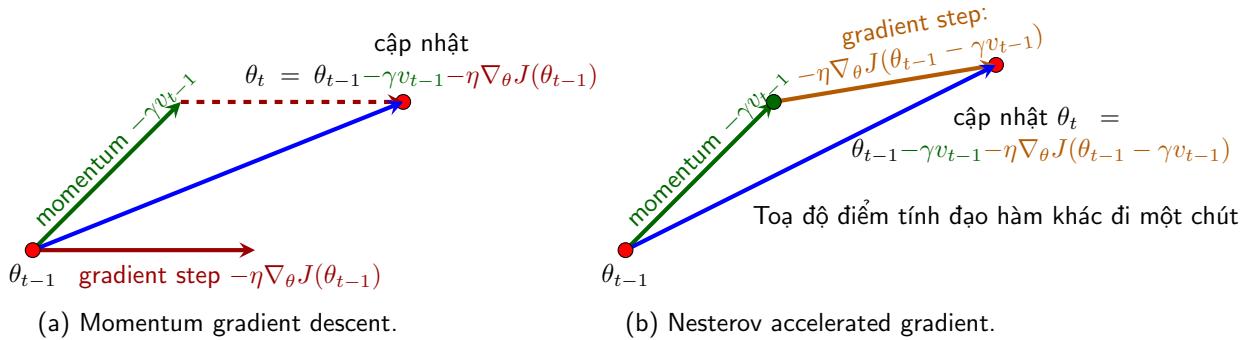
$$f(x) = x^2 + 10 \sin(x) \quad (1.9)$$

với đạo hàm  $f'(x) = 2x + 10 \cos(x)$ . Hình 1.9 thể hiện đường đi của nghiệm cho bài toán này khi không sử dụng momentum. Ta thấy rằng thuật toán hội tụ nhanh chóng sau chỉ bốn vòng lặp. Tuy nhiên, nghiệm dừng lại ở một điểm local minimum. Trong khi đó, Hình 1.10 thể hiện đường đi của nghiệm khi có sử dụng momentum. Chúng ta thấy rằng *hòn bi* vượt được dốc thứ nhất nhờ có *đà*, theo tính tiếp tục vượt qua điểm global minimum, nhưng quay trở lại điểm này sau 50 vòng lặp rồi chuyển động chậm dần quanh đó tới khi dừng hẳn ở vòng lặp thứ 100. Ví dụ này cho thấy momentum thực sự đã giúp nghiệm thoát được khu vực local minimum.

Nếu biết trước điểm *đặt bi* ban đầu **theta**, đạo hàm của hàm mất mát tại một điểm bất kỳ **grad(theta)**, lượng thông tin lưu trữ từ vận tốc trước đó **gamma** và learning rate **eta**, chúng ta có thể viết hàm số **GD\_momentum** như sau.

**Hình 1.9:** Gradient descent thông thường.**Hình 1.10:** Gradient descent với momentum.

```
def GD_momentum(grad, theta_init, eta, gamma):
    # Suppose we want to store history of theta
    theta = [theta_init]
    v_old = np.zeros_like(theta_init)
    for it in range(100):
        v_new = gamma*v_old + eta*grad(theta[-1])
        theta_new = theta[-1] - v_new
        if np.linalg.norm(grad(theta_new))/np.array(theta_init).size < 1e-3:
            break
        theta.append(theta_new)
        v_old = v_new
    return theta
```

**Hình 1.11:** Ý tưởng của Nesterov accelerated gradient.

## 1.5 Nesterov accelerated gradient

Momentum giúp *hòn bi* vượt qua được *dốc locaminimum*, tuy nhiên, có một hạn chế chúng ta có thể thấy trong ví dụ trên: khi tới gần *dích*, momentum vẫn mất khá nhiều thời gian trước khi dừng lại, chính vì có *dà*. Một kỹ thuật có tên *Nesterov accelerated gradient* – NAG [Nes07] giúp cho thuật toán momentum GD hội tụ nhanh hơn.

### Ý tưởng chính

Ý tưởng trung tâm của thuật toán là *dự đoán vị trí của nghiệm trước một bước*. Cụ thể, nếu sử dụng số hạng momentum  $\gamma v_{t-1}$  để cập nhật thì ta có thể *xấp xỉ* được vị trí tiếp theo của nghiệm là  $\theta - \gamma v_{t-1}$ . Vậy, thay vì sử dụng gradient của điểm hiện tại, NAG *đi trước một bước*, sử dụng gradient của điểm tiếp theo. Ý tưởng này được thể hiện trên Hình 1.11.

- 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.

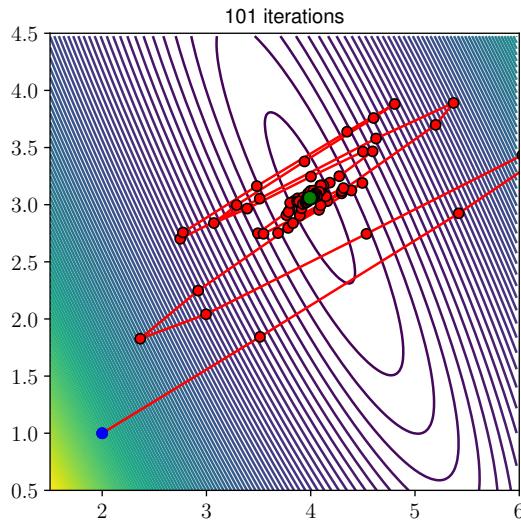
Sự khác nhau giữa momentum và NAG nằm ở điểm lấy đạo hàm. Ở momentum, điểm được lấy đạo hàm chính là vị trí hiện tại của nghiệm. Ở NAG, điểm được lấy đạo hàm là điểm *có được nếu sử dụng momentum*.

### Công thức cập nhật

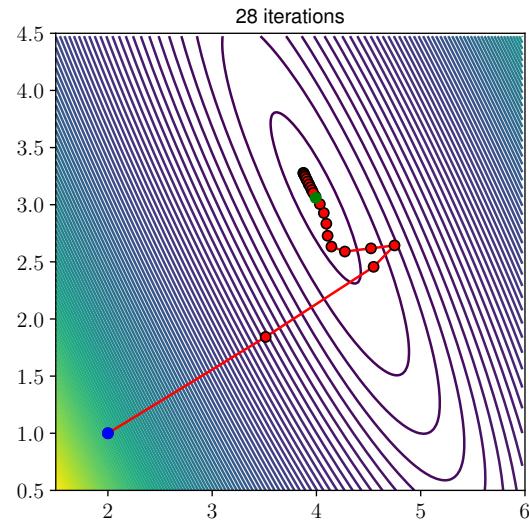
Công thức cập nhật của NAG được cho như sau:

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1}) \quad (1.10)$$

$$\theta \leftarrow \theta - v_t \quad (1.11)$$



(a) GD với momentum.



(b) Nesterov accelerated gradient.

**Hình 1.12:** Đường đi của nghiệm cho bài toán linear regression với hai phương pháp gradient descent khác nhau. NAG cho nghiệm mượt hơn và nhanh hơn.

Đoạn code dưới đây là hàm số cho NAG.

```
def GD_NAG(grad, theta_init, eta, gamma):
    theta = [theta_init]
    v = [np.zeros_like(theta_init)]
    for it in range(100):
        v_new = gamma*v[-1] + eta*grad(theta[-1] - gamma*v[-1])
        theta_new = theta[-1] - v_new
        if np.linalg.norm(grad(theta_new))/np.array(theta_init).size < 1e-3:
            break
        theta.append(theta_new)
        v.append(v_new)
    return theta
```

### Ví dụ minh họa

Chúng ta cùng áp dụng cả GD với momentum và GD với NAG cho bài toán linear regression đề cập ở trên. Hình 1.12 thể hiện đường đi của nghiệm khi sử dụng hai phương pháp này.

Hình 1.12a là đường đi của nghiệm với phương pháp momentum. nghiệm đi khá zigzag và mất nhiều vòng lặp hơn. Hình 1.12b là đường đi của nghiệm với phương pháp NAG, nghiệm hội tụ nhanh hơn, và đường đi ít zigzag hơn.

## 1.6 Stochastic Gradient Descent

### 1.6.1 Batch gradient descent

Thuật toán GD chúng ta sử dụng từ đầu chương tối hiện tại còn được gọi là *batch gradient descent*. Batch ở đây được hiểu là *tất cả*, tức khi cập nhật các tham số  $\theta$ , chúng ta sử dụng **tất cả** các điểm dữ liệu  $\mathbf{x}_i$ . Hạn chế của việc này là khi lượng cơ sở dữ liệu lớn tới hàng triệu, việc tính toán đạo hàm trên toàn bộ dữ liệu tại mỗi vòng lặp sẽ tốn rất nhiều thời gian.

*Online learning* là khi cơ sở dữ liệu được cập nhật liên tục, mỗi lần tăng thêm vài điểm dữ liệu mới, yêu cầu cập nhật mô hình mới. Kéo theo đó là mô hình cũng phải được thay đổi một chút để phù hợp với dữ liệu mới. Nếu làm theo batch GB, 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 có thể không còn mang tính *online* nữa do mất quá nhiều thời gian tính toán.

Một thuật toán đơn giản hơn, chấp nhận việc có sai số một chút nhưng lại lợi ích tính toán cao, thường được sử dụng có tên gọi là *stochastic gradient descent* – SGD.

### 1.6.2 Stochastic gradient descent

Trong SGD, tại một thời điểm (vòng lặp – iteration), ta chỉ tính đạo hàm của hàm mất mát dựa trên *chỉ một* điểm dữ liệu  $\mathbf{x}_i$  rồi cập nhật  $\theta$  dựa trên đạo hàm này. Chú ý rằng hàm mất mát thường được lấy trung bình trên *mỗi* điểm dữ liệu nên đạo hàm tại một điểm cũng thường khá gần với đạo hàm của hàm mất mát. Sau khi duyệt qua *tất cả* các điểm dữ liệu, thuật toán quay lại quá trình trên. Biến thể đơn giản này trên thực tế làm việc rất hiệu quả.

#### epoch

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 một lần cập nhật  $\theta$ , với SGD thì mỗi epoch ứng với  $N$  lần cập nhật  $\theta$  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 một epoch. Nhưng nhìn vào một mặt khác, với SGD, nghiệm có thể hội tụ sau vài vòng lặp. Vì vậy SGD phù hợp với các bài toán có lượng cơ sở dữ liệu lớn và các bài toán yêu cầu mô hình thay đổi liên tục như online learning. Với một mô hình đã được huấn luyện từ trước, khi có thêm dữ liệu, ta có thể chỉ cần chạy thêm một vài epoch nữa là đã có nghiệm hội tụ.

#### Thứ tự lựa chọn điểm dữ liệu

Một điểm cần lưu ý đó là sau mỗi epoch, chúng ta cần *xáo trộn* (*shuffle*) thứ tự của các dữ liệu để đảm bảo tính ngẫu nhiên. Việc này cũng ảnh hưởng tới hiệu năng của SGD. Đây cũng chính là lý do thuật toán này có chứa từ *stochastic*.

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

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta; \mathbf{x}_i; \mathbf{y}_i) \quad (1.12)$$

trong đó  $J(\theta; \mathbf{x}_i; \mathbf{y}_i) \triangleq J_i(\theta)$  là hàm mất mát với chỉ một cặp điểm dữ liệu là thứ  $i$ . Chú ý rằng các thuật toán biến thể của GD như momentum hay NAG hoàn toàn có thể được áp dụng vào SGD.

### 1.6.3 Mini-batch gradient descent

Khác với SGD, mini-batch sử dụng một số lượng  $k$  lớn hơn một (nhưng vẫn nhỏ hơn tổng số điểm dữ liệu  $N$  rất nhiều). Giống với SGD, mini-batch GD 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  $k$  điểm dữ liệu (trừ mini-batch cuối có thể có ít hơn nếu  $N$  không chia hết cho  $k$ ). Ở mỗi vòng lặp, thuật toán này lấy ra một mini-batch để tính toán đạo hàm rồi cập nhật. Một epoch cũng là khi thuật toán chạy hết dữ liệu một lượt.

Mini-batch GD được sử dụng trong hầu hết các thuật toán machine learning, đặc biệt là trong deep learning. Giá trị  $k$  thường được chọn là khoảng từ vài chục đến vài trăm.

## 1.7 Điều kiện dừng thuật toán

Có một điểm chúng ta chưa đề cập kỹ – khi nào thì nên dừng thuật toán gradient descent?

Trong thực nghiệm, chúng ta có thể kết hợp các phương pháp sau.

1. Giới hạn số vòng lặp. Đây là phương pháp phổ biến nhất và cũng dễ đảm bảo rằng chương trình chạy không quá lâu. Tuy nhiên, một nhược điểm của cách làm này là có thể thuật toán dừng lại trước khi nghiệm đủ tốt.
2. So sánh gradient của nghiệm tại hai lần cập nhật liên tiếp, khi nào giá trị này đủ nhỏ thì dừng lại. Phương pháp này cũng có một nhược điểm lớn là việc tính đạo hàm đôi khi trả nên quá phức tạp.
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 của phương pháp này là nếu tại một thời điểm, đồ thị hàm số có dạng *bảng phẳng* tại một khu vực nhưng khu vực đó không chứa điểm local minimum, thuật toán cũng dừng lại trước khi đạt giá trị mong muốn.
4. Vừa chạy gradient descent, vừa kiểm tra kết quả. Một kỹ thuật thường được sử dụng nữa là cho thuật toán chạy với số lượng vòng lặp cực lớn. Trong quá trình chạy, chương trình thường xuyên kiểm tra chất lượng mô hình bằng cách áp dụng nó lên dữ liệu tập

huấn luyện hoặc validation. Đồng thời, mô hình sau một vài vòng lặp được lưu lại trong bộ nhớ. Mô hình tốt nhất có thể không phải là mô hình với số vòng lặp lớn hơn (xem thêm early stopping để tránh overfitting).

## 1.8 Đọc thêm

Source code trong chương này có thể được tìm thấy [tại đây](#).

Ngoài các thuật toán đã đề cập trong chương này, rất nhiều thuật toán khác giúp cải thiện gradient descent được đề xuất gần đây [Rud16]. Bạn đọc có thể đọc thêm AdaGrad [DHS11], RMSProp [TH12], Adam [KB14], v.v..

Các trang web và video dưới đây cũng là các tài liệu tốt cho gradient descent.

1. [An overview of gradient descent optimization algorithms](#)
2. [Stochastic Gradient Descent - Wikipedia](#)
3. [Stochastic Gradient Descent - Andrew Ng](#)
4. [An Interactive Tutorial on Numerical Optimization](#)
5. [Machine Learning cơ bản – Bài 7, 8](#)

# Perceptron Learning Algorithm

---

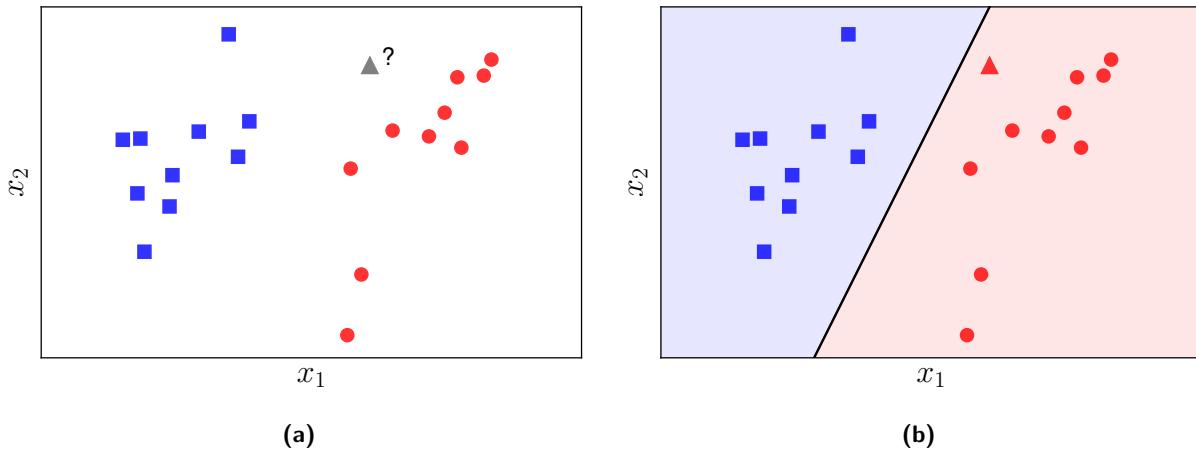
## 2.1 Giới thiệu

Trong chương này, chúng ta cùng tìm hiểu một trong các thuật toán đầu tiên trong lịch sử machine learning. Đây là một thuật toán phân lớp đơn giản có tên là *perceptron learning algorithm* (PLA [Ros57]). Thuật toán này được thiết kế cho bài toán *phân lớp nhị phân (binary classification)* với chỉ hai lớp dữ liệu. Đây là nền tảng cho các thuật toán liên quan tới neural networks rồi deep learning sau này.

Giả sử có hai tập hợp dữ liệu đã được gán nhãn được minh họa trong Hình 2.1a. Hai lớp dữ liệu là tập các điểm màu xanh và tập các điểm màu đỏ. Bài toán đặt ra là từ dữ liệu của hai tập được gán nhãn cho trước, hãy xây dựng một bộ phân lớp có khả năng dự đoán được nhãn (màu) của một điểm dữ liệu mới, chẳng hạn điểm màu xám.

Nếu coi mỗi vector đặc trưng là một điểm trong không gian nhiều chiều, bài toán phân lớp có thể được coi như bài toán xác định mỗi điểm trong không gian thuộc vào lớp nào. Nói cách khác, nếu ta coi mỗi lớp *chiếm* một hoặc vài vùng *lãnh thổ* trong không gian, ta cần đi tìm *ranh giới (boundary)* giữa các vùng đó. Ranh giới đơn giản nhất trong không gian hai chiều là một đường thẳng, trong không gian ba chiều là một mặt phẳng, trong không gian nhiều chiều hơn là một *siêu mặt phẳng* hoặc *siêu phẳng (hyperplane)*. Những ranh giới phẳng này được coi là đơn giản vì chúng có thể được biểu diễn dưới dạng toán học bằng một hàm số tuyến tính. Tất nhiên, ta đang giả sử rằng tồn tại một siêu phẳng như vậy. Hình 2.1b minh họa một đường thẳng phân chia hai lớp trong không gian hai chiều. Lãnh thổ của hai lớp xanh và đỏ được mô tả bởi hai nửa mặt phẳng với màu tương ứng. Trong trường hợp này, điểm dữ liệu mới hình tam giác được phân vào lớp đỏ.

Perceptron learning algorithm (PLA) là một thuật toán đơn giản giúp tìm một ranh giới siêu phẳng cho bài toán phân lớp nhị phân, với giả sử rằng tồn tại ranh giới phẳng đó. Nếu hai lớp dữ liệu có thể được phân chia hoàn toàn bằng một siêu phẳng, ta nói rằng hai lớp đó *linearly separable*.



**Hình 2.1:** Bài toán phân lớp nhị phân trong không gian hai chiều. (a) Cho hai lớp dữ liệu vuông xanh và tròn đỏ, hãy xác định điểm dữ liệu tam giác xám thuộc lớp nào. (b) Ví dụ về một ranh giới phẳng của hai lớp, điểm tam giác được phân vào lớp đỏ với đường ranh giới này.

## 2.2 Thuật toán perceptron

### 2.2.1 Cách phân lớp của perceptron learning algorithm

Giả sử  $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N] \in \mathbb{R}^{d \times N}$  là ma trận chứa các điểm dữ liệu huấn luyện mà mỗi cột  $\mathbf{x}_i$  là một điểm dữ liệu trong không gian  $d$  chiều. Giả sử thêm các nhãn tương ứng với từng điểm dữ liệu được lưu trong một vector hàng  $\mathbf{y} = [y_1, y_2, \dots, y_N] \in \mathbb{R}^{1 \times N}$ , với  $y_i = 1$  nếu  $\mathbf{x}_i$  thuộc lớp thứ nhất (vuông xanh) và  $y_i = -1$  nếu  $\mathbf{x}_i$  thuộc lớp còn lại (tròn đỏ).

Tại một thời điểm, giả sử ta tìm được ranh giới là một siêu phẳng có phương trình

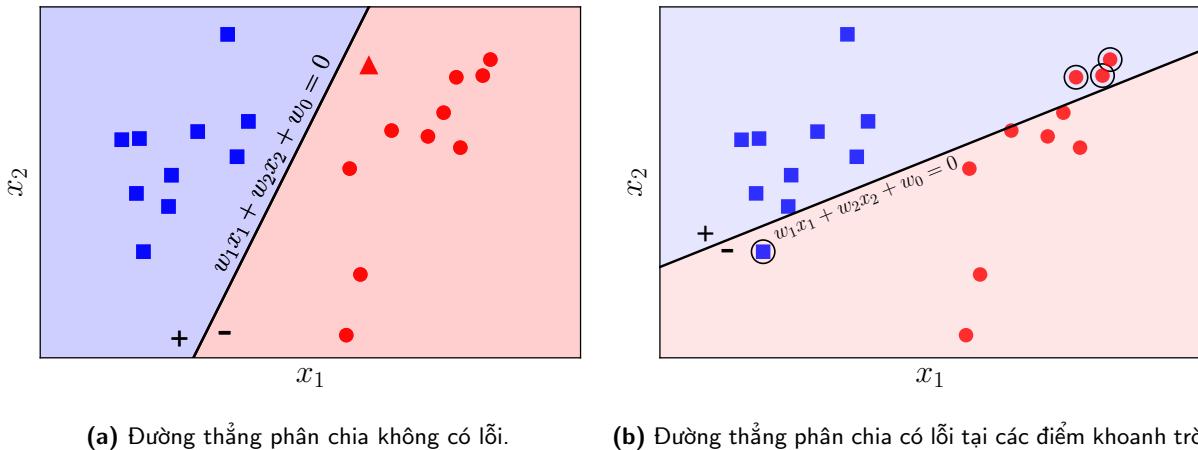
$$f_{\mathbf{w}}(\mathbf{x}) = w_1x_1 + \cdots + w_dx_d + w_0 = \mathbf{w}^T\mathbf{x} + w_0 = 0 \quad (2.1)$$

với  $\mathbf{w} \in \mathbb{R}^d$  là vector hệ số và  $w_0$  là số hạng tự do được gọi là bias. Bằng cách sử dụng bias trick (xem Mục ??), ta có thể coi phương trình siêu phẳng là  $f_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T\mathbf{x} = 0$  với  $\mathbf{x}$  ở đây được ngầm hiểu là vector đặc trưng mở rộng thêm một đặc trưng bằng 1. Vector hệ số  $\mathbf{w}$  cũng chính là *vector pháp tuyến* của siêu phẳng  $\mathbf{w}^T\mathbf{x} = 0$ .

Trong không gian hai chiều, giả sử đường thẳng  $w_1x_1 + w_2x_2 + w_0 = 0$  chính là nghiệm cần tìm như Hình 2.2a. Nhận xét rằng các điểm nằm về cùng một phía so với đường thẳng này sẽ làm cho hàm số  $f_{\mathbf{w}}(\mathbf{x})$  mang cùng dấu. Chỉ cần đổi dấu của  $\mathbf{w}$  nếu cần thiết, ta có thể giả sử các điểm nằm trong nửa mặt phẳng nền xanh mang dấu dương (+), các điểm nằm trong nửa mặt phẳng nền đỏ mang dấu âm (-). Các dấu này cũng tương đương với nhãn  $y$  của mỗi nhãn. Vậy nếu  $\mathbf{w}$  là một nghiệm của bài toán perceptron, với một điểm dữ liệu mới  $\mathbf{x}$  chưa được gán nhãn, ta có thể xác định nhãn của nó bằng một phép toán đơn giản:

$$\text{label}(\mathbf{x}) = \begin{cases} 1 & \text{nếu } \mathbf{w}^T\mathbf{x} \geq 0 \\ -1 & \text{o.w.} \end{cases} \quad (2.2)$$

Nói cách khác,  $\text{label}(\mathbf{x}) = \text{sgn}(\mathbf{w}^T\mathbf{x})$  với  $\text{sgn}$  là hàm xác định dấu, giả sử rằng  $\text{sgn}(0) = 1$ .



**Hình 2.2:** Ví dụ về các đường thẳng trong không gian hai chiều: (a) một nghiệm của bài toán PLA, (b) không phải nghiệm.

### 2.2.2 Xây dựng hàm măt măt

Tiếp theo, chúng ta xây dựng một hàm măt măt với tham số  $\mathbf{w}$  bất kỳ. Vẫn trong không gian hai chiều, giả sử đường thẳng  $w_1x_1 + w_2x_2 + w_0 = 0$  được cho như Hình 2.2b. Các điểm được khoanh tròn là các điểm bị *phân lớp lỗi* (*misclassified*). Ta luôn muốn rằng không có điểm nào bị phân lớp lỗi. Một cách tự nhiên, ta có thể sử dụng hàm *đếm* số lượng các điểm bị phân lớp lỗi và tìm cách tối thiểu hàm số này.

Xét một điểm  $\mathbf{x}_i$  bất kỳ với nhãn  $y_i$ . Nếu nó bị phân lớp lỗi, ta phải có  $\text{sgn}(\mathbf{w}^T \mathbf{x}) \neq y_i$ . Vì hai giá trị này chỉ bằng 1 hoặc -1, ta sẽ có  $y_i \text{sgn}(\mathbf{w}^T \mathbf{x}) = -1$ . Như vậy, hàm số đếm số lượng điểm bị phân lớp lỗi có thể được viết dưới dạng

$$J_1(\mathbf{w}) = \sum_{\mathbf{x}_i \in \mathcal{M}} (-y_i \text{sgn}(\mathbf{w}^T \mathbf{x}_i)) \quad (2.3)$$

trong đó  $\mathcal{M}$  ký hiệu tập các điểm bị phân lớp lỗi ứng với mỗi  $\mathbf{w}$ . Mục đích cuối cùng là đi tìm  $\mathbf{w}$  sao cho không có điểm nào bị phân lớp lỗi, tức  $J_1(\mathbf{w}) = 0$ . Một điểm quan trọng, hàm số này là rắc rạc, không có đạo hàm tại nhiều điểm nên rất khó được tối ưu. Chúng ta cần tìm một hàm măt măt khác để việc tối ưu khả thi hơn. Xét hàm măt măt

$$J(\mathbf{w}) = \sum_{\mathbf{x}_i \in \mathcal{M}} (-y_i \mathbf{w}^T \mathbf{x}_i) \quad (2.4)$$

Hàm  $J(\mathbf{w})$  khác một chút với hàm  $J_1(\mathbf{w})$  ở chỗ hàm rắc rạc  $\text{sgn}$  đã được lược bỏ. Ngoài ra, khi một điểm bị phân lớp lỗi  $\mathbf{x}_i$  nằm càng xa ranh giới, giá trị  $-y_i \mathbf{w}^T \mathbf{x}_i$  sẽ càng lớn, nghĩa là hàm măt măt sẽ lớn lên. Vì tổng vẫn được tính trên các tập điểm bị phân lớp lỗi  $\mathcal{M}$ , giá trị nhỏ nhất của hàm măt măt này cũng bằng không nếu không có điểm nào bị phân lớp lỗi. Vì vậy,  $J(\mathbf{w})$  được cho là tốt hơn  $J_1(\mathbf{w})$  vì nó *trừng phạt* rất nặng những điểm *lân sâm sang lanh thở* của lớp kia. Trong khi đó,  $J_1()$  *trừng phạt* các điểm phân lớp lỗi một lượng như nhau bằng một, bất kể chúng gần hay xa ranh giới.

### 2.2.3 Tối ưu hàm mất mát

Tại một thời điểm, nếu ta chỉ quan tâm tới các điểm bị phân lớp lỗi thì hàm số  $J(\mathbf{w})$  khả vi tại mọi  $\mathbf{w}$ , vậy ta có thể sử dụng gradient descent hoặc stochastic gradient descent (SGD) để tối ưu hàm mất mát này. Chúng ta sẽ giải quyết bài toán tối ưu hàm mất mát  $J(\mathbf{w})$  bằng SGD bằng cách cập nhật  $\mathbf{w}$  tại mỗi vòng lặp dựa trên chỉ một điểm dữ liệu. Với chỉ một điểm dữ liệu  $\mathbf{x}_i$  bị phân lớp lỗi, hàm mất mát và đạo hàm của nó lần lượt là

$$J(\mathbf{w}; \mathbf{x}_i; y_i) = -y_i \mathbf{w}^T \mathbf{x}_i; \quad \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{x}_i; y_i) = -y_i \mathbf{x}_i \quad (2.5)$$

Vậy quy tắc cập nhật  $\mathbf{w}$  sử dụng SGD là

$$\mathbf{w} \leftarrow \mathbf{w} - \eta (-y_i \mathbf{x}_i) = \mathbf{w} + \eta y_i \mathbf{x}_i \quad (2.6)$$

với  $\eta$  là learning rate. Trong PLA,  $\eta$  được chọn bằng 1. Ta có một quy tắc cập nhật rất gọn:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + y_i \mathbf{x}_i \quad (2.7)$$

Nói cách khác, với mỗi điểm  $\mathbf{x}_i$  bị phân lớp lỗi, bằng cách nhân điểm đó với nhãn  $y_i$  của nó, lấy kết quả cộng vào  $\mathbf{w}$  hiện tại, ta sẽ được  $\mathbf{w}$  mới. Tiếp theo, ta thấy rằng

$$\mathbf{w}_{t+1}^T \mathbf{x}_i = (\mathbf{w}_t + y_i \mathbf{x}_i)^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i + y_i \|\mathbf{x}_i\|_2^2 \quad (2.8)$$

Nếu  $y_i = 1$ , vì  $\mathbf{x}_i$  bị phân lớp lỗi nên  $\mathbf{w}_t^T \mathbf{x}_i < 0$ . Cũng vì  $y_i = 1$  nên  $y_i \|\mathbf{x}_i\|_2^2 = \|\mathbf{x}_i\|_2^2 \geq 1$  (chú ý  $\mathbf{x}_i$  là một vector đặc trưng mở rộng với một phần tử bằng 1). Từ đó suy ra  $\mathbf{w}_{t+1}^T \mathbf{x}_i > \mathbf{w}_t^T \mathbf{x}_i$ . Nói cách khác,  $-y_i \mathbf{w}_{t+1}^T \mathbf{x}_i < -y_i \mathbf{w}_t^T \mathbf{x}_i$ . Điều tương tự cũng xảy ra với  $y_i = -1$ . Việc này chỉ ra rằng đường thẳng được mô tả bởi  $\mathbf{w}_{t+1}$  có xu hướng khiến hàm mất mát tại điểm bị phân lớp lỗi  $\mathbf{x}_i$  giảm đi. *Chú ý rằng việc này không đảm bảo hàm mất mát trên toàn bộ dữ liệu sẽ giảm, vì rất có thể đường thẳng mới sẽ làm cho một điểm lúc trước được phân lớp đúng trở thành một điểm bị phân lớp sai. Tuy nhiên, thuật toán này được đảm bảo sẽ hội tụ sau một số hữu hạn bước.* Thuật toán perceptron được tóm tắt dưới đây.

#### Thuật toán 2.1: Perceptron

1. Tại thời điểm  $t = 0$ , chọn ngẫu nhiên một vector hệ số  $\mathbf{w}_0$ .
2. Tại thời điểm  $t$ , nếu không có điểm dữ liệu nào bị phân lớp lỗi, dừng thuật toán.
3. Giả sử  $\mathbf{x}_i$  là một điểm bị phân lớp lỗi. Cập nhật

$$\mathbf{w}_{t+1} = \mathbf{w}_t + y_i \mathbf{x}_i$$

4. Thay đổi  $t = t + 1$  rồi quay lại Bước 2.

### 2.2.4 Chứng minh hội tụ

Gọi  $\mathbf{w}^*$  là một nghiệm của bài toán phân lớp nhị phân với hai lớp linearly separable. Nghiệm này luôn tồn tại khi hai lớp là linearly separable. Ta sẽ chứng minh Thuật toán 2.1 kết thúc sau một số hữu hạn bước bằng phản chứng.

Giả sử ngược lại, tồn tại một  $\mathbf{w}$  mà Thuật toán 2.1 chạy mãi mãi. Trước hết ta thấy rằng, với  $\alpha > 0$  bất kỳ, nếu  $\mathbf{w}^*$  là nghiệm,  $\alpha\mathbf{w}^*$  cũng là nghiệm của bài toán. Xét dãy số không âm  $u_\alpha(t) = \|\mathbf{w}_t - \alpha\mathbf{w}^*\|_2^2$ . Theo giả thiết phản chứng, tồn tại một điểm bị phân lớp lỗi khi dùng nghiệm  $\mathbf{w}_t$ . Giả sử đó là điểm  $\mathbf{x}_i$  với nhãn  $y_i$ . Ta có

$$u_\alpha(t+1) = \|\mathbf{w}_{t+1} - \alpha\mathbf{w}^*\|_2^2 \quad (2.9)$$

$$= \|\mathbf{w}_t + y_i\mathbf{x}_i - \alpha\mathbf{w}^*\|_2^2 \quad (2.10)$$

$$= \|\mathbf{w}_t - \alpha\mathbf{w}^*\|_2^2 + y_i^2\|\mathbf{x}_i\|_2^2 + 2y_i\mathbf{x}_i^T(\mathbf{w}_t - \alpha\mathbf{w}^*) \quad (2.11)$$

$$< u_\alpha(t) + \|\mathbf{x}_i\|_2^2 - 2\alpha y_i \mathbf{x}_i^T \mathbf{w}^* \quad (2.12)$$

Dấu nhỏ hơn ở dòng cuối là vì  $y_i^2 = 1$  và  $2y_i\mathbf{x}_i^T \mathbf{w}_t < 0$ . Nếu tiếp tục đặt

$$\beta^2 = \max_{i=1,2,\dots,N} \|\mathbf{x}_i\|_2^2, \quad \gamma = \min_{i=1,2,\dots,N} y_i \mathbf{x}_i^T \mathbf{w}^* \quad (2.13)$$

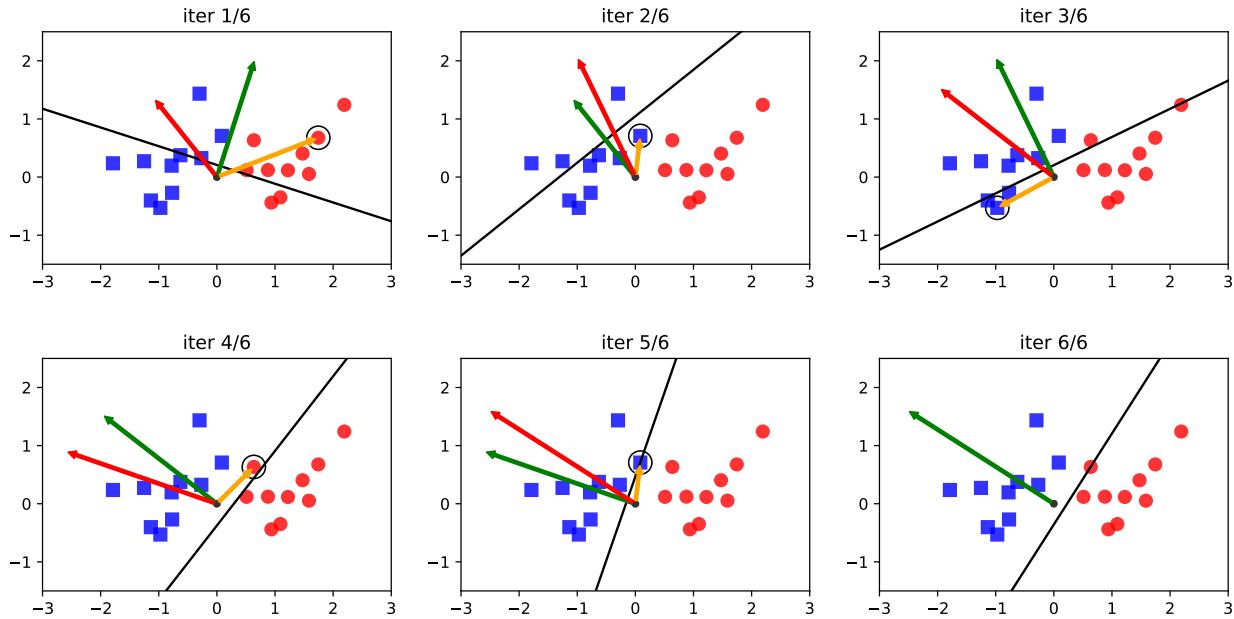
và chọn  $\alpha = \frac{\beta^2}{\gamma}$ , ta sẽ có  $0 \leq u_\alpha(t+1) < u_\alpha(t) + \beta^2 - 2\alpha\gamma = u_\alpha(t) - \beta^2$ . Ta có thể chọn giá trị này vì  $\alpha$  và (2.12) đúng với  $\alpha$  bất kỳ. Điều này chỉ ra rằng nếu luôn có điểm bị phân lớp lỗi thì dãy  $u_\alpha(t)$  là một dãy giảm, bị chặn dưới bởi 0, và phần tử sau kém phần tử trước ít nhất một lượng là  $\beta^2 > 0$ . Điều vô lý này chứng tỏ đến một lúc nào đó sẽ không còn điểm nào bị phân lớp lỗi. Nói cách khác, thuật toán perceptron hội tụ sau một số hữu hạn bước.

## 2.3 Ví dụ và minh họa trên Python

Thuật toán 2.1 có thể được triển khai như sau:

```
import numpy as np
def predict(w, X):
    ''' predict label of each row of X, given w
    X: a 2-d numpy array of shape (N, d), each row is a datapoint
    w_init: a 1-d numpy array of shape (d) '''
    return np.sign(X.dot(w))

def perceptron(X, y, w_init):
    ''' perform perceptron learning algorithm
    X: a 2-d numpy array of shape (N, d), each row is a datapoint
    y: a 1-d numpy array of shape (N), label of each row of X. y[i] = 1/-1
    w_init: a 1-d numpy array of shape (d) '''
    w = w_init
    while True:
        pred = predict(w, X)
        # find indexes of misclassified points
        mis_idxs = np.where(np.equal(pred, y) == False)[0]
        # number of misclassified points
        num_mis = mis_idxs.shape[0]
        if num_mis == 0: # no more misclassified points
            return w
        # random pick one misclassified point
        random_id = np.random.choice(mis_idxs, 1)[0]
        # update w
        w = w + y[random_id]*X[random_id]
```



**Hình 2.3:** Minh họa thuật toán perceptron. Các điểm màu lam thuộc lớp 1, các điểm màu đỏ thuộc lớp  $-1$ . Tại mỗi vòng lặp, đường thẳng màu đen là đường ranh giới. Vector màu lục là  $w_t$ . Điểm được khoanh tròn là một điểm bị phân lớp lỗi  $x_i$ . Vector màu cam thể hiện vector  $x_i$ . Vector màu đỏ chính là  $w_{t+1}$ . Nếu  $y_i = 1$  (màu lam), vector màu đỏ bằng tổng hai vector kia. Nếu  $y_i = -1$ , vector màu đỏ bằng hiệu hai vector kia.

Trong đó, hàm `predict(w, X)` dự đoán nhãn của mỗi hàng của  $\mathbf{x}$  dựa trên công thức (2.2). Hàm `perceptron(X, y, w_init)` thực hiện thuật toán PLA với tập dữ liệu  $\mathbf{X}$ , nhãn  $\mathbf{y}$  và nghiệm ban đầu  $w_{\text{init}}$ .

Để kiểm tra đoạn code trên, ta áp dụng nó vào một ví dụ với dữ liệu trong không gian hai chiều như dưới đây.

```

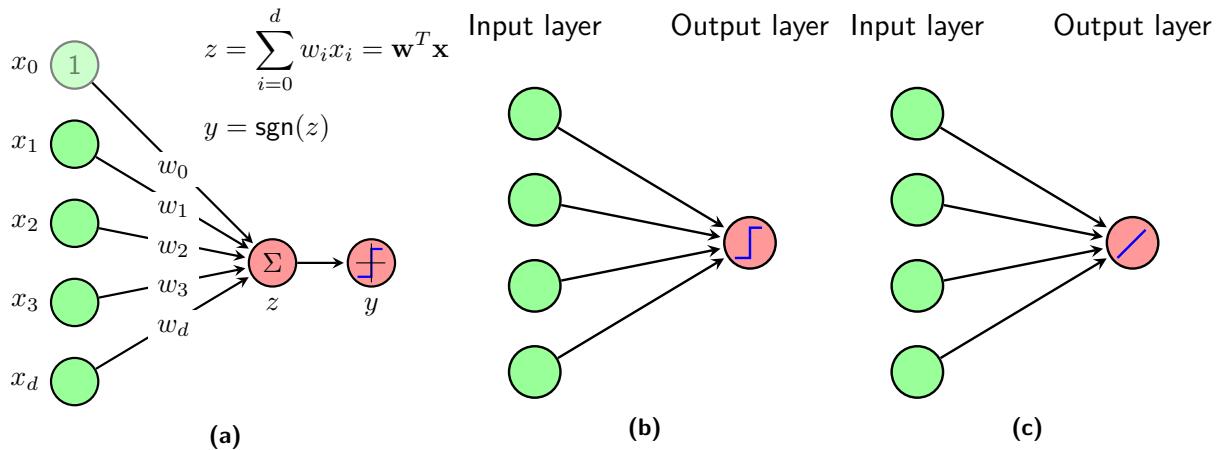
means = [[-1, 0], [1, 0]]
cov = [[.3, .2], [.2, .3]]
N = 10
X0 = np.random.multivariate_normal(means[0], cov, N)
X1 = np.random.multivariate_normal(means[1], cov, N)

X = np.concatenate((X0, X1), axis = 0)
y = np.concatenate((np.ones(N), -1*np.ones(N) ))

Xbar = np.concatenate((np.ones((2*N, 1)), X), axis = 1)
w_init = np.random.randn(Xbar.shape[1])
w = perceptron(Xbar, y, w_init)

```

Mỗi lớp có 10 phần tử, là các vector ngẫu nhiên lấy theo phân phối chuẩn có ma trận hiệp phương sai  $cov$  và vector kỳ vọng được lưu trong `means`. Hình 2.3 minh họa nghiệm sau mỗi vòng lặp. Ta thấy rằng perceptron hội tụ sau chỉ sáu vòng lặp.



**Hình 2.4:** Biểu diễn perceptron và linear regression dưới dạng neural network. (a) perceptron đầy đủ, (b) perceptron thu gọn, (c) linear regression thu gọn.

## 2.4 Mô hình Neural Network đầu tiên

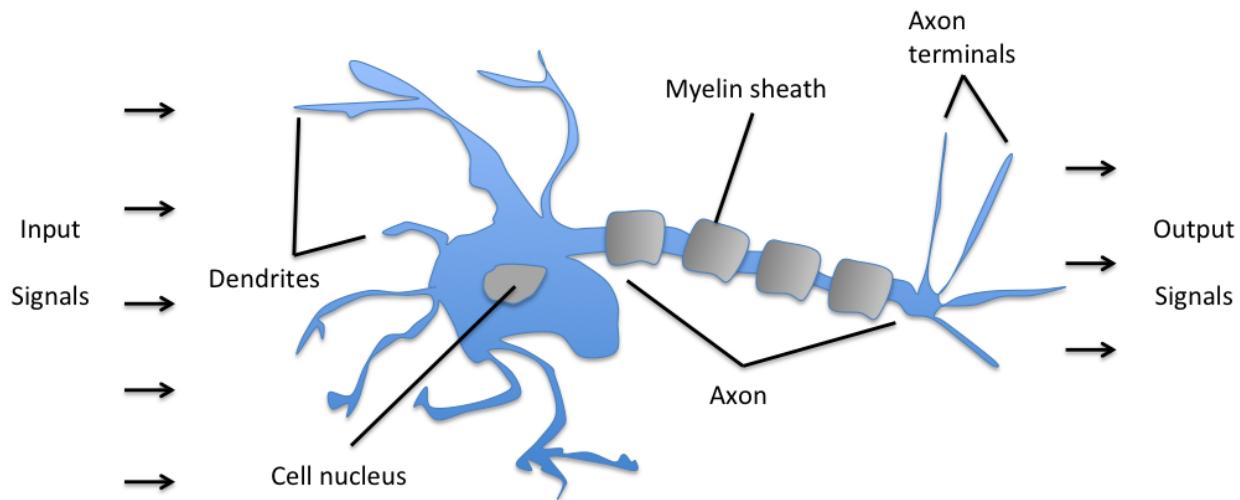
Hàm số dự đoán đầu ra của perceptron  $\text{label}(\mathbf{x}) = \text{sgn}(\mathbf{w}^T \mathbf{x})$  có thể được mô tả trên Hình 2.4a. Đây chính là dạng đơn giản của neural network.

Đầu vào của network  $\mathbf{x}$  được minh họa bằng các *node* màu lục với node  $x_0$  luôn luôn bằng 1. Tập hợp các node màu lục được gọi là *tầng đầu vào (input layer)*. Số node trong input layer là  $d + 1$ . Đôi khi node  $x_0 = 1$  này đổi khi được ẩn đi. Các *trọng số*  $w_0, w_1, \dots, w_d$  được gán vào các mũi tên đi tới node  $z = \sum_{i=0}^d w_i x_i = \mathbf{w}^T \mathbf{x}$ . Node  $y = \text{sgn}(z)$  là *output* của network. Ký hiệu hình chữ Z ngược màu lam trong node  $y$  thể hiện đồ thị của hàm  $\text{sgn}$ . Hàm  $y = \text{sgn}(z)$  còn được gọi là *hàm kích hoạt (activation function)*. Dữ liệu đầu vào được đặt vào input layer, lấy tổng có trọng số lưu vào biến  $z$  rồi đi qua hàm kích hoạt để có kết quả ở  $y$ . Đây chính là dạng đơn giản nhất của một neural network. Perceptron cũng có thể được vẽ giản lược như Hình 2.4b, với ẩn ý rằng dữ liệu ở input layer được lấy tổng có trọng số trước khi đi qua hàm lấy dấu  $y = \text{sgn}(z)$ .

Các neural network có thể có một hoặc nhiều node ở output tạo thành một *tầng đầu ra (output layer)*, hoặc có thể có thêm các layer trung gian giữa *input layer* và *output layer*, được gọi là *tầng ẩn (hidden layer)*. Các neural network thường có nhiều hidden layer và các layer có thể có các hàm kích hoạt khác nhau. Chúng ta sẽ đi sâu vào các neural network với nhiều hidden layer ở Chương 5. Trước đó, chúng ta sẽ tìm hiểu các neural network đơn giản hơn không có hidden layer.

Để ý rằng nếu ta thay *activation function* bởi hàm *identity*  $y = z$ , ta sẽ có một neural network mô tả linear regression như Hình 2.4c. Với đường thẳng chéo màu xanh thể hiện đồ thị hàm số  $y = z$ . Các trực tọa độ đã được lược bỏ.

Mô hình perceptron ở trên khá giống với một node nhỏ của dây thần kinh sinh học như Hình 2.5.



Schematic of a biological neuron.

**Hình 2.5:** Cấu trúc của một neuron thần kinh sinh học (Nguồn: [Single-Layer Neural Networks and Gradient Descent](#)).

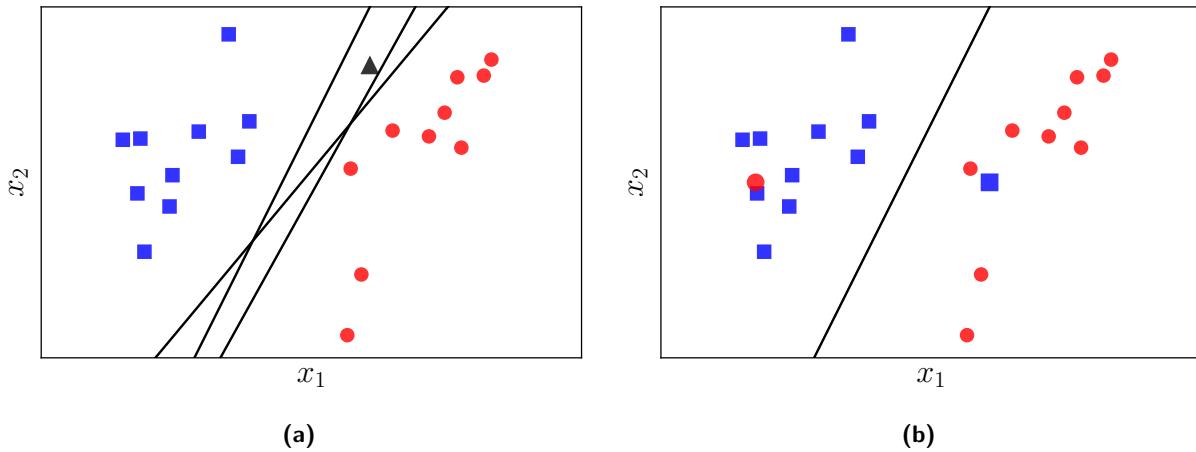
Dữ liệu từ nhiều dây thần kinh (tương tự như  $x_i$ ) đi về một *cell nucleus*. Cell nucleus đóng vai trò như bộ lấy tổng có trọng số  $\sum_{i=0}^d w_i x_i$ . Thông tin này sau đó được tổng hợp (tương tự như hàm kích hoạt) và đưa ra ở output. Tên gọi *neural network* hoặc *artificial neural network* được khởi nguồn từ đây.

## 2.5 Thảo Luận

**PLA có thể cho vô số nghiệm khác nhau.** Nếu hai lớp dữ liệu là linearly separable thì có vô số đường thẳng ranh giới của hai lớp dữ liệu đó như trên Hình 2.6a. Tất cả các đường thẳng màu đen đều có thể đóng vài trò là đường ranh giới. Tuy nhiên, các đường khác nhau sẽ quyết định điểm hình tam giác thuộc các lớp khác nhau. Trong các đường đó, đường nào là tốt nhất? Và định nghĩa “tốt nhất” được hiểu theo nghĩa nào? Các câu hỏi này sẽ được thảo luận kỹ hơn trong Chương ??.

**PLA đòi hỏi hai lớp dữ liệu phải linearly separable.** Hình 2.6b mô tả hai lớp dữ liệu *tương đối* linearly separable. Mỗi lớp có một điểm coi như *nhiều* nằm lẩn trong các điểm của lớp kia. PLA sẽ không làm việc, tức không bao giờ dừng lại, trong trường hợp này vì với mọi đường thẳng ranh giới, luôn có ít nhất hai điểm bị phân lớp lỗi.

Trong một chừng mực nào đó, đường thẳng màu đen vẫn có thể coi là một nghiệm tốt vì nó đã giúp phân loại chính xác hầu hết các điểm. Việc không hội tụ với dữ liệu *gần* linearly separable chính là một nhược điểm lớn của PLA.



**Hình 2.6:** Với bài toán phân lớp nhị phân, PLA có thể (a) cho vô số nghiệm, hoặc (b) vô nghiệm thậm chí với chỉ một nhiễu nhỏ.

Nhược điểm này có thể được khắc phục bằng thuật toán Pocket Algorithm dưới đây.

**Pocket Algorithm [AMMIL12]:** một cách tự nhiên, nếu có một vài *nhiều*, ta sẽ đi tìm một đường thẳng phân chia hai class sao cho có ít điểm bị phân lớp lõi nhất. Việc này có thể được thực hiện thông qua PLA với một chút thay đổi nhỏ:

1. Giới hạn số lượng vòng lặp của PLA. Đặt nghiệm  $\mathbf{w}$  sau vòng lặp đầu tiên và số điểm bị phân lớp lõi vào trong *túi quần* (*pocket*).
2. Mỗi lần cập nhật nghiệm  $\mathbf{w}_t$  mới, ta đếm xem có bao nhiêu điểm bị phân lớp lõi. So sánh số điểm bị phân lớp lõi này với số điểm bị phân lớp lõi trong *pocket*, nếu nhỏ hơn thì lấy nghiệm cũ ra, đặt nghiệm mới này vào. Lặp lại bước này đến khi hết số vòng lặp.

Thuật toán này giống với thuật toán tìm phần tử nhỏ nhất trong một mảng một chiều.

Source code cho Chương này có thể được tìm thấy [tại đây](#).

# Logistic regression

---

## 3.1 Giới thiệu

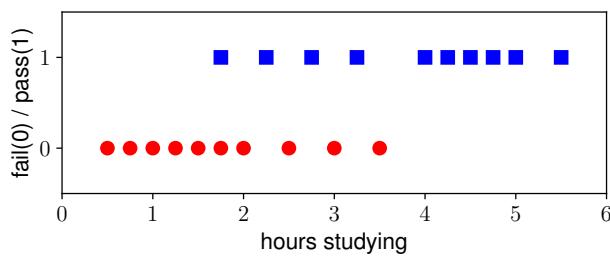
### 3.1.1 Nhắc lại hai mô hình tuyến tính

Hai mô hình tuyến tính đã thảo luận trong cuốn sách này, linear regression và perceptron (PLA), đều có thể viết chung dưới dạng  $y = f(\mathbf{w}^T \mathbf{x})$  với  $f(s)$  là một hàm kích hoạt. Với  $f(s) = s$  trong linear regression, và  $f(s) = \text{sgn}(s)$  trong PLA. Trong linear regression, tích vô hướng  $\mathbf{w}^T \mathbf{x}$  được trực tiếp sử dụng để dự đoán output  $y$ , loại này phù hợp nếu ta cần dự đoán một đầu ra không bị chặn trên và dưới. Trong PLA, đầu ra chỉ nhận một trong hai giá trị 1 hoặc -1, phù hợp với các bài toán phân lớp nhị phân. Trong chương này, chúng ta sẽ thảo luận một mô hình tuyến tính với một hàm kích hoạt khác, thường được áp dụng cho các bài toán phân lớp nhị phân. Trong mô hình này, đầu ra có thể được thể hiện dưới dạng xác suất. Ví dụ, xác suất thi đỗ nếu biết thời gian ôn thi, xác suất ngày mai có mưa dựa trên những thông tin đo được trong ngày hôm nay, v.v.. Mô hình này có tên là *logistic regression*. Mặc dù trong tên có chứa từ *regression*, logistic regression thường được sử dụng nhiều hơn cho các bài toán phân lớp.

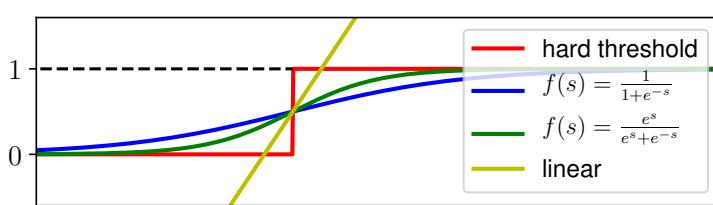
### 3.1.2 Một ví dụ nhỏ

Bảng 3.1: Thời gian ôn thi (Hours) và kết quả thi của 20 sinh viên.

Hours	Pass	Hours	Pass	Hours	Pass	Hours	Pass
0.5	0	0.75	0	1	0	1.25	0
1.5	0	1.75	0	1.75	1	2	0
2.25	1	2.5	0	2.75	1	4	0
3.25	1	3.5	0	4	1	4.25	1
4.5	1	4.75	1	5	1	5.5	1



**Hình 3.1:** Ví dụ về kết quả thi dựa trên số giờ ôn tập. Trục hoành thể hiện thời gian ôn tập của mỗi sinh viên, trục tung gồm hai giá trị 0/fail (các điểm màu đỏ) và 1/pass (các điểm màu xanh).



**Hình 3.2:** Một vài ví dụ về các hàm kích hoạt khác nhau.

Xét một ví dụ về sự liên quan giữa thời gian ôn thi và kết quả thi của 20 sinh viên được cho trong Bảng 3.1. Bài toán đặt ra là từ dữ liệu này hãy xây dựng một mô hình đánh giá khả năng đỗ của một sinh viên dựa trên thời gian ôn thi. Dữ liệu trong Bảng 3.1 được mô tả trên Hình 3.1. Nhìn chung, thời gian học càng nhiều thì khả năng đỗ càng cao. Tuy nhiên, không có một ngưỡng nào để có thể khẳng định rằng mọi sinh viên học nhiều thời gian hơn ngưỡng đó sẽ chắc chắn đỗ. Nói cách khác, dữ liệu của hai lớp này là không linearly separable, và vì vậy PLA sẽ không hữu ích ở đây. Tuy nhiên, thay vì dự đoán chính xác hai giá trị đỗ/trượt, ta có thể dự đoán xác suất để một sinh viên thi đỗ dựa trên thời gian ôn thi.

### 3.1.3 Mô hình logistic regression

Quan sát Hình 3.2 với các hàm kích hoạt  $f(s)$  khác nhau.

- Đường màu vàng biểu diễn một hàm kích hoạt tuyến tính. Đường này không bị chặn nên không phù hợp cho bài toán đang xét với điều ra là một giá trị trong khoảng  $[0, 1]$ .
- Đường màu đỏ tương tự với hàm kích hoạt của PLA với ngưỡng có thể khác không<sup>1</sup>.
- Các đường màu lam và lục phù hợp với bài toán đang xét hơn. Chúng có một vài tính chất quan trọng:
  - Là các hàm số liên tục nhận giá trị thực, bị chặn trong khoảng  $(0, 1)$ .
  - Nếu coi điểm có tung độ là  $1/2$  là ngưỡng, các điểm càng xa ngưỡng về phía bên trái có giá trị càng gần 0, các điểm càng xa ngưỡng về phía phải có giá trị càng gần 1. Điều này khớp với nhận xét rằng học càng nhiều thì xác suất đỗ càng cao và ngược lại.
  - Hai hàm này có đạo hàm mọi nơi, vì vậy có thể được lợi trong việc tối ưu.

<sup>1</sup> Đường này chỉ khác với activation function của PLA ở chỗ hai class là 0 và 1 thay vì -1 và 1.

## Hàm sigmoid và tanh

Trong số các hàm số có ba tính chất nói trên, hàm *sigmoid*:

$$f(s) = \frac{1}{1 + e^{-s}} \triangleq \sigma(s) \quad (3.1)$$

được sử dụng nhiều nhất, vì nó bị chấn trong khoảng  $(0, 1)$ . Thêm nữa,

$$\lim_{s \rightarrow -\infty} \sigma(s) = 0; \quad \lim_{s \rightarrow +\infty} \sigma(s) = 1 \quad (3.2)$$

Thú vị hơn,

$$\sigma'(s) = \frac{e^{-s}}{(1 + e^{-s})^2} = \frac{1}{1 + e^{-s}} \frac{e^{-s}}{1 + e^{-s}} = \sigma(s)(1 - \sigma(s)) \quad (3.3)$$

Với đạo hàm đơn giản, hàm sigmoid được sử dụng rộng rãi trong neural network.

Ngoài ra, hàm *tanh* cũng hay được sử dụng:  $\tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}}$ . Hàm số này nhận giá trị trong khoảng  $(-1, 1)$  nhưng có thể dễ dàng đưa nó về khoảng  $(0, 1)$ . Bạn đọc có thể chứng minh được rằng  $\tanh(s) = 2\sigma(2s) - 1$ .

Hàm sigmoid có thể được thực hiện trên Python như sau.

```
def sigmoid(S):
    """
    S: an numpy array
    return sigmoid function of each element of S
    """
    return 1/(1 + np.exp(-S))
```

Các bạn sẽ sớm thấy được hàm sigmoid được khám phá ra như thế nào.

## 3.2 Hàm mất mát và phương pháp tối ưu

### 3.2.1 Xây dựng hàm mất mát

Với các mô hình với hàm mất mát màu lam và lục như trên, ta có thể giả sử rằng xác suất để một điểm dữ liệu  $\mathbf{x}$  rơi vào lớp thứ nhất là  $f(\mathbf{w}^T \mathbf{x})$  và rơi vào lớp còn lại là  $1 - f(\mathbf{w}^T \mathbf{x})$ :

$$p(y_i = 1 | \mathbf{x}_i; \mathbf{w}) = f(\mathbf{w}^T \mathbf{x}_i) \quad (3.4)$$

$$p(y_i = 0 | \mathbf{x}_i; \mathbf{w}) = 1 - f(\mathbf{w}^T \mathbf{x}_i) \quad (3.5)$$

trong đó  $p(y_i = 1 | \mathbf{x}_i; \mathbf{w})$  được hiểu là xác suất xảy ra sự kiện đầu ra  $y_i = 1$  khi biết tham số mô hình  $\mathbf{w}$  và dữ liệu đầu vào  $\mathbf{x}_i$ .

Mục đích cuối cùng là tìm các hệ số  $\mathbf{w}$  sao cho với các điểm dữ liệu ứng với  $y_i = 1$ ,  $f(\mathbf{w}^T \mathbf{x}_i)$  gần với 1, và ngược lại. Ký hiệu  $z_i = f(\mathbf{w}^T \mathbf{x}_i)$ , hai biểu thức (3.4) và (3.5) có thể được viết chung dưới dạng

$$p(y_i | \mathbf{x}_i; \mathbf{w}) = z_i^{y_i} (1 - z_i)^{1-y_i} \quad (3.6)$$

Biểu thức này tương đương với hai biểu thức (3.4) và (3.5) ở trên vì khi  $y_i = 1$ , phần thứ hai của vế phải sẽ bằng 1, khi  $y_i = 0$ , phần thứ nhất sẽ bằng 1. Chúng ta muốn mô hình gần với dữ liệu đã cho nhất, tức xác suất này đạt giá trị cao nhất.

Xét toàn bộ tập huấn luyện với ma trận dữ liệu  $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N] \in \mathbb{R}^{d \times N}$  và vector đầu ra tương ứng với mỗi cột  $\mathbf{y} = [y_1, y_2, \dots, y_N]$ . Ta cần giải bài toán tối ưu

$$\mathbf{w} = \arg \max_{\mathbf{w}} p(\mathbf{y} | \mathbf{X}; \mathbf{w}) \quad (3.7)$$

Đây chính là một bài toán maximum likelihood estimation với tham số mô hình  $\mathbf{w}$  cần được ước lượng. Giả sử rằng các điểm dữ liệu được sinh ra một cách ngẫu nhiên độc lập với nhau, ta có thể viết

$$p(\mathbf{y} | \mathbf{X}; \mathbf{w}) = \prod_{i=1}^N p(y_i | \mathbf{x}_i; \mathbf{w}) = \prod_{i=1}^N z_i^{y_i} (1 - z_i)^{1-y_i} \quad (3.8)$$

Lấy logarit tự nhiên, đổi dấu, và lấy trung bình, ta thu được hàm số

$$J(\mathbf{w}) = -\frac{1}{N} \log p(\mathbf{y} | \mathbf{X}; \mathbf{w}) = -\frac{1}{N} \sum_{i=1}^N (y_i \log z_i + (1 - y_i) \log(1 - z_i)) \quad (3.9)$$

với chú ý rằng  $z_i$  là một hàm số của  $\mathbf{w}$  và  $\mathbf{x}_i$ . Hàm số này chính là hàm mất mát của logistic regression. Ta cần đi tìm  $\mathbf{w}$  để  $J(\mathbf{w})$  đạt giá trị nhỏ nhất.

### 3.2.2 Tối ưu hàm mất mát

Bài toán tối ưu hàm mất mát của logistic regression có thể được giải quyết bằng stochastic gradient descent (SGD). Tại mỗi vòng lặp,  $\mathbf{w}$  sẽ được cập nhật dựa trên một điểm dữ liệu ngẫu nhiên. Hàm mất mát của logistic regression với chỉ một điểm dữ liệu  $(\mathbf{x}_i, y_i)$  và đạo hàm của nó lần lượt là

$$J(\mathbf{w}; \mathbf{x}_i, y_i) = -(y_i \log z_i + (1 - y_i) \log(1 - z_i)) \quad (3.10)$$

$$\frac{\partial J(\mathbf{w}; \mathbf{x}_i, y_i)}{\partial \mathbf{w}} = -\left(\frac{y_i}{z_i} - \frac{1 - y_i}{1 - z_i}\right) \frac{\partial z_i}{\partial \mathbf{w}} = \frac{z_i - y_i}{z_i(1 - z_i)} \frac{\partial z_i}{\partial \mathbf{w}} \quad (3.11)$$

ở đây ta đã sử dụng quy tắc chuỗi để tính đạo hàm với  $z_i = f(\mathbf{w}^T \mathbf{x})$ . Để cho biểu thức này trở nên gọn và đẹp hơn, ta sẽ tìm hàm  $z = f(\mathbf{w}^T \mathbf{x})$  sao cho mẫu số bị triệt tiêu.

Nếu đặt  $s = \mathbf{w}^T \mathbf{x}$ , ta sẽ có

$$\frac{\partial z_i}{\partial \mathbf{w}} = \frac{\partial z_i}{\partial s} \frac{\partial s}{\partial \mathbf{w}} = \frac{\partial z_i}{\partial s} \mathbf{x}_i \quad (3.12)$$

Một cách tự nhiên, ta sẽ tìm hàm số  $z = f(s)$  sao cho:

$$\frac{\partial z}{\partial s} = z(1 - z) \quad (3.13)$$

để triệt tiêu mẫu số trong biểu thức (3.11). Phương trình vi phân này không quá phức tạp. Thật vậy, (3.13) tương đương với

$$\begin{aligned} & \frac{\partial z}{z(1 - z)} = \partial s \\ \Leftrightarrow & \left(\frac{1}{z} + \frac{1}{1 - z}\right)\partial z = \partial s \\ \Leftrightarrow & \log z - \log(1 - z) = s + C \\ \Leftrightarrow & \log \frac{z}{1 - z} = s + C \\ \Leftrightarrow & \frac{z}{1 - z} = e^{s+C} \\ \Leftrightarrow & z = e^{s+C}(1 - z) \\ \Leftrightarrow & z = \frac{e^{s+C}}{1 + e^{s+C}} = \frac{1}{1 + e^{-s-C}} = \sigma(s + C) \end{aligned}$$

với  $C$  là một hằng số. Đơn giản chọn  $C = 0$ , ta được  $z = f(\mathbf{w}^T \mathbf{x}) = \sigma(s)$ . Đây chính là lý do hàm sigmoid được ra đời. Logistic regression với hàm kích hoạt là hàm sigmoid được sử dụng phổ biến nhất. Mô hình này còn có tên là *logistic sigmoid regression*. Dôi khi, khi nói logistic regression, ta ngầm hiểu rằng đó chính là logistic sigmoid regression.

Thay (3.12) và (3.13) vào (3.11) ta thu được

$$\frac{\partial J(\mathbf{w}; \mathbf{x}_i, y_i)}{\partial \mathbf{w}} = (z_i - y_i)\mathbf{x}_i \quad (3.14)$$

Và công thức cập nhật nghiệm cho logistic sigmoid regression sử dụng SGD là

$$\mathbf{w} \leftarrow \mathbf{w} - \eta(z_i - y_i)\mathbf{x}_i = \mathbf{w} - \eta(\sigma(\mathbf{w}^T \mathbf{x}_i) - y_i)\mathbf{x}_i \quad (3.15)$$

với  $\eta$  là một learning rate dương.

### 3.2.3 Logistic regression với weight decay

Một trong các kỹ thuật phổ biến giúp tránh overfitting với các neural network là sử dụng *weight decay*. Weight decay là một kỹ thuật regularization, trong đó một đại lượng tỉ lệ với bình phương norm 2 của vector hệ số được cộng vào hàm mất mát để hạn chế độ lớn của các hệ số. Hàm mất mát trở thành

$$\bar{J}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N \left( -y_i \log z_i - (1 - y_i) \log(1 - z_i) + \frac{\lambda}{2} \|\mathbf{w}\|_2^2 \right) \quad (3.16)$$

Công thức cập nhật cho  $\mathbf{w}$  với hàm này cũng đơn giản vì phần regularization có đạo hàm đơn giản:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta ((\sigma(\mathbf{w}^T \mathbf{x}_i) - y_i)\mathbf{x}_i + \lambda \mathbf{w}) \quad (3.17)$$

### 3.3 Triển khai thuật toán trên Python

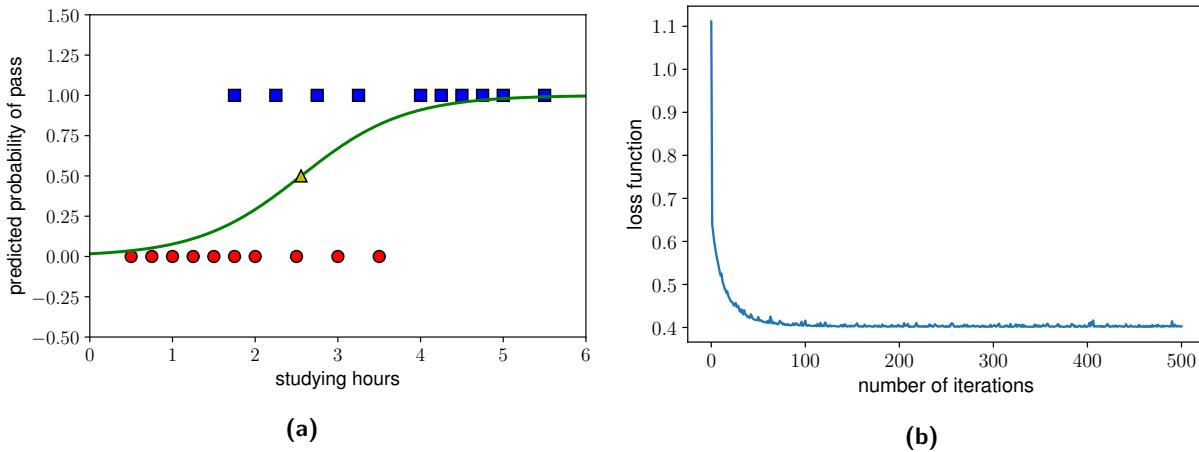
Hàm ước lượng xác suất cho mỗi điểm dữ liệu và hàm tính giá trị hàm mất mát với weight decay có thể được thực hiện như sau trong Python.

```
def prob(w, X):
    """
    X: a 2d numpy array of shape (N, d). N datapoint, each with size d
    w: a 1d numpy array of shape (d)
    """
    return sigmoid(X.dot(w))

def loss(w, X, y, lam):
    """
    X, w as in prob
    y: a 1d numpy array of shape (N). Each elem = 0 or 1
    """
    z = prob(w, X)
    return -np.mean(y*np.log(z) + (1-y)*np.log(1-z)) + 0.5*lam/X.shape[0]*np.sum(w*w)
```

Từ công thức (3.17), ta có thể thực hiện thuật toán tìm  $w$  cho logistic regression như sau.

```
def logistic_regression(w_init, X, y, lam = 0.001, lr = 0.1, nepoches = 2000):
    # lam - reg parameter, lr - learning rate, nepoches - number of epochs
    N, d = X.shape[0], X.shape[1]
    w = w_init
    loss_hist = [loss(w_init, X, y, lam)] # store history of loss in loss_hist
    ep = 0
    while ep < nepoches:
        ep += 1
        mix_ids = np.random.permutation(N)
        for i in mix_ids:
            xi = X[i]
            yi = y[i]
            zi = sigmoid(xi.dot(w))
            w = w - lr*((zi - yi)*xi + lam*w)
        loss_hist.append(loss(w, X, y, lam))
        if np.linalg.norm(w - w_old)/d < 1e-6:
            break
        w_old = w
    return w, loss_hist
```



**Hình 3.3:** Nghiệm của logistic regression cho bài toán dự đoán kết quả thi dựa trên thời gian học. (a) Đường màu lục thể hiện xác suất thi đỗ dựa trên thời gian học. Điểm tam giác vàng thể hiện ngưỡng ra quyết định đỗ/trượt. Điểm này có thể thay đổi tùy vào bài toán. (b) Giá trị của hàm mất mát qua các vòng lặp. Hàm mất mát giảm rất nhanh và hội tụ rất sớm.

### 3.3.1 Logistic regression cho ví dụ ban đầu

Áp dụng vào bài toán ví dụ ban đầu.

```
X = np.array([[0.50, 0.75, 1.00, 1.25, 1.50, 1.75, 1.75, 2.00, 2.25, 2.50,
              2.75, 3.00, 3.25, 3.50, 4.00, 4.25, 4.50, 4.75, 5.00, 5.50]]).T
y = np.array([0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1])

# bias trick
Xbar = np.concatenate((X, np.ones((N, 1))), axis = 1)
w_init = np.random.randn(Xbar.shape[1])
lam = 0.0001
w, loss_hist = logistic_regression(w_init, Xbar, y, lam, lr = 0.05, nepoches = 500)
print(w)
print(loss(w, Xbar, y, lam))
```

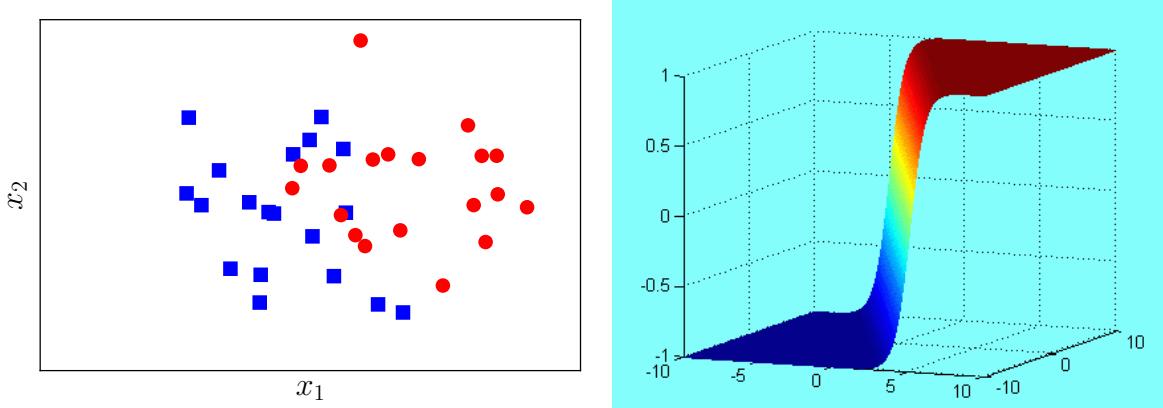
Kết quả:

```
Solution of Logistic Regression: [ 1.54337021 -4.06486702]
Final loss: 0.402446724975
```

Từ đây ta có thể rút ra xác suất thi đỗ dựa trên công thức:

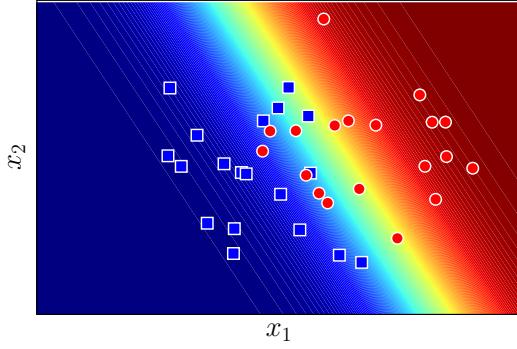
```
probability_of_pass = sigmoid(1.54 - 4.06*hours_of_studying)
```

Nghiệm của mô hình logistic regression và giá trị hàm mất mát qua mỗi epoch được mô tả trên Hình 3.3.



(a) Dữ liệu cho bài toán phân lớp trong không gian hai chiều. (b) Đồ thị hàm sigmoid trong không gian hai chiều.

**Hình 3.4:** Ví dụ về dữ liệu trong không gian hai chiều và hàm sigmoid trong không gian đó.



**Hình 3.5:** Ví dụ về Logistic Regression với dữ liệu hai chiều. Vùng màu đỏ càng đậm thể hiện xác suất thuộc lớp dữ liệu đỏ càng cao, vùng màu xanh càng đậm thể hiện xác suất thuộc lớp dữ liệu xanh càng thấp - tức xác suất thuộc lớp dữ liệu xanh càng cao. Vùng biên giữa hai lớp thể hiện các điểm thuộc vào mỗi lớp với xác suất thấp hơn (độ tin cậy thấp hơn).

### 3.3.2 Ví dụ với dữ liệu hai chiều

Chúng ta xét thêm một ví dụ nhỏ trong không gian hai chiều. Giả sử có hai lớp xanh và đỏ với dữ liệu được phân bố như Hình 3.4a. Với dữ liệu đầu vào nằm trong không gian hai chiều, hàm sigmoid có dạng như thác nước như Hình 3.4b.

Kết quả tìm được khi áp dụng mô hình logistic regression được minh họa như Hình 3.5 với màu nền thể hiện xác suất điểm đó thuộc class đỏ. Màu xanh đậm thể hiện giá trị 0, màu đỏ đậm thể hiện giá trị rất gần với 1.

Khi phải lựa chọn một ranh giới thay vì xác suất, ta quan sát thấy các đường thẳng nằm trong khu vực lục và vàng là một lựa chọn hợp lý. Ta sẽ chứng minh ở phần sau rằng tập hợp các điểm cho cùng xác suất đều ra tạo thành một siêu phẳng.

Source code cho chương này có thể được tìm thấy tại <https://goo.gl/9e7sPF>.

Cách sử dụng logistic regression trong thư viện scikit-learn có thể được tìm thấy tại <https://goo.gl/BJLJNx>.

### 3.4 Tính chất của logistic regression

#### 1. Logistic Regression được sử dụng nhiều trong các bài toán Classification.

Mặc dù trong tên có từ regression, logistic regression được sử dụng nhiều trong các bài toán classification. Sau khi tìm được mô hình, việc xác định class  $y$  cho một điểm dữ liệu  $\mathbf{x}$  được xác định bằng việc so sánh hai biểu thức xác suất

$$P(y = 1|\mathbf{x}; \mathbf{w}); \quad P(y = 0|\mathbf{x}; \mathbf{w}) \quad (3.18)$$

Nếu biểu thức thứ nhất lớn hơn, ta kết luận điểm dữ liệu thuộc class 1, và ngược lại. Vì tổng hai biểu thức này luôn bằng một nên một cách gọn hơn, ta chỉ cần xác định xem  $P(y = 1|\mathbf{x}; \mathbf{w})$  lớn hơn 0.5 hay không.

#### 2. Đường ranh giới tạo bởi logistic regression là một siêu phẳng

Thật vậy, giả sử những điểm có xác suất đầu ra lớn hơn 0.5 được coi là thuộc vào lớp có nhãn là 1. Tập hợp các điểm này là nghiệm của bất phương trình

$$P(y = 1|\mathbf{x}; \mathbf{w}) > 0.5 \Leftrightarrow \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}}} > 0.5 \Leftrightarrow e^{-\mathbf{w}^T \mathbf{x}} < 1 \Leftrightarrow \mathbf{w}^T \mathbf{x} > 0 \quad (3.19)$$

Nói cách khác, tập hợp các điểm thuộc lớp 1 tạo thành *nửa không gian* (*halfspace*)  $\mathbf{w}^T \mathbf{x} > 0$ , tập hợp các điểm thuộc lớp 0 tạo thành nửa không gian còn lại. Ranh giới giữa hai lớp chính là siêu phẳng  $\mathbf{w}^T \mathbf{x} = 0$ .

Chính vì điều này, logistic regression được coi như một bộ phân lớp tuyến tính.

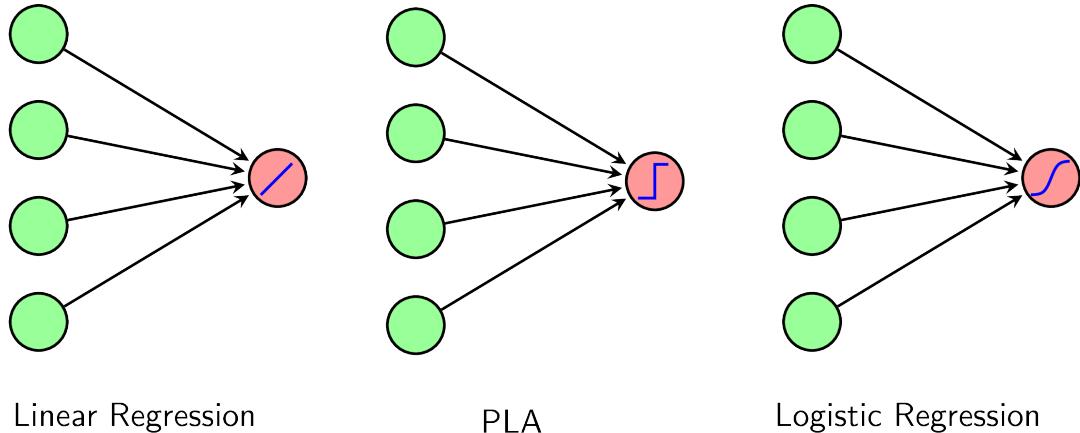
#### 3. Logistic regression không yêu cầu giả thiết linearly separable.

Một điểm cộng của logistic regression so với PLA là nó không cần có giả thiết dữ liệu hai lớp là linearly separable. Tuy nhiên, ranh giới tìm được vẫn có dạng tuyến tính. Vì vậy, mô hình này chỉ phù hợp với loại dữ liệu mà hai lớp gần với linearly separable, tức chỉ có một vài điểm dữ liệu phá vỡ tính linearly separable của hai lớp.

#### 4. Nguồn ra quyết định có thể thay đổi.

Hàm dự đoán đầu ra của các điểm dữ liệu mới có thể được viết như sau.

```
def predict(w, X, threshold = 0.5):
    """
    predict output of each row of X
    X: a numpy array of shape (N, d)
    threshold: a threshold between 0 and 1
    return a 1d numpy array, each element is 0 or 1
    """
    res = np.zeros(X.shape[0])
    res[np.where(prob(w, X) > threshold)[0]] = 1
    return res
```



**Hình 3.6:** Biểu diễn linear regression, PLA, và logistic regression dưới dạng neural network.

Trong các ví dụ đã nêu, ngưỡng ra quyết định đều được lấy tại 0.5. Trong nhiều trường hợp, ngưỡng này có thể được thay đổi. Ví dụ, việc xác định các giao dịch là lừa đảo của một công ty tín dụng là rất quan trọng. Việc phân lớp nhầm một giao dịch lừa đảo thành một giao dịch thông thường gây ra hậu quả nghiêm trọng hơn chiều ngược lại. Trong bài toán đó, ngưỡng phân loại có thể giảm xuống còn 0.3. Nghĩa là các giao dịch được dự đoán là lừa đảo với xác suất lớn hơn 0.3 sẽ được xếp vào lớp lừa đảo và nên được tiếp tục đánh giá bằng các bước khác.

- Khi biểu diễn theo neural networks, linear regression, PLA, và logistic regression có thể được biểu diễn như trên Hình 3.6. Sự khác nhau chỉ nằm ở lựa chọn hàm kích hoạt.

### 3.5 Bài toán phân biệt hai chữ số viết tay

Chúng ta cùng làm một ví dụ thực tế hơn với bài toán phân biệt hai chữ số 0 và 1 trong bộ cơ sở dữ liệu MNIST. Trong mục này, chúng ta sẽ sử dụng class **LogisticRegression** trong thư viện scikit-learn. Trước tiên, ta khai báo các thư viện và download bộ cơ sở dữ liệu MNIST.

```
import numpy as np
from sklearn.datasets import fetch_mldata
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
mnist = fetch_mldata('MNIST original', data_home='../../data/')
N, d = mnist.data.shape
print('Total {:d} digits, each has {:d} pixels.'.format(N, d))
```

Kết quả:

```
Total 70000 digits, each has 784 pixels.
```

Có tổng cộng 70000 điểm dữ liệu trong tập dữ liệu MNIST, mỗi điểm là một mảng 784 phần tử tương ứng với 784 pixel. Mỗi chữ số từ 0 đến 9 chiếm khoảng 10%. Chúng ta sẽ lấy ra tất cả các điểm ứng với chữ số 0 và 1, sau đó lấy ra ngẫu nhiên 2000 điểm làm tập kiểm thử, phần còn lại đóng vai trò tập huấn luyện.

```
X_all = mnist.data
y_all = mnist.target

X0 = X_all[np.where(y_all == 0)[0]] # all digit 0
X1 = X_all[np.where(y_all == 1)[0]] # all digit 1
y0 = np.zeros(X0.shape[0]) # class 0 label
y1 = np.ones(X1.shape[0]) # class 1 label

X = np.concatenate((X0, X1), axis = 0) # all digits
y = np.concatenate((y0, y1)) # all labels

# split train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=2000)
```

Tiếp theo, ta xây dựng mô hình logistic regression trên tập huấn luyện và dự đoán nhãn của các điểm trong tập kiểm thử. Kết quả này được so sánh với nhãn thật của mỗi điểm dữ liệu để tính độ chính xác của bộ phân lớp trên tập kiểm thử.

```
model = LogisticRegression(C = 1e5) # C is inverse of lam
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
print("Accuracy %.2f %%" % (100*accuracy_score(y_test, y_pred.tolist())))
```

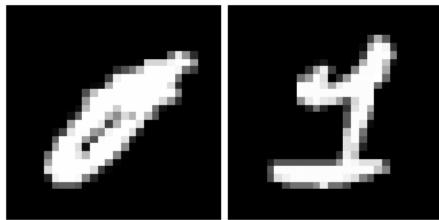
Tuyệt vời, gần như 100% được phân loại chính xác. Điều này là dễ hiểu vì hai chữ số 0 và 1 khác nhau rất nhiều.

Tiếp theo, ta cùng đi tìm những ảnh bị phân loại sai và hiển thị chúng.

```
mis = np.where((y_pred - y_test) != 0)[0]
Xmis = X_test[mis, :]

from display_network import *
filename = 'mnist_mis.pdf'
with PdfPages(filename) as pdf:
    plt.axis('off')
    A = display_network(Xmis.T, 1, Xmis.shape[0])
    f2 = plt.imshow(A, interpolation='nearest')
    plt.gray()
    pdf.savefig(bbox_inches='tight')
    plt.show()
```

Chỉ có hai chữ số bị phân lớp lối được cho trên Hình 3.7. Trong đó, chữ số 0 bị phân lớp lối là dễ hiểu vì nó trông rất giống chữ số 1.



**Hình 3.7:** Các chữ số bị phân lớp lỗi trong bài toán phân lớp nhị phân với hai chữ số 0 và 1.

Bạn đọc có thể xem thêm ví dụ về bài toán xác định giới tính dựa trên ảnh khuôn mặt tại <https://goo.gl/9V8wdD>.

### 3.6 Bộ phân lớp nhị phân cho bài toán phân lớp đa lớp

Logistic regression được áp dụng cho các bài toán phân lớp nhị phân. Các bài toán phân lớp thực tế có thể có nhiều hơn hai lớp dữ liệu rất nhiều, được gọi là bài toán *phân lớp đa lớp* (*multi-class classification*). Với một vài kỹ thuật nhỏ, ta có thể áp dụng logistic regression cho các bài toán phân lớp đa lớp.

Có ít nhất bốn cách để áp dụng logistic regression hay các bộ phân lớp nhị phân vào các bài toán phân lớp đa lớp.

#### 3.6.1 One-vs-one

Xây dựng rất nhiều các bộ phân lớp nhị phân cho từng cặp hai lớp dữ liệu. Bộ thứ nhất phân biệt lớp thứ nhất và thứ hai, bộ thứ hai phân biệt lớp thứ nhất và lớp thứ ba, v.v.. Dữ liệu mới được đưa vào tất cả các bộ phân lớp nhị phân nêu trên. Kết quả cuối cùng có thể được xác định bằng cách xem lớp nào mà điểm dữ liệu đó được phân vào nhiều nhất. Hoặc với logistic regression thì ta có thể tính *tổng xác suất* mà điểm dữ liệu đó rơi vào mỗi lớp. Như vậy, nếu có  $C$  lớp thì số bộ phân lớp cần dùng là  $\frac{C(C-1)}{2}$ . Đây là một con số lớn, cách làm này không lợi về tính toán. Hơn nữa, nếu một chữ số thực ra là chữ số 1, nhưng lại được đưa vào bộ phân lớp giữa các chữ số 5 và 6, thì cả hai khả năng đều không hợp lý.

#### 3.6.2 Phân tầng

Cách làm như **one-vs-one** sẽ mất rất nhiều thời gian huấn luyện vì có quá nhiều bộ phân lớp cần được xây dựng. Một cách giúp giảm số bộ phân lớp nhị phân là *phân tầng* (*hierarchical*). Ý tưởng của phương pháp này có thể được thấy qua ví dụ sau.

Giả sử ta có bài toán phân lớp 4 chữ số 4, 5, 6, 7 trong MNIST. Vì chữ số 4 và 7 khá giống nhau, chữ số 5 và 6 khá giống nhau nên trước tiên ta xây dựng bộ phân lớp [4, 7] vs [5, 6].

Sau đó xây dựng thêm hai bộ  $4 \text{ vs } 7$  và  $5 \text{ vs } 6$ . Tổng cộng, ta cần ba bộ phân lớp. Chú ý rằng có nhiều cách chia khác nhau, ví dụ  $[4, 5, 6] \text{ vs } 7$ ,  $[4, 5] \text{ vs } 6$ , rồi  $4 \text{ vs } 5$ . Ưu điểm của kỹ thuật này là nó sử dụng ít bộ phân lớp tuyến tính hơn *one-vs-one*. Hạn chế lớn nhất của nó là việc nếu chỉ một bộ phân lớp cho kết quả sai thì kết quả cuối cùng chắc chắn sẽ sai. Ví dụ, nếu một ảnh chứa chữ số 5 bị phân lớp lỗi bởi bộ phân lớp đầu tiên, nó sẽ bị phân sang nhánh  $[4, 7]$ . Cuối cùng, nó sẽ bị phân vào lớp 4 hoặc 7, cả hai đều không chính xác.

### 3.6.3 Binary coding

Có một cách giảm số binary classifiers hơn nữa là *binary coding*, tức mã hóa output của mỗi lớp bằng một số nhị phân. Ví dụ, nếu có bốn lớp dữ liệu thì các lớp được mã hóa là 00, 01, 10, và 11. Với cách làm này, số bộ phân lớp nhị phân cần xây dựng chỉ là  $m = \lceil \log_2(C) \rceil$  trong đó  $C$  là số lớp,  $[a]$  là số nguyên nhỏ nhất không nhỏ hơn  $a$ . Bộ thứ nhất đi tìm bit đầu tiên của output (đã được mã hóa nhị phân), bộ thứ hai sẽ đi tìm bit thứ hai, v.v.. Cách làm này sử dụng một số lượng nhỏ nhất các bộ phân lớp tuyến tính. Tuy nhiên, nó có một hạn chế rất lớn là chỉ cần một bit bị phân loại sai sẽ dẫn đến dữ liệu bị phân loại sai. Hơn nữa, nếu số lớp không phải là lũy thừa của hai, mã nhị phân nhận được có thể là một giá trị không tương ứng với lớp nào.

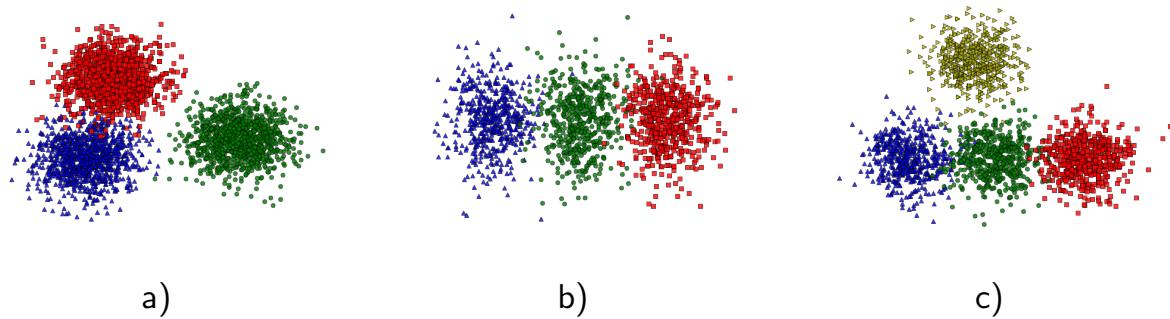
### 3.6.4 one-vs-rest hay one-hot coding

Kỹ thuật được sử dụng nhiều nhất là **one-vs-rest** (một số tài liệu gọi là **ove-vs-all**, **one-against-rest**, hoặc **one-against-all**) . Cụ thể, nếu có  $C$  lớp thì ta sẽ xây dựng  $C$  bộ phân lớp nhị phân, mỗi bộ tương ứng với một lớp. Bộ thứ nhất giúp phân biệt lớp thứ nhất với các lớp còn lại, tức xem một điểm có thuộc lớp thứ nhất hay không, hoặc xác suất để một điểm rơi vào lớp đó là bao nhiêu. Tương tự như thế, bộ thứ hai sẽ phân biệt lớp thứ hai với các lớp còn lại, v.v.. Kết quả cuối cùng có thể được xác định bằng cách xác định lớp mà một điểm rơi vào với xác suất cao nhất.

Logistic regression trong thư viện sklearn có thể được dùng trực tiếp để áp dụng vào các bài toán phân lớp đa lớp với kỹ thuật **one-vs-rest**. Với bài toán MNIST, để dùng logistic regression kết hợp với one-vs-rest (mặc định trong scikit-learn), ta có thể làm như sau.

```
# all class
X_train, X_test, y_train, y_test = train_test_split(X_all, y_all, test_size=10000)
model = LogisticRegression(C = 1e5) # C is inverse of lam
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("Accuracy %.2f %%" % (100*accuracy_score(y_test, y_pred.tolist())))
```

Kết quả thu được khoảng 91.7%. Đây vẫn là một kết quả quá thấp so với con số 99.7% mà deep learning đã đạt được. Ngay cả K-nearest neighbors cũng đã đạt khoảng 96 %.



**Hình 3.8:** Một số ví dụ về phân phối của các lớp dữ liệu trong bài toán phân lớp đa lớp.

## 3.7 Thảo luận

### 3.7.1 Kết hợp các phương pháp trên

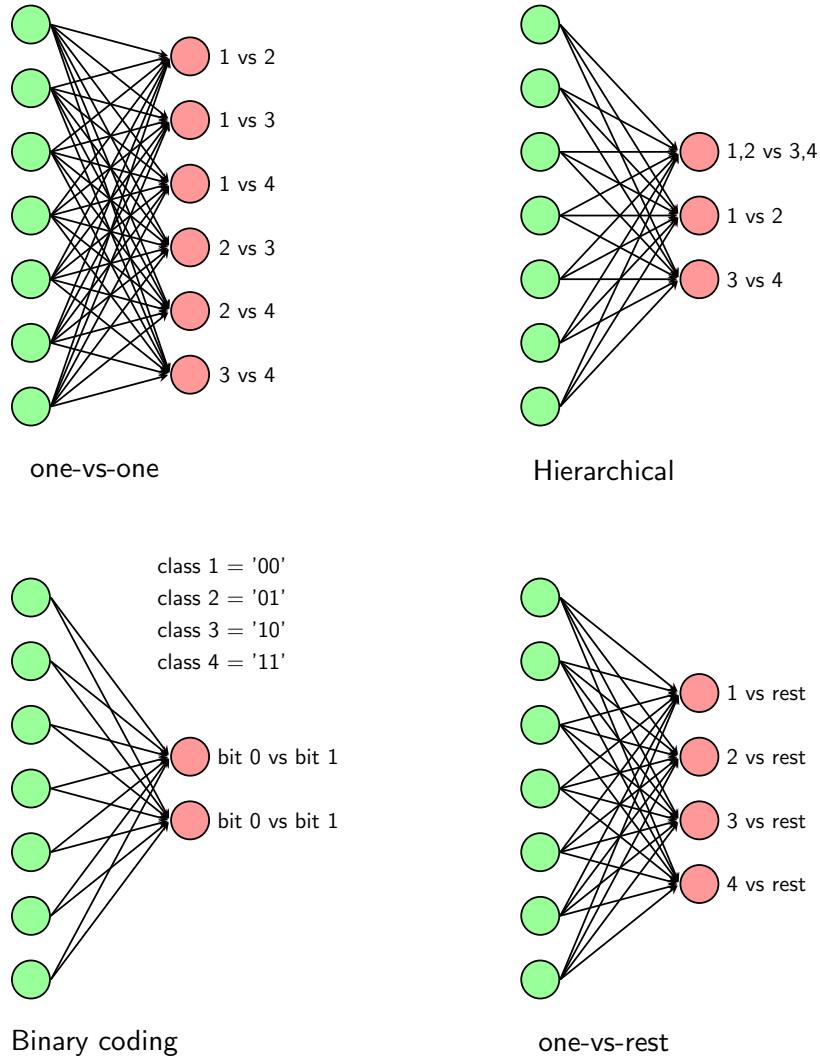
Trong nhiều trường hợp, ta cần phải kết hợp hai hoặc ba trong số bốn kỹ thuật đã đề cập. Xét ba ví dụ trong Hình 3.8.

- Hình 3.8a: cả 4 phương pháp trên đây đều có thể áp dụng được.
- Hình 3.8b: one-vs-rest không phù hợp vì lớp màu lục và lớp lam và lớp đỏ là không (gần) *linearly separable*. Lúc này, one-vs-one hoặc hierarchical phù hợp hơn.
- Hình 3.8c: Tương tự như trên, ba lớp lam, lục, đỏ thẳng hàng nên sẽ không dùng được one-vs-rest. Trong khi đó, one-vs-one vẫn hiệu quả vì từng cặp lớp dữ liệu là *linearly separable*. Tương tự hierarchical cũng làm việc nếu ta phân chia các nhóm một cách hợp lý. Hoặc chúng ta có thể kết hợp nhiều phương pháp. Ví dụ: dùng one-vs-rest để tìm *đỏ* với *không đỏ*. Nếu một điểm dữ liệu là *không đỏ*, với ba lớp còn lại, ta lại quay lại trường hợp Hình 3.8a và có thể dùng các phương pháp khác. Nhưng khó khăn vẫn nằm ở việc phân nhóm như thế nào, liệu rằng những lớp nào có thể cho vào cùng một nhóm?

Với bài toán phân lớp đa lớp, nhìn chung các kỹ thuật sử dụng các bộ phân lớp nhị phân đã trở nên ít hiệu quả hơn so với các phương pháp mới. Mời bạn đọc thêm Chương 4 và Chương ?? để tìm hiểu về các bộ phân lớp đa lớp phổ biến nhất hiện nay.

### 3.7.2 Biểu diễn các kỹ thuật đa neuron dưới dạng neural network

Lấy ví dụ với bài toán có bốn lớp dữ liệu 1, 2, 3, 4; ta có thể biểu diễn các mô hình được đề cập trong Mục 3.6 dưới dạng neural network như trên Hình 3.9. Các node màu đỏ thể hiện đầu ra là một trong hai giá trị.



**Hình 3.9:** Mô hình neural networks cho các kỹ thuật sử dụng các bộ phân lớp nhị phân cho bài toán phân lớp đa lớp.

Các network này đều có nhiều node ở output layer, và một vector trọng số  $w$  bây giờ đã trở thành *ma trận trọng số*  $W$  mà mỗi cột của nó tương ứng với vector trọng số của một node output. Việc tối ưu đồng thời các bộ phân lớp nhị phân trong mỗi network cũng được tổng quát lên nhờ các phép tính với ma trận. Lúc này, công thức cập nhật của logistic regression

$$\mathbf{w} \leftarrow \mathbf{w} - \eta(z_i - y_i)\mathbf{x}_i \quad (3.20)$$

có thể được tổng quát thành

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \mathbf{x}_i (\mathbf{z}_i - \mathbf{y}_i)^T \quad (3.21)$$

Với  $\mathbf{W}$ ,  $\mathbf{y}_i$ ,  $\mathbf{z}_i$  lần lượt là ma trận trọng số, vector (cột) output *thật* với toàn bộ các bộ phân lớp nhị phân tương ứng với điểm dữ liệu  $\mathbf{x}_i$ , và vector output tìm được của networks tại thời điểm đang xét nếu đầu vào mỗi network là  $\mathbf{x}_i$ . Chú ý rằng vector  $\mathbf{y}_i$  là một vector nhị phân, vector  $\mathbf{z}_i$  gồm các phần tử nằm trong khoảng  $(0, 1)$ .

# Softmax Regression

---

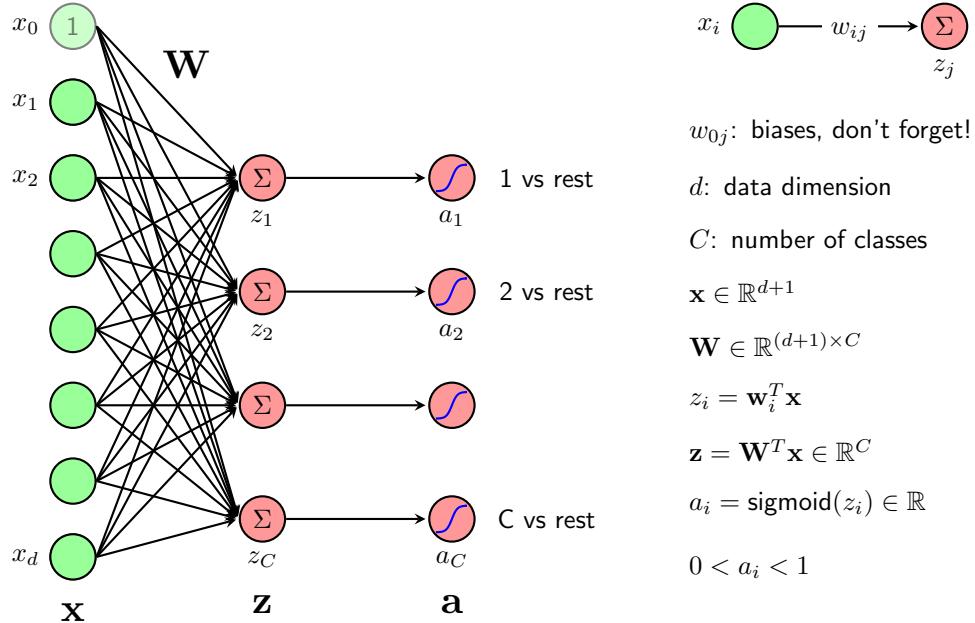
Các bài toán phân lớp thực tế thường có rất nhiều lớp dữ liệu. Như đã thảo luận trong Chương 3, các bộ phân lớp nhị phân tuy có thể kết hợp được với nhau để giải quyết các bài toán này, chúng vẫn có những hạn chế nhất định. Trong chương này, một phương pháp mở rộng của logistic regression có tên là *softmax regression* sẽ được giới thiệu nhằm khắc phục những hạn chế đã đề cập. Một lần nữa, mặc dù trong tên có chứa từ “regression”, softmax regression được sử dụng cho các bài toán phân lớp. Nó cũng chính là một trong những thành phần phổ biến nhất trong các bộ phân lớp hiện nay.

## 4.1 Giới thiệu

Với bài toán phân lớp nhị phân sử dụng logistic regression, đầu ra của neural network là một số thực trong khoảng  $(0, 1)$ , đóng vai trò như là xác suất để đầu vào thuộc một trong hai lớp. Ý tưởng này cũng có thể mở rộng cho bài toán phân lớp đa lớp, ở đó có  $C$  node ở output layer và giá trị mỗi node đóng vai trò như xác suất để đầu vào rơi vào lớp tương ứng. Như vậy, các đầu ra này liên kết với nhau qua việc chúng đều là các số dương và có tổng bằng một. Mô hình softmax regression thảo luận trong chương này đảm bảo tính chất đó.

Nhắc lại kỹ thuật *one-vs-rest* được trình bày trong chương trước được biểu diễn dưới dạng neuron network như trong Hình 4.1. Output layer màu đỏ có thể phân tách thành hai *sublayer* và mỗi thành phần của sublayer thứ hai  $a_i$  chỉ phụ thuộc vào thành phần tương ứng ở sublayer thứ nhất  $z_i$  thông qua hàm sigmoid  $a_i = \sigma(z_i)$ . Các giá trị đầu ra  $a_i$  đều là các số dương nhưng vì không có ràng buộc giữa chúng, tổng của chúng có thể là một số dương bất kỳ.

Chú ý rằng các mô hình linear regression, PLA, và logistic regression chỉ có một node ở output layer. Trong các trường hợp đó, tham số mô hình chỉ là một vector  $\mathbf{w}$ . Trong trường hợp output layer có nhiều hơn một node, tham số mô hình sẽ là tập hợp tham số  $\mathbf{w}_i$  ứng



**Hình 4.1:** Phân lớp đa lớp với logistic regression và one-vs-rest.

với từng node. Lúc này, ta có một *mảng trọng số* (*weight matrix*)  $\mathbf{W} = [\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_C]$ , mỗi cột ứng với một node ở output layer.

## 4.2 Softmax function

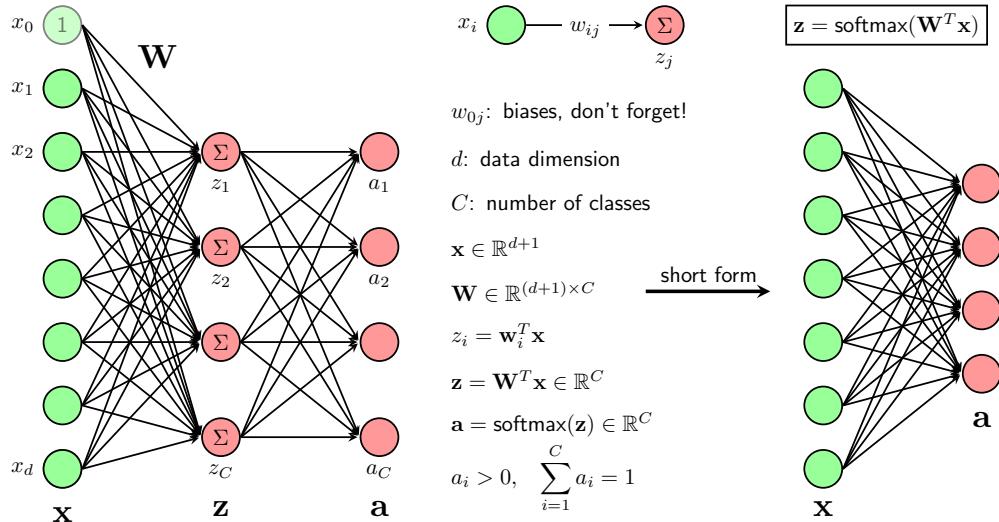
### 4.2.1 Công thức của Softmax function

Chúng ta cần một mô hình xác suất sao cho với mỗi input  $\mathbf{x}$ ,  $a_i$  thể hiện xác suất để input đó rơi vào lớp thứ  $i$ . Vậy điều kiện cần là các  $a_i$  phải dương và tổng của chúng bằng một. Ngoài ra, ta sẽ thêm một điều kiện cũng rất tự nhiên nữa, đó là giá trị  $z_i = \mathbf{w}_i^T \mathbf{x}$  càng lớn thì xác suất dữ liệu rơi vào lớp thứ  $i$  càng cao. Điều kiện cuối này chỉ ra rằng ta cần một quan hệ đồng biến ở đây.

Chú ý rằng  $z_i$  có thể nhận giá trị cả âm và dương vì nó là một tổ hợp tuyến tính của các thành phần của vector đặc trưng  $\mathbf{x}$ . Một hàm số khả vi đơn giản có thể chắc chắn biến  $z_i$  thành một giá trị dương, và hơn nữa, đồng biến, là hàm  $\exp(z_i) = e^{z_i}$ . Điều kiện khả vi để thuận lợi cho việc sử dụng đạo hàm cho việc tối ưu. Điều kiện cuối cùng, tổng các  $a_i$  bằng một có thể được đảm bảo nếu

$$a_i = \frac{\exp(z_i)}{\sum_{j=1}^C \exp(z_j)}, \quad \forall i = 1, 2, \dots, C \quad (4.1)$$

Mỗi quan hệ này, với mỗi  $a_i$  phụ thuộc vào tất cả các  $z_i$ , thỏa mãn tất cả các điều kiện đã xét: dương, tổng bằng một, giữ được thứ tự của  $z_i$ . Hàm số này được gọi là *softmax function*.



**Hình 4.2:** Mô hình softmax regression dưới dạng neural network.

Lúc này, ta có thể coi rằng

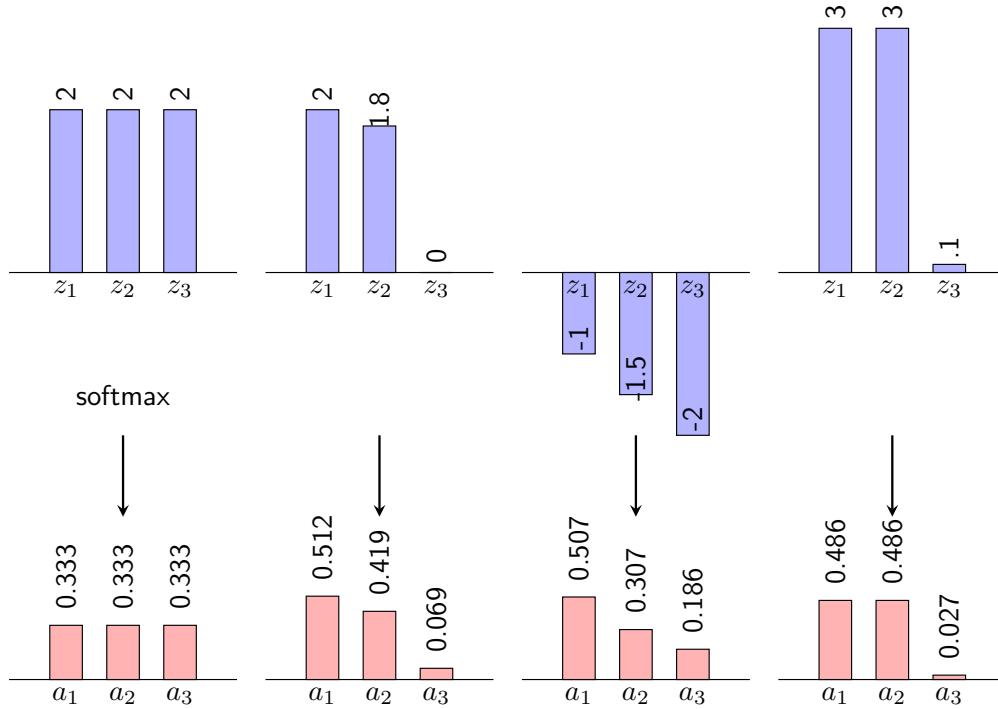
$$p(y_k = i | \mathbf{x}_k; \mathbf{W}) = a_i \quad (4.2)$$

Trong đó,  $p(y = i | \mathbf{x}; \mathbf{W})$  được hiểu là xác suất để một điểm dữ liệu  $\mathbf{x}$  rơi vào lớp thứ  $i$  nếu biết tham số mô hình là ma trận trọng số  $\mathbf{W}$ . Hình 4.2 thể hiện mô hình softmax regression dưới dạng neural network. Sự khác nhau giữa mô hình này và mô hình one-vs-rest nằm ở chỗ nó có các liên kết giữa mọi node của hai sublayer màu đỏ.

#### 4.2.2 Softmax function trong Python

Dưới đây là một đoạn code thực hiện hàm softmax. Đầu vào là một ma trận với mỗi hàng là một vector  $\mathbf{z}$ , đầu ra cũng là một ma trận mà mỗi hàng có giá trị là  $\mathbf{a} = \text{softmax}(\mathbf{z})$ . Các giá trị của  $\mathbf{z}$  còn được gọi là **scores**.

```
import numpy as np
def softmax(Z):
    """
    Compute softmax values for each sets of scores in V.
    each column of V is a set of scores.
    Z: a numpy array of shape (N, C)
    return a numpy array of shape (N, C)
    """
    e_Z = np.exp(Z)
    A = e_Z / e_Z.sum(axis = 1, keepdims = True)
    return A
```



**Hình 4.3:** Một số ví dụ về đầu vào và đầu ra của hàm softmax.

### 4.2.3 Một vài ví dụ

Hình 4.3 mô tả một vài ví dụ về mối quan hệ giữa đầu vào và đầu ra của hàm softmax. Hàng trên màu xanh nhạt thể hiện các scores  $z_i$  với giả sử rằng số lớp dữ liệu là ba. Hàng dưới màu đỏ nhạt thể hiện các giá trị đầu ra  $a_i$  của hàm softmax.

Có một vài quan sát như sau:

- Cột 1: Nếu các  $z_i$  bằng nhau (bằng 2 hoặc một số bất kỳ), thì các  $a_i$  cũng bằng nhau và bằng  $1/3$ .
- Cột 2: Nếu giá trị lớn nhất trong các  $z_i$  là  $z_1$  vẫn bằng 2, thì mặc dù xác suất tương ứng  $a_1$  vẫn là lớn nhất, nó đã thay đổi lên hơn 0.5. Sự chênh lệch ở đầu ra là đáng kể, nhưng thứ tự tương ứng không thay đổi.
- Cột 3: Khi các giá trị  $z_i$  là âm thì các giá trị  $a_i$  vẫn là dương và thứ tự vẫn được đảm bảo.
- Cột 4: Nếu  $z_1 = z_2$ , thì  $a_1 = a_2$ .

Bạn đọc có thể thử với các giá trị khác trực tiếp trên trình duyệt tại <https://goo.gl/pKxQYc>, phần Softmax.

#### 4.2.4 Phiên bản ổn định hơn của softmax function

Khi một trong các  $z_i$  quá lớn, việc tính toán  $\exp(z_i)$  có thể gây ra hiện tượng *tràn số* (*overflow*), ảnh hưởng lớn tới kết quả của hàm softmax. Có một cách khắc phục hiện tượng này bằng cách dựa trên quan sát

$$\frac{\exp(z_i)}{\sum_{j=1}^C \exp(z_j)} = \frac{\exp(-c) \exp(z_i)}{\exp(-c) \sum_{j=1}^C \exp(z_j)} = \frac{\exp(z_i - c)}{\sum_{j=1}^C \exp(z_j - c)} \quad (4.3)$$

với  $c$  là một hằng số bất kỳ. Vậy một phương pháp đơn giản giúp khắc phục hiện tượng overflow là trừ tất cả các  $z_i$  đi một giá trị đủ lớn. Trong thực nghiệm, giá trị đủ lớn này thường được chọn là  $c = \max_i z_i$ . Vậy chúng ta có thể sửa đoạn code cho hàm **softmax** phía trên bằng cách trừ mỗi hàng của ma trận đầu vào  $Z$  đi giá trị lớn nhất trong hàng đó. Ta có phiên bản ổn định hơn là **softmax\_stable**<sup>1</sup>.

```
def softmax_stable(Z):
    """
    Compute softmax values for each sets of scores in Z.
    each row of Z is a set of scores.
    """
    # Z = Z.reshape(Z.shape[0], -1)
    e_Z = np.exp(Z - np.max(Z, axis = 1, keepdims = True))
    A = e_Z / e_Z.sum(axis = 1, keepdims = True)
    return A
```

### 4.3 Hàm mất mát và phương pháp tối ưu

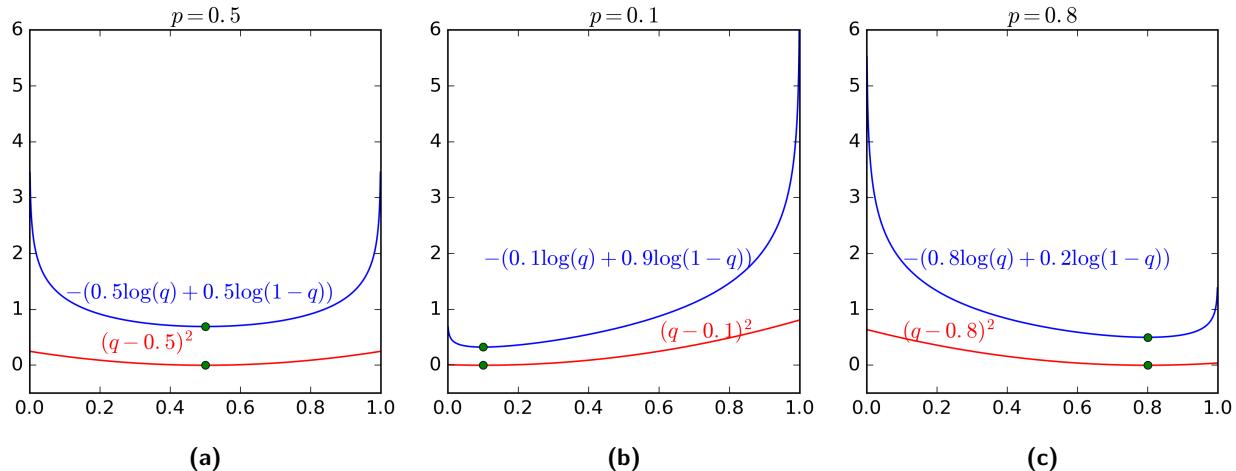
#### 4.3.1 Cross entropy

Dầu ra của softmax network không còn là một giá trị biểu thị lớp cho dữ liệu mà đã trở thành một vector ở dạng one-hot với chỉ một phần tử bằng 1 tại vị trí tương ứng với lớp đó (tính từ 1), các phần tử còn lại bằng 0.

Với mỗi đầu vào  $\mathbf{x}$ , đầu ra tương ứng qua softmax network sẽ là vector  $\mathbf{a} = \text{softmax}(\mathbf{W}^T \mathbf{x})$ ; đầu ra này được gọi là *đầu ra dự đoán*. Trong khi đó, *đầu ra thực sự* của nó là một vector ở dạng one-hot.

Hàm mất mát của softmax regression được xây dựng dựa trên bài toán tối thiểu sự khác nhau giữa *đầu ra dự đoán*  $\mathbf{a}$  và *đầu ra thực sự*  $\mathbf{y}$  (ở dạng one-hot). Khi cả hai là các vector thể hiện xác suất, khoảng cách giữa chúng thường được đo bằng một đại lượng được gọi là *cross entropy*. Một đặc điểm nổi bật của đại lượng này là nếu cố định một vector xác suất, giá trị của nó *đạt giá trị nhỏ nhất khi hai vector xác suất bằng nhau, và rất lớn khi hai vector đó lệch nhau nhiều*.

<sup>1</sup> Xem thêm về cách xử lý mảng numpy trong Python tại <https://fundaml.com>



**Hình 4.4:** So sánh giữa hàm cross entropy và hàm bình phương khoảng cách. Các điểm màu lục thể hiện các giá trị nhỏ nhất của mỗi hàm.

Cross entropy giữa hai vector phân phối  $\mathbf{p}$  và  $\mathbf{q}$  rời rạc được định nghĩa bởi

$$H(\mathbf{p}, \mathbf{q}) = - \sum_{i=1}^C p_i \log q_i \quad (4.4)$$

Hình 4.4 thể hiện rõ ưu điểm của hàm cross entropy so với hàm bình phương khoảng cách Euclid. Đây là ví dụ trong trường hợp  $C = 2$  và  $p_1$  lần lượt nhận các giá trị 0.5, 0.1 và 0.8. Chú ý rằng  $p_2 = 1 - p_1$ . Có hai nhận xét quan trọng sau đây:

1. Giá trị nhỏ nhất của cả hai hàm số đạt được khi  $q = p$  tại hoành độ các điểm màu lục.
2. Quan trọng hơn, hàm cross entropy nhận giá trị rất cao (tức mất mát rất cao) khi  $q$  ở xa  $p$ . Trong khi đó, sự chênh lệch giữa các mất mát ở gần hay xa nghiệm của hàm bình phương khoảng cách  $(q - p)^2$  là ít đáng kể hơn. Về mặt tối ưu, hàm cross entropy sẽ cho nghiệm gần với  $p$  hơn vì những nghiệm ở xa bị phạt rất nặng.

Hai tính chất trên đây khiến cho cross entropy được sử dụng rộng rãi khi tính khoảng cách giữa hai phân phối xác suất. Tiếp theo, chúng ta sẽ chứng minh nhận định sau.

Cho  $\mathbf{p} \in \mathbb{R}_+^C$  là một vector với các thành phần dương và tổng bằng 1. Bài toán tối ưu

$$\begin{aligned} \mathbf{q} &= \arg \min_{\mathbf{q}} H(\mathbf{p}, \mathbf{q}) \\ \text{thoả mãn: } &\sum_{i=1}^C q_i = 1; q_i > 0 \end{aligned}$$

có nghiệm  $\mathbf{q} = \mathbf{p}$ .

Bài toán này có thể giải quyết bằng phương pháp nhân tử Lagrange (xem Phụ lục ??).

Lagrangian của bài toán này là

$$\mathcal{L}(q_1, q_2, \dots, q_C, \lambda) = - \sum_{i=1}^C p_i \log(q_i) + \lambda \left( \sum_{i=1}^C q_i - 1 \right)$$

Ta cần giải hệ phương trình

$$\nabla_{q_1, \dots, q_C, \lambda} \mathcal{L}(q_1, \dots, q_C, \lambda) = 0 \Leftrightarrow \begin{cases} -\frac{p_i}{q_i} + \lambda = 0, & i = 1, \dots, C \\ q_1 + q_2 + \dots + q_C = 1 \end{cases}$$

Từ phương trình thứ nhất ta có  $p_i = \lambda q_i$ . Vì vậy,  $1 = \sum_{i=1}^C p_i = \lambda \sum_{i=1}^C q_i = \lambda \Rightarrow \lambda = 1$ . Điều này tương đương với  $q_i = p_i, \forall i$ .  $\square$

### Chú ý

1. *Hàm cross entropy không có tính đối xứng  $H(\mathbf{p}, \mathbf{q}) \neq H(\mathbf{q}, \mathbf{p})$ . Điều này có thể dễ dàng nhận ra ở việc các thành phần của  $\mathbf{p}$  trong công thức (1) có thể nhận giá trị bằng 0, trong khi đó các thành phần của  $\mathbf{q}$  phải là dương vì  $\log(0)$  không xác định. Chính vì vậy, khi sử dụng cross entropy trong các bài toán phân lớp,  $\mathbf{p}$  là đầu ra thực sự vì đầu ra thực sự vì nó là một vector ở dạng one-hot,  $\mathbf{q}$  là đầu ra dự đoán, khi mà không có xác suất nào tuyệt đối bằng 1 hoặc tuyệt đối bằng 0 (do hàm exp luôn trả về một giá trị dương).*
2. *Khi  $\mathbf{p}$  là một vector ở dạng one-hot, giả sử chỉ có  $p_c = 1$ , khi đó biểu thức cross entropy trở thành  $-\log(p_c)$ . Biểu thức này đạt giá trị nhất nếu  $p_c = 1$ , điều này là không khả thi vì nghiệm này không thuộc miền xác định của bài toán. Tuy nhiên, giá trị cross entropy tiệm cận tới 0 khi  $p_c$  tiến đến 1. Điều này xảy ra khi  $z_c$  rất lớn so với các  $z_i$  còn lại.*

### 4.3.2 Xây dựng hàm mất mát

Trong trường hợp có  $C$  lớp dữ liệu, *mất mát* giữa đầu ra dự đoán và đầu ra thực sự của một điểm dữ liệu  $\mathbf{x}_i$  với label (one-hot)  $\mathbf{y}_i$  được tính bởi

$$J_i(\mathbf{W}) \triangleq J(\mathbf{W}; \mathbf{x}_i, \mathbf{y}_i) = - \sum_{j=1}^C y_{ji} \log(a_{ji}) \quad (4.5)$$

với  $y_{ji}$  và  $a_{ji}$  lần lượt là phần tử thứ  $j$  của vector xác suất  $\mathbf{y}_i$  và  $\mathbf{a}_i$ . Nhắc lại rằng đầu ra  $\mathbf{a}_i$  phụ thuộc vào đầu vào  $\mathbf{x}_i$  và ma trận trọng số  $\mathbf{W}$ . Tới đây, nếu để ý rằng chỉ có đúng một  $j$  sao cho  $y_{ji} = 1, \forall i$ , biểu thức (4.5) chỉ còn lại một số hạng tương ứng với giá trị  $j$  đó. Để tránh việc sử dụng quá nhiều ký hiệu, chúng ta giả sử rằng  $y_i$  là nhãn của điểm dữ liệu  $\mathbf{x}_i$  (các nhãn là các số tự nhiên từ 1 tới  $C$ ), khi đó  $j$  chính bằng  $y_i$ . Sau khi có ký hiệu này, ta có thể viết lại

$$J_i(\mathbf{W}) = - \log(a_{y_i, i}) \quad (4.6)$$

với  $a_{y_i,i}$  là phần tử thứ  $y_i$  của vector  $\mathbf{a}_i$ .

Kết hợp tất cả các cặp dữ liệu  $\mathbf{x}_i, \mathbf{y}_i, i = 1, 2, \dots, N$ , hàm mất mát cho softmax regression được xác định bởi

$$J(\mathbf{W}; \mathbf{X}, \mathbf{Y}) = -\frac{1}{N} \sum_{i=1}^N \log(a_{y_i,i}) \quad (4.7)$$

Ở đây, ma trận trọng số  $\mathbf{W}$  là biến cần tối ưu. Mặc dù hàm mất mát này trông phức tạp, đạo hàm của nó rất gọn. Ta cũng có thể thêm weight decay để tránh overfitting bằng cách cộng thêm một đại lượng tỉ lệ với  $\|\mathbf{W}\|_F^2$ .

$$\bar{J}(\mathbf{W}; \mathbf{X}, \mathbf{Y}) = -\frac{1}{N} \left( \sum_{i=1}^N \log(a_{y_i,i}) + \frac{\lambda}{2} \|\mathbf{W}\|_F^2 \right) \quad (4.8)$$

Trong các mục tiếp theo, chúng ta sẽ làm việc với hàm mất mát (4.7). Việc mở rộng cho hàm mất mát với regularization (4.8) không phức tạp vì đạo hàm của số hạng regularized  $\frac{\lambda}{2} \|\mathbf{W}\|_F^2$  đơn giản là  $\lambda \mathbf{W}$ . Hàm mất mát (4.7) có thể được thực hiện trên Python như sau<sup>2</sup>.

```
def softmax_loss(X, y, W):
    """
    W: 2d numpy array of shape (d, C),
        each column corresponding to one output node
    X: 2d numpy array of shape (N, d), each row is one data point
    y: 1d numpy array -- label of each row of X
    """
    A = softmax_stable(X.dot(W))
    id0 = range(X.shape[0]) # indexes in axis 0, indexes in axis 1 are in y
    return -np.mean(np.log(A[id0, y]))
```

### Chú ý

1. Khi biểu diễn dưới dạng toán học, mỗi điểm dữ liệu là một cột của ma trận  $\mathbf{X}$ ; nhưng khi làm việc với numpy, mỗi điểm dữ liệu được đọc theo  $axis = 0$  của mảng hai chiều  $\mathbf{X}$ . Việc này thống nhất với các thư viện scikit-learn hay tensorflow ở chỗ  $\mathbf{X}[i]$  được dùng để chỉ điểm dữ liệu thứ  $i$ , tính từ  $0$ . Tức là, nếu có  $N$  điểm dữ liệu trong không gian  $d$  chiều thì  $\mathbf{X} \in \mathbb{R}^{d \times N}$ , nhưng  $\mathbf{X}.shape == (N, d)$ .
2.  $\mathbf{W} \in \mathbb{R}^{d \times C}$ ,  $\mathbf{W}.shape == (d, C)$ .
3.  $\mathbf{W}^T \mathbf{X}$  sẽ được biểu diễn bởi  $\mathbf{X}.dot(\mathbf{W})$ , và có  $shape == (N, C)$ .
4. Khi làm việc với phép nhân ma trận hay mảng nhiều chiều trong numpy, ta luôn nhớ chú ý tới kích thước của các ma trận sao cho các phép nhân thực hiện được.

<sup>2</sup> Xem thêm: Truy cập vào nhiều phần tử của mảng hai chiều trong numpy - FundaML <https://goo.gl/SzLDxa>.

### 4.3.3 Tối ưu hàm mất mát

Hàm mất mát sẽ được tối ưu bằng gradient descent, cụ thể là mini-batch gradient descent.

Mỗi lần cập nhật của mini-batch gradient descent được thực hiện trên một *batch* có số phần tử  $1 < k \ll N$ . Để tính được đạo hàm của hàm mất mát theo tập con này, trước hết ta xem xét đạo hàm của hàm mất mát tại một điểm dữ liệu.

Với chỉ một cặp dữ liệu  $(\mathbf{x}_i, \mathbf{y}_i)$ , ta lại dùng (4.5)

$$\begin{aligned} J_i(\mathbf{W}) &= -\sum_{j=1}^C y_{ji} \log(a_{ji}) = -\sum_{j=1}^C y_{ji} \log\left(\frac{\exp(\mathbf{w}_j^T \mathbf{x}_i)}{\sum_{k=1}^C \exp(\mathbf{w}_k^T \mathbf{x}_i)}\right) \\ &= -\sum_{j=1}^C \left( y_{ji} \mathbf{w}_j^T \mathbf{x}_i - y_{ji} \log\left(\sum_{k=1}^C \exp(\mathbf{w}_k^T \mathbf{x}_i)\right) \right) \\ &= -\sum_{j=1}^C y_{ji} \mathbf{w}_j^T \mathbf{x}_i + \log\left(\sum_{k=1}^C \exp(\mathbf{w}_k^T \mathbf{x}_i)\right) \end{aligned} \quad (4.9)$$

Tiếp theo ta sử dụng công thức

$$\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{W}} = \left[ \frac{\partial J_i(\mathbf{W})}{\partial \mathbf{w}_1}, \frac{\partial J_i(\mathbf{W})}{\partial \mathbf{w}_2}, \dots, \frac{\partial J_i(\mathbf{W})}{\partial \mathbf{w}_C} \right] \quad (4.10)$$

Trong đó, đạo hàm theo từng cột của  $\mathbf{w}_j$  có thể tính được dựa theo (4.9) và quy tắc chuỗi tính đạo hàm

$$\begin{aligned} \frac{\partial J_i(\mathbf{W})}{\partial \mathbf{w}_j} &= -y_{ji} \mathbf{x}_i + \frac{\exp(\mathbf{w}_j^T \mathbf{x}_i)}{\sum_{k=1}^C \exp(\mathbf{w}_k^T \mathbf{x}_i)} \mathbf{x}_i \\ &= -y_{ji} \mathbf{x}_i + a_{ji} \mathbf{x}_i = \mathbf{x}_i (a_{ji} - y_{ji}) \\ &= e_{ji} \mathbf{x}_i \quad (\text{với } e_{ji} = a_{ji} - y_{ji}) \end{aligned} \quad (4.11)$$

Giá trị  $e_{ji} = a_{ji} - y_{ji}$  chính là sự sai khác giữa đầu ra dự đoán và đầu ra thực sự tại thành phần thứ  $j$ . Kết hợp (4.10) và (4.11) với  $\mathbf{e}_i = \mathbf{a}_i - \mathbf{y}_i$ , ta có

$$\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{W}} = \mathbf{x}_i [e_{1i}, e_{2i}, \dots, e_{Ci}] = \mathbf{x}_i \mathbf{e}_i^T \quad (4.12)$$

Từ đây ta cũng có thể suy ra rằng

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{N} \sum_{i=1}^N \mathbf{x}_i \mathbf{e}_i^T = \frac{1}{N} \mathbf{X} \mathbf{E}^T \quad (4.13)$$

với  $\mathbf{E} = \mathbf{A} - \mathbf{Y}$ . Công thức tính đạo hàm đơn giản này giúp cho cả batch gradient descent, và mini-batch gradient descent đều có thể dễ dàng được áp dụng. Trong trường hợp mini-batch gradient, giả sử kích thước batch là  $k$ , ký hiệu  $\mathbf{X}_b \in \mathbb{R}^{d \times k}$ ,  $\mathbf{Y}_b \in \{0, 1\}^{C \times k}$ ,  $\mathbf{A}_b \in \mathbb{R}^{C \times k}$  là dữ liệu ứng với một batch, công thức cập nhật cho một batch sẽ là

$$\mathbf{W} \leftarrow \mathbf{W} - \frac{\eta}{N_b} \mathbf{X}_b \mathbf{E}_b^T \quad (4.14)$$

với  $N_b$  là kích thước của mỗi batch. Hàm số tính đạo hàm theo  $\mathbf{W}$  trong Python có thể được thực hiện như sau:

```
def softmax_grad(X, y, W):
    """
    W: 2d numpy array of shape (d, C),
        each column corresponding to one output node
    X: 2d numpy array of shape (N, d), each row is one data point
    y: 1d numpy array -- label of each row of X
    """
    A = softmax_stable(X.dot(W)) # shape of (N, C)
    id0 = range(X.shape[0])
    A[id0, y] -= 1 # A - Y, shape of (N, C)
    return X.T.dot(A) / X.shape[0]
```

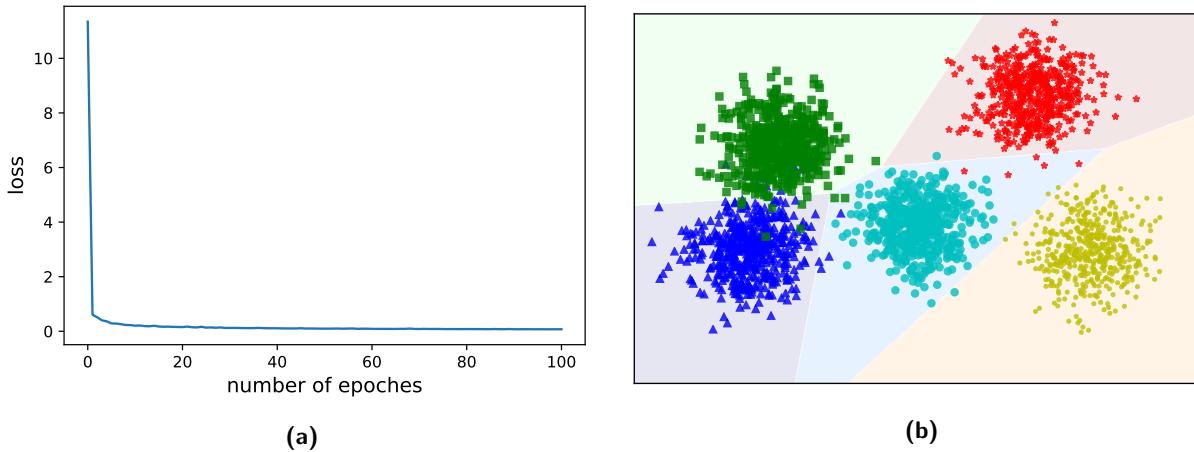
Hàm này đã được kiểm chứng lại bằng hàm `check_grad`.

Từ đó, ta có thể viết hàm số huấn luyện softmax regression như sau:

```
def softmax_fit(X, y, W, lr = 0.01, nepoches = 100, tol = 1e-5, batch_size = 10):
    W_old = W.copy()
    ep = 0
    loss_hist = [loss(X, y, W)] # store history of loss
    N = X.shape[0]
    nbatches = int(np.ceil(float(N)/batch_size))
    while ep < nepoches:
        ep += 1
        mix_ids = np.random.permutation(N) # mix data
        for i in range(nbatches):
            # get the i-th batch
            batch_ids = mix_ids[batch_size*i:min(batch_size*(i+1), N)]
            X_batch, y_batch = X[batch_ids], y[batch_ids]
            W -= lr*softmax_grad(X_batch, y_batch, W) # update gradient descent
        loss_hist.append(softmax_loss(X, y, W))
        if np.linalg.norm(W - W_old)/W.size < tol:
            break
        W_old = W.copy()
    return W, loss_hist
```

Cuối cùng là hàm dự đoán nhãn của các điểm dữ liệu mới. Nhãn của một điểm dữ liệu mới được xác định bằng chỉ số của lớp dữ liệu có xác suất rơi vào cao nhất, và cũng chính là chỉ số của score cao nhất.

```
def pred(W, X):
    """
    predict output of each columns of X . Class of each x_i is determined by
    location of max probability. Note that classes are indexed from 0.
    """
    return np.argmax(X.dot(W), axis =1)
```



**Hình 4.5:** Ví dụ về sử dụng softmax regression cho năm lớp dữ liệu. (a) Nghiệm qua các epoches. (b) Kết quả phân lớp cuối cùng.

#### 4.4 Ví dụ trên Python

Để minh họa ranh giới của các lớp dữ liệu khi sử dụng softmax regression, chúng ta cùng làm một ví dụ nhỏ trong không gian hai chiều với 5 lớp dữ liệu:

```
C = 5      # number of classes
N = 500    # number of points per class
means = [[2, 2], [8, 3], [3, 6], [14, 2], [12, 8]]
cov = [[1, 0], [0, 1]]

X0 = np.random.multivariate_normal(means[0], cov, N)
X1 = np.random.multivariate_normal(means[1], cov, N)
X2 = np.random.multivariate_normal(means[2], cov, N)
X3 = np.random.multivariate_normal(means[3], cov, N)
X4 = np.random.multivariate_normal(means[4], cov, N)

X = np.concatenate((X0, X1, X2, X3, X4), axis = 0) # each row is a datapoint
Xbar = np.concatenate((X, np.ones((X.shape[0], 1))), axis = 1) # bias trick

y = np.asarray([0]*N + [1]*N + [2]*N+ [3]*N + [4]*N)

W_init = np.random.randn(Xbar.shape[1], C)
W, loss_hist = softmax_fit(Xbar, y, W_init, batch_size = 10, nepoches = 100, lr =
0.05)
```

Giá trị của hàm mất mát qua các vòng lặp được cho trên Hình 4.5a. Ta thấy rằng hàm mất mát giảm rất nhanh sau đó hội tụ. Các điểm dữ liệu huấn luyện của mỗi lớp là các điểm có màu khác nhau trong Hình 4.5b. Các phần có màu nền khác nhau là các *lãnh thổ* của mỗi lớp dữ liệu tìm được bằng softmax regression. Ta có thể thấy rằng các đường ranh giới có dạng đường thẳng. Kết quả phân chia lãnh thổ cũng khá tốt, chỉ có một số ít điểm trong tập huấn luyện bị phân lớp sai. Để ý thấy rằng dùng softmax regression tốt hơn rất nhiều so với phương pháp kết hợp các bộ phân lớp nhị phân.

## MNIST với softmax regression trong scikit-learn

Trong scikit-learn, softmax regression được tích hợp trong class `sklearn.linear_model.LogisticRegression`. Như sẽ thấy trong phần thảo luận, logistic regression chính là softmax regression cho bài toán phân lớp nhị phân. Với bài toán phân lớp đa lớp, thư viện này mặc định sử dụng kỹ thuật one-vs-rest. Để sử dụng softmax regression, ta thay đổi thuộc tính `multi_class = 'multinomial'` và `solver = 'lbfgs'`. Ở đây, '`lbfgs`' là một phương pháp tối ưu rất mạnh cũng dựa trên đạo hàm. Trong khuôn khổ của cuốn sách, chúng ta sẽ không thảo luận về phương pháp này.

Quay lại với bài toán phân lớp chữ số viết tay trong cơ sở dữ liệu MNIST. Đoạn code dưới đây thực hiện việc lấy ra 10000 điểm dữ liệu trong số 70000 điểm làm tập kiểm thử, còn lại là tập huấn luyện. Bộ phân lớp được sử dụng là softmax regression.

```
import numpy as np
from sklearn.datasets import fetch_mldata
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
mnist = fetch_mldata('MNIST original', data_home='../../data/')

X = mnist.data
y = mnist.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=10000)

model = LogisticRegression(C = 1e5,
                           solver = 'lbfgs', multi_class = 'multinomial') # C is inverse of lam
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
print("Accuracy %.2f %%" % (100*accuracy_score(y_test, y_pred.tolist())))
```

Kết quả:

Accuracy: 92.19 %

So với kết quả hơn 91.7% của one-vs-rest logistic regression, kết quả của softmax regression đã được cải thiện được một chút. Kết quả thấp như thế này là có thể dự đoán được vì thực ra softmax regression vẫn chỉ tạo ra các đường ranh giới là các đường tuyến tính. Kết quả tốt nhất của bài toán phân loại chữ số trong MNIST hiện nay vào khoảng hơn 99.7%, đạt được bằng một convolutional neural network với rất nhiều hidden layer và layer cuối cùng chính là một softmax regression.

## 4.5 Thảo luận

### 4.5.1 Logistic regression là một trường hợp đặc biệt của softmax regression

Khi  $C = 2$ , softmax regression và logistic regression là giống nhau. Thật vậy, với  $C = 2$ , đầu ra của hàm softmax cho một đầu vào  $\mathbf{x}$  là

$$a_1 = \frac{\exp(\mathbf{w}_1^T \mathbf{x})}{\exp(\mathbf{w}_1^T \mathbf{x}) + \exp(\mathbf{w}_2^T \mathbf{x})} = \frac{1}{1 + \exp((\mathbf{w}_2 - \mathbf{w}_1)^T \mathbf{x})}; \quad a_2 = 1 - a_1 \quad (4.15)$$

Từ đây ta thấy rằng,  $a_1$  có dạng là một hàm sigmoid với vector hệ số  $\mathbf{w} = -(\mathbf{w}_2 - \mathbf{w}_1)$ . Khi  $C = 2$ , bạn đọc cũng có thể thấy rằng hàm matsu của logistic regression và softmax regression là như nhau. Hơn nữa, mặc dù có hai outputs, softmax regression có thể biểu diễn bởi một output vì tổng của hai outputs luôn luôn bằng 1.

Softmax Regression còn có các tên gọi khác là Multinomial Logistic Regression, hay Maximum Entropy Classifier.

### 4.5.2 Ranh giới tạo bởi softmax regression là một mặt tuyến tính

Thật vậy, dựa vào hàm softmax thì một điểm dữ liệu  $\mathbf{x}$  được dự đoán là rơi vào class  $j$  nếu  $a_j \geq a_k, \forall k \neq j$ . Bạn đọc có thể chứng minh được rằng

$$a_j \geq a_k \Leftrightarrow z_j \geq z_k \Leftrightarrow \mathbf{w}_j^T \mathbf{x} \geq \mathbf{w}_k^T \mathbf{x} \Leftrightarrow (\mathbf{w}_j - \mathbf{w}_k)^T \mathbf{x} \geq 0 \quad (4.16)$$

Như vậy, một điểm thuộc lớp thứ  $j$  nếu và chỉ nếu  $(\mathbf{w}_j - \mathbf{w}_k)^T \mathbf{x} \geq 0, \forall k \neq j$ . Như vậy, *lãnh thổ* của mỗi lớp dữ liệu là giao của các nửa không gian. Nói cách khác, đường ranh giới giữa các lớp là các mặt tuyến tính.

### 4.5.3 Softmax Regression là một trong hai classifiers phổ biến nhất

Softmax regression cùng với multi-class support vector machine (Chương ??) là hai bộ phân lớp phổ biến nhất được dùng hiện nay. Softmax regression đặc biệt được sử dụng nhiều trong các deep neural network với rất nhiều hidden layer. Những layer phía trước có thể được coi như một bộ tạo vector đặc trưng, layer cuối cùng thường là một softmax regression.

### 4.5.4 Source code

Source code cho chương này có thể được tìm thấy [tại đây](#).

# Multilayer neural network và Backpropagation

---

## 5.1 Giới thiệu

### 5.1.1 Perceptron cho các hàm logic cơ bản

Chúng ta cùng xét khả năng biểu diễn của perceptron (PLA) cho các bài toán biểu diễn các hàm logic nhị phân: NOT, AND, OR, và XOR<sup>1</sup>. Để có thể sử dụng perceptron (với đầu ra là 1 hoặc -1), chúng ta sẽ thay các giá trị bằng 0 (false) của tại đầu ra của các hàm này bởi -1. Quan sát hàng trên của Hình 5.1, các điểm hình vuông màu xanh là các điểm có nhãn bằng 1, các điểm hình tròn màu đỏ là các điểm có nhãn bằng -1. Hàng dưới của Hình 5.1 là các mô hình perceptron với các hệ số tương ứng.

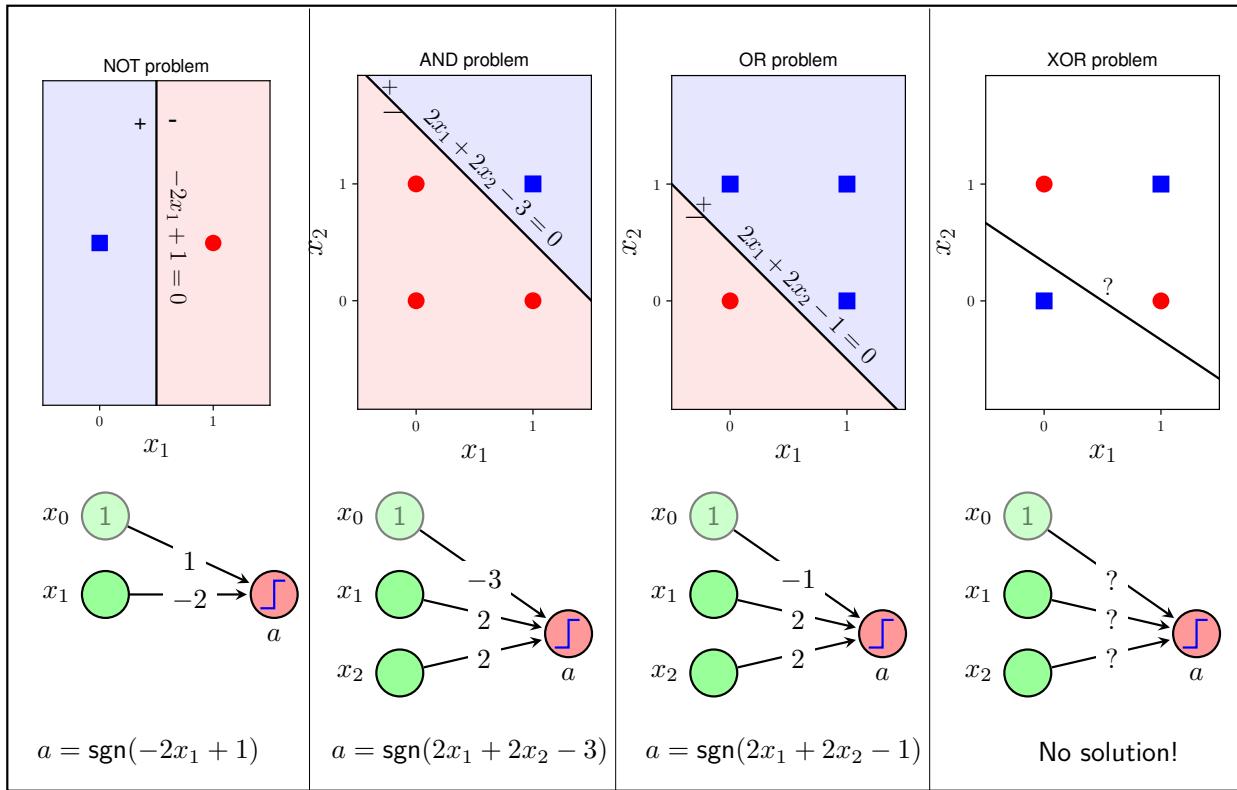
Nhận thấy rằng với các bài toán OR, AND, và OR, dữ liệu hai lớp là linearly separable, vì vậy ta có thể tìm được các hệ số cho perceptron giúp biểu diễn chính xác mỗi hàm số. Chẳng hạn với hàm NOT, khi  $x_1 = 0$ , ta có  $a = \text{sgn}(-2 \times 0 + 1) = 1$ ; khi  $x_1 = 1$ ,  $a = \text{sgn}(-2 \times 1 + 1) = -1$ . Trong cả hai trường hợp, đầu ra dự đoán đều giống với đầu ra thực sự. Bạn đọc có thể tự kiểm chứng các hệ số với hàm AND và OR.

### 5.1.2 Biểu diễn hàm XOR với nhiều perceptron

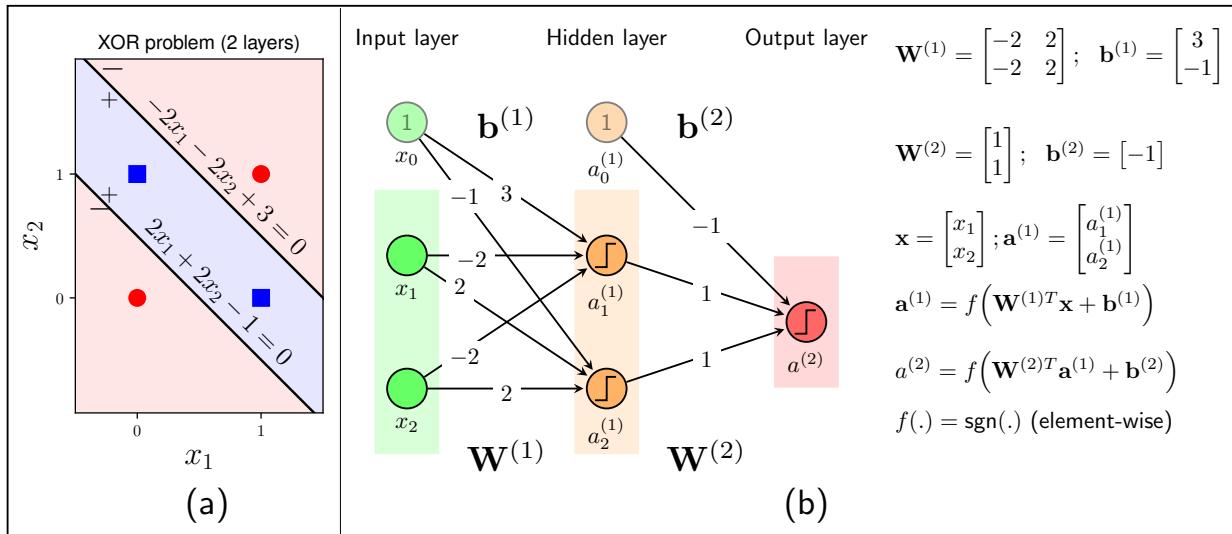
Hàm XOR, vì dữ liệu không linearly separable, không thể biểu diễn bằng một perceptron. Nếu thay perceptron bằng logistic regression tức thay hàm kích hoạt từ hàm sign sang hàm *sigmoid*, ta cũng không tìm được các hệ số thỏa mãn, vì về bản chất, logistic regression (hay cả softmax regression) cũng chỉ tạo ra các ranh giới có dạng tuyến tính. Như vậy là các mô hình neural network chúng ta đã biết không thể biểu diễn được hàm số logic đơn giản này.

---

<sup>1</sup> đầu ra bằng 1 (true) nếu và chỉ nếu hai đầu vào logic khác nhau.



**Hình 5.1:** Biểu diễn các hàm logic cơ bản sử dụng perceptron learning algorithm.



**Hình 5.2:** Ba perceptron biểu diễn hàm XOR.

Nhận thấy rằng nếu cho phép sử dụng hai đường thẳng, bài toán biểu diễn hàm XOR có thể được giải quyết như Hình 5.2. Các hệ số tương ứng với hai đường thẳng trong Hình 5.2a được minh họa trên Hình 5.2b bằng các mũi tên xuất phát từ các điểm màu lục và cam. Đầu ra  $a_1^{(1)}$  bằng 1 với các điểm nằm về phía (+) của đường thẳng  $3 - 2x_1 - 2x_2 = 0$ , bằng

$-1$  với các điểm nằm về phía  $(-)$ . Tương tự, đầu ra  $a_2$  bằng  $1$  với các điểm nằm về phía  $(+)$  của đường thẳng  $-1 + 2x_1 + 2x_2 = 0$ . Như vậy, hai đường thẳng ứng với hai perceptron này tạo ra hai đầu ra tại các node  $a_1^{(1)}, a_2^{(1)}$ . Vì hàm XOR chỉ có một đầu ra nên ta cần làm thêm một bước nữa: coi  $a_1, a_2$  như là đầu vào của một perceptron khác. Trong perceptron mới này, input là các node màu cam (đừng quên bias node luôn có giá trị bằng  $1$ ), đầu ra là node màu đỏ. Các hệ số được cho trên Hình 5.2b. Kiểm tra lại một chút, với các điểm hình vuông xanh (Hình 5.2a),  $a_1^{(1)} = a_2^{(1)} = 1$ , khi đó  $a^{(2)} = \text{sgn}(-1 + 1 + 1) = 1$ . Với các điểm hình tròn đỏ, vì  $a_1^{(1)} = -a_2^{(1)}$  nên  $a^{(2)} = \text{sgn}(-1 + a_1^{(1)} + a_2^{(1)}) = \text{sgn}(-1) = -1$ . Trong cả hai trường hợp, đầu ra dự đoán đều giống với đầu ra thực sự. Vậy, nếu sử dụng ba perceptron tương ứng với các đầu ra  $a_1^{(1)}, a_2^{(1)}, a^{(2)}$ , ta sẽ biểu diễn được hàm XOR. Ba perceptron kể trên được xếp vào hai *layers*. Layer thứ nhất: đầu vào - lục, đầu ra - cam. Layer thứ hai: đầu vào - cam, đầu ra - đỏ. Ở đây, đầu ra của layer thứ nhất chính là đầu vào của layer thứ hai. Tổng hợp lại ta được một mô hình mà ngoài layer đầu vào (lục) và đầu ra (đỏ), ta còn có một layer nữa (cam).

Một neural network với nhiều hơn hai layer còn được gọi là *multilayer neural network*, *multilayer perceptrons* (MLPs), *deep feedforward network* hoặc *feedforward neural network*. Từ *feedforward* được hiểu là dữ liệu đi *thẳng* từ đầu vào tới đầu ra theo các mũi tên mà *không quay* lại ở điểm nào, tức là network có dạng một *acyclic graph* (đồ thị không chứa chu trình kín). Tên gọi *perceptron* ở đây có thể gây nhầm lẫn một chút<sup>2</sup>, vì cụm từ này để chỉ neural network với nhiều layer và mỗi layer không nhất thiết, nếu không muốn nói là rất hiếm khi, là một hoặc nhiều perceptron. Hàm kích hoạt có thể là các hàm phi tuyến khác thay vì hàm sgn.

Cụ thể hơn, một multilayer neural network là một neural network có nhiều layer, làm nhiệm vụ xấp xỉ mối quan hệ giữa các cặp quan hệ  $(\mathbf{x}, \mathbf{y})$  trong tập huấn luyện bằng một hàm số có dạng

$$\mathbf{y} \approx g^{(L)}(g^{(L-1)}(\dots(g^{(2)}(g^{(1)}(\mathbf{x}))))), \quad (5.1)$$

Trong đó, layer thứ nhất đóng vai trò như hàm  $\mathbf{a}^{(1)} \triangleq g^{(1)}(\mathbf{x})$ ; layer thứ hai đóng vai trò như hàm  $\mathbf{a}^{(2)} \triangleq g^{(2)}(g^{(1)}(\mathbf{x})) = f^{(2)}(\mathbf{a}^{(1)})$ , v.v..

Trong phạm vi cuốn sách, chúng ta quan tâm tới các layer đóng vai trò như các hàm có dạng

$$g^{(l)}(\mathbf{a}^{(l-1)}) = f^{(l)}(\mathbf{W}^{(l)T} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}) \quad (5.2)$$

với  $\mathbf{W}^{(l)}, \mathbf{b}^{(l)}$  là ma trận và vector với số chiều phù hợp,  $f^{(l)}$  là một hàm số được gọi là *hàm kích hoạt* (*activation function*).

<sup>2</sup> Geofrey Hinton, *phù thuỷ Deep Learning*, từng thừa nhận trong khoá học “Neural Networks for Machine Learning” (<https://goo.gl/UfdT1t>) rằng “Multilayer Neural Networks should never have been called Multilayer Perceptron. It is partly my fault, and I’m sorry.”.

### Một vài lưu ý:

- Để cho đơn giản, chúng ta sử dụng ký hiệu  $\mathbf{W}^{(l)T}$  để thay cho  $(\mathbf{W}^{(l)})^T$  (ma trận chuyển vị). Trong Hình 5.2b, ký hiệu ma trận  $\mathbf{W}^{(2)}$  được sử dụng, mặc dù đúng ra nó phải là vector, để biểu diễn tổng quát cho trường hợp output layer có thể có nhiều hơn một node. Tương tự với bias  $\mathbf{b}^{(2)}$ .
- Khác với các chương trước về neural networks, khi làm việc với multilayer neural network, ta nên tách riêng phần bias và ma trận hệ số. Điều này đồng nghĩa với việc vector input  $\mathbf{x}$  là vector KHÔNG mở rộng.

Đầu ra của multilayer neural network loại này ứng với một đầu vào  $\mathbf{x}$  có thể được tính theo

$$\mathbf{a}^{(0)} = \mathbf{x} \quad (5.3)$$

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)T} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}, \quad l = 1, 2, \dots, L \quad (5.4)$$

$$\mathbf{a}^{(l)} = f^{(l)}(\mathbf{z}^{(l)}), \quad l = 1, 2, \dots, L \quad (5.5)$$

$$\hat{\mathbf{y}} = \mathbf{a}^{(L)} \quad (5.6)$$

Đây chính là đầu ra dự đoán. Bước này được gói là *feedforward* vì cách tính toán được thực hiện từ đầu đến cuối của network. Hàm mất mát thỏa mãn đạt giá trị nhỏ khi đầu ra này gần với đầu ra thực sự. Tuỳ vào bài toán, là classification hoặc regression, chúng ta cần thiết kế các hàm mất mát phù hợp.

## 5.2 Các ký hiệu và khái niệm

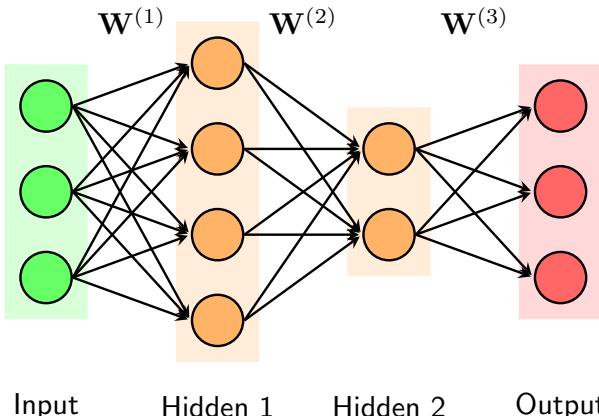
### 5.2.1 Layer

Ngoài *input layer* và *output layer*, một multilayer neural network có thể có nhiều *hidden layer* ở giữa. Các *hidden layer* theo thứ tự từ input layer đến output layer được đánh số thứ tự là *hidden layer 1*, *hidden layer 2*, v.v.. Hình 5.3 là một ví dụ về một multilayer neural network với hai hidden layer.

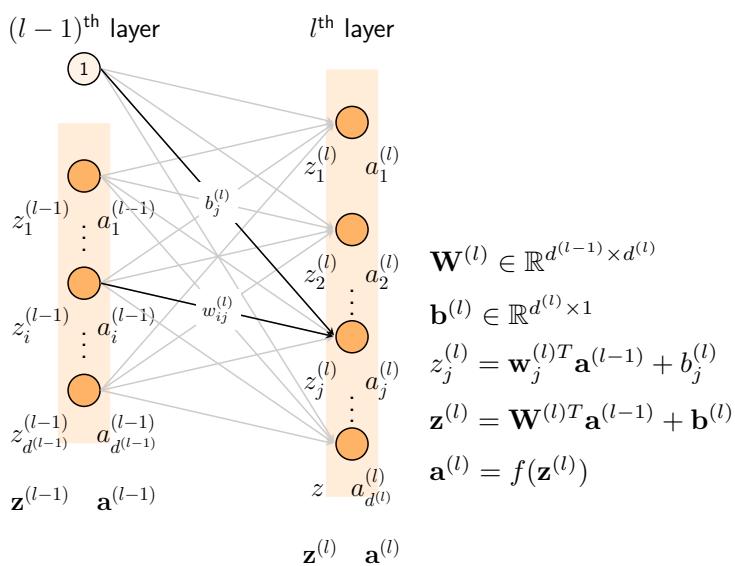
Số lượng layer trong một multilayer neural network, được ký hiệu là  $L$ , được tính bằng số hidden layer cộng với một. Khi đếm số layer của một multilayer neural network, ta không tính input layer. Trong Hình 5.3,  $L = 3$ .

### 5.2.2 Units

Quan sát Hình 5.4, mỗi *node* hình tròn trong một layer được gọi là một *unit*. Unit ở input layer, các hidden layer, và output layer được lần lượt gọi là input unit, hidden unit, và output



**Hình 5.3:** MLP với hai hidden layers (các biases đã bị ẩn).

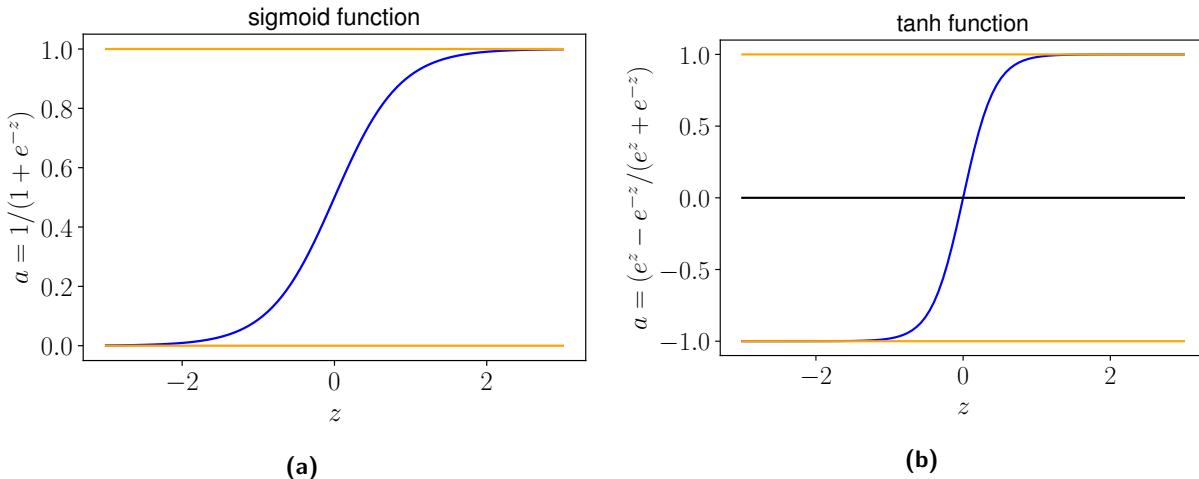


**Hình 5.4:** Các ký hiệu sử dụng trong multilayer neural network.

unit. Đầu vào của hidden layer thứ  $l$  được ký hiệu bởi  $\mathbf{z}^{(l)}$ , đầu ra của mỗi unit thường được ký hiệu là  $\mathbf{a}^{(l)}$  (thể hiện *activation*, tức giá trị của mỗi unit sau khi ta áp dụng activation function lên đầu vào  $\mathbf{z}^{(l)}$ ). Đầu ra của unit thứ  $i$  trong layer thứ  $l$  được ký hiệu là  $a_i^{(l)}$ . Giả sử thêm rằng số unit trong layer thứ  $l$  (không tính bias) là  $d^{(l)}$ . Vector biểu diễn output của layer thứ  $l$  được ký hiệu là  $\mathbf{a}^{(l)} \in \mathbb{R}^{d^{(l)}}$ .

### 5.2.3 Weights và Biases

Có  $L$  ma trận trọng số cho một multilayer neural network có  $L$  layer. Các ma trận này được ký hiệu là  $\mathbf{W}^{(l)} \in \mathbb{R}^{d^{(l-1)} \times d^{(l)}}$ ,  $l = 1, 2, \dots, L$  trong đó  $\mathbf{W}^{(l)}$  thể hiện các *kết nối* từ layer thứ  $l-1$  tới layer thứ  $l$  (nếu ta coi input layer là layer thứ 0). Cụ thể hơn, phần tử  $w_{ij}^{(l)}$  thể hiện kết nối từ node thứ  $i$  của layer thứ  $(l-1)$  tới node từ  $j$  của layer thứ  $(l)$ . Các bias của layer thứ  $(l)$  được ký hiệu là  $\mathbf{b}^{(l)} \in \mathbb{R}^{d^{(l)}}$ . Các trọng số này được ký hiệu như trên Hình 5.4. Khi tối ưu một multilayer neural network cho một công việc nào đó, chúng ta cần đi tìm các weights và biases này. Tập hợp các weight và bias lần lượt được ký hiệu là  $\mathbf{W}$  và  $\mathbf{b}$ .



**Hình 5.5:** Ví dụ về đồ thị của hàm (a)sigmoid và (b)tanh.

### 5.3 Activation function – Hàm kích hoạt

Mỗi output của một layer (trừ input layer) được tính dựa vào công thức

$$\mathbf{a}^{(l)} = f^{(l)}(\mathbf{W}^{(l)T} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}) \quad (5.7)$$

Trong đó  $f^{(l)}(.)$  là một hàm kích hoạt phi tuyến. Nếu hàm kích hoạt tại một layer là một hàm tuyến tính, layer này và layer tiếp theo có thể rút gọn thành một layer vì *hợp của các hàm tuyến tính là một hàm tuyến tính*.

Hàm kích hoạt thường là một hàm số áp dụng lên *từng phần tử* của ma trận hoặc vector đầu vào, nói cách khác, hàm kích hoạt thường là *element-wise*<sup>3</sup>.

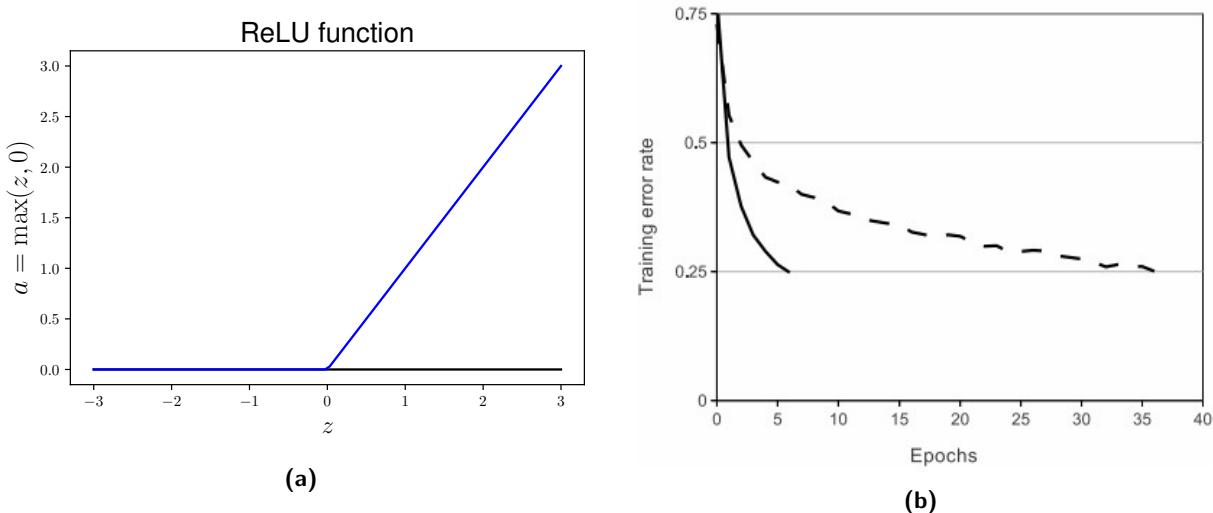
### 5.3.1 Hàm $sgn$ không được sử dụng trong MLP

Hàm  $sgn$  chỉ được sử dụng trong perceptron. Trong thực tế, hàm  $sgn$  không được sử dụng vì và đạo hàm tại hầu hết các điểm bằng 0 (trừ tại điểm 0 không có đạo hàm). Việc đạo hàm bằng 0 này khiến cho các thuật toán dựa trên gradient không hoạt động.

### 5.3.2 Sigmoid và tanh

Hàm *sigmoid* có dạng  $\text{sigmoid}(z) = 1/(1 + \exp(-z))$  với đồ thị như trong Hình 5.5a. Nếu đầu vào lớn, hàm số sẽ cho đầu ra gần với 1. Với đầu vào nhỏ (rất âm), hàm số sẽ cho đầu ra gần với 0. Trước đây, hàm kích hoạt này được sử dụng nhiều vì có đạo hàm rất *đẹp*. Những năm gần đây, hàm số này ít khi được sử dụng. Một hàm tương tự thường được sử dụng và

<sup>3</sup> Hàm softmax không phải là một hàm *element-wise* vì nó sử dụng mọi thành phần của vector đầu vào.



**Hình 5.6:** Hàm ReLU và tốc độ hội tụ khi so sánh với hàm tanh.

mang lại hiệu quả tốt hơn là hàm tanh với  $\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$ . Hàm số này có tính chất đầu ra chạy từ -1 đến 1, khiến cho nó có tính chất zero-centered, thay vì chỉ dương như hàm sigmoid. Gần đây, hàm sigmoid chỉ được sử dụng ở output layer khi yêu cầu của đầu ra là các giá trị nhị phân. Một nhược điểm dễ nhận thấy là khi đầu vào có trị tuyệt đối lớn (rất âm hoặc rất dương), đạo hàm của cả sigmoid và tanh sẽ rất gần với 0. Điều này đồng nghĩa với việc các hệ số tương ứng với unit đang xét sẽ gần như không được cập nhật khi sử dụng công thức cập nhật gradient descent.Thêm nữa, khi khởi tạo các hệ số cho multilayer neural network với hàm kích hoạt sigmoid, chúng ta phải tránh trường hợp đầu vào một hidden layer nào đó quá lớn, vì khi đó đầu ra của hidden layer đó sẽ rất gần với 0 hoặc 1, dẫn đến đạo hàm bằng 0 và gradient descent hoạt động không hiệu quả.

### 5.3.3 ReLU

ReLU (Rectified Linear Unit) được sử dụng rộng rãi gần đây vì tính đơn giản của nó. Đồ thị của hàm ReLU được minh họa trên Hình 5.6a. Hàm ReLU có công thức toán học  $f(z) = \max(0, z)$  - rất đơn giản, rất lợi về mặt tính toán. Đạo hàm của nó bằng 0 tại các điểm âm, bằng 1 tại các điểm dương. ReLU được chứng minh giúp cho việc huấn luyện các multilayer neural network và deep network (rất nhiều hidden layer) nhanh hơn rất nhiều so với hàm tanh [KSH12]. Hình 5.6b so sánh sự hội tụ của hàm mất mát khi sử dụng hai hàm kích hoạt ReLU và tanh. Sự tăng tốc này được cho là vì ReLU được tính toán gần như tức thời và gradient của nó cũng được tính cực nhanh.

Mặc dù cũng có nhược điểm đạo hàm bằng 0 với các giá trị đầu vào âm, ReLU được chứng minh bằng thực nghiệm rằng có thể khắc phục việc này bằng việc tăng số hidden unit<sup>4</sup>.

<sup>4</sup> Neural Networks and Deep Learning – Activation function (<https://goo.gl/QGjKmU>).

ReLU trở thành hàm kích hoạt đầu tiên chúng ta nên thử khi thiết kế một multilayer neural network. Hầu hết các network đều có hàm kích hoạt là ReLU trong các hidden unit, trừ hàm kích hoạt ở output layer phụ thuộc vào đầu ra thực sự của mỗi bài toán (có thể nhận giá trị âm, hoặc nhị phân, v.v.).

Ngoài ra, các biến thể của ReLU như leaky rectified linear unit (Leaky ReLU), parametric rectified linear unit (PReLU) và randomized leaky rectified linear units (RReLU) [XWCL15] cũng được sử dụng và được báo cáo có kết quả tốt. Trong thực tế, trước khi thiết kế, ta thường không biết chính xác hàm kích hoạt nào sẽ cho kết quả tốt nhất. Tuy nhiên, ta nên bắt đầu bằng ReLU, nếu kết quả chưa khả quan thì có thể thay thế bằng các biến thể của nó và so sánh kết quả.

## 5.4 Backpropagation

Phương pháp phổ biến nhất để tối ưu multilayer neural network chính là gradient descent (GD). Để áp dụng GD, chúng ta cần tính được đạo hàm của hàm mất mát theo từng ma trận trọng số  $\mathbf{W}^{(l)}$  và vector bias  $\mathbf{b}^{(l)}$ .

Giả sử  $J(\mathbf{W}, \mathbf{b}, \mathbf{X}, \mathbf{Y})$  là một hàm mất mát của bài toán, trong đó  $\mathbf{W}, \mathbf{b}$  là tập hợp tất cả các ma trận trọng số giữa các layer và vector bias của mỗi layer.  $\mathbf{X}, \mathbf{Y}$  là cặp dữ liệu huấn luyện với mỗi cột tương ứng với một điểm dữ liệu. Để có thể áp dụng các phương pháp gradient descent, chúng ta cần tính được

$$\frac{\partial J}{\partial \mathbf{W}^{(l)}}; \frac{\partial J}{\partial \mathbf{b}^{(l)}}, \quad l = 1, 2, \dots, L \quad (5.8)$$

Nhắc lại quá trình feedforward

$$\mathbf{a}^{(0)} = \mathbf{x} \quad (5.9)$$

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)T} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}, \quad l = 1, 2, \dots, L \quad (5.10)$$

$$\mathbf{a}^{(l)} = f^{(l)}(\mathbf{z}^{(l)}), \quad l = 1, 2, \dots, L \quad (5.11)$$

$$\hat{\mathbf{y}} = \mathbf{a}^{(L)} \quad (5.12)$$

Một ví dụ của hàm mất mát là hàm mean square error (MSE) tức *trung bình của bình phương lỗi* trong bài toán regression

$$J(\mathbf{W}, \mathbf{b}, \mathbf{X}, \mathbf{Y}) = \frac{1}{N} \sum_{n=1}^N \|\mathbf{y}_n - \hat{\mathbf{y}}_n\|_2^2 = \frac{1}{N} \sum_{n=1}^N \|\mathbf{y}_n - \mathbf{a}_n^{(L)}\|_2^2 \quad (5.13)$$

Với  $N$  là số cặp dữ liệu ( $\mathbf{x}, \mathbf{y}$ ) trong tập huấn luyện.

Theo các công thức này, việc tính toán trực tiếp các giá trị đạo hàm là cực kỳ phức tạp vì hàm mất mát không phụ thuộc trực tiếp vào các ma trận hệ số và vector bias. Phương pháp phổ biến nhất được dùng có tên là backpropagation giúp tính đạo hàm ngược từ layer cuối

cùng đến layer đầu tiên. Layer cuối cùng được tính toán trước vì nó *gần gần* hơn với *đầu ra dự đoán* và hàm mất mát. Việc tính toán đạo hàm của các ma trận hệ số trong các layer trước được thực hiện dựa trên một quy tắc chuỗi quen thuộc cho *đạo hàm của hàm hợp*.

Stochastic gradient descent có thể được sử dụng để tính gradient cho các ma trận trọng số và biases dựa trên một cặp điểm training  $\mathbf{x}, \mathbf{y}$ . Để cho đơn giản, ta coi  $J$  là hàm mất mát nếu chỉ xét cặp điểm này, ở đây  $J$  là hàm mất mát bất kỳ, không chỉ hàm MSE như ở trên. Đạo hàm của hàm mất mát theo *chỉ một thành phần* của ma trận trọng số của output layer

$$\frac{\partial J}{\partial w_{ij}^{(L)}} = \frac{\partial J}{\partial z_j^{(L)}} \cdot \frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} = e_j^{(L)} a_i^{(L-1)} \quad (5.14)$$

Trong đó  $e_j^{(L)} = \frac{\partial J}{\partial z_j^{(L)}}$  thường là một đại lượng *không quá khó để tính toán* và  $\frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} = a_i^{(L-1)}$  vì  $z_j^{(L)} = \mathbf{w}_j^{(L)T} \mathbf{a}^{(L-1)} + b_j^{(L)}$ . Tương tự như thế, đạo hàm của hàm mất mát theo bias của layer cuối cùng là

$$\frac{\partial J}{\partial b_j^{(L)}} = \frac{\partial J}{\partial z_j^{(L)}} \cdot \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} = e_j^{(L)} \quad (5.15)$$

Với đạo hàm theo hệ số ở các lớp  $l$  thấp hơn, chúng ta hãy xem Hình 5.7. Ở đây, tại mỗi unit, đầu vào  $z$  và đầu ra  $a$  được viết riêng để chúng ta tiện theo dõi.

Dựa vào Hình 5.7, bằng quy nạp ngược từ cuối, ta có thể tính được

$$\frac{\partial J}{\partial w_{ij}^{(l)}} = \frac{\partial J}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} = e_j^{(l)} a_i^{(l-1)} \quad (5.16)$$

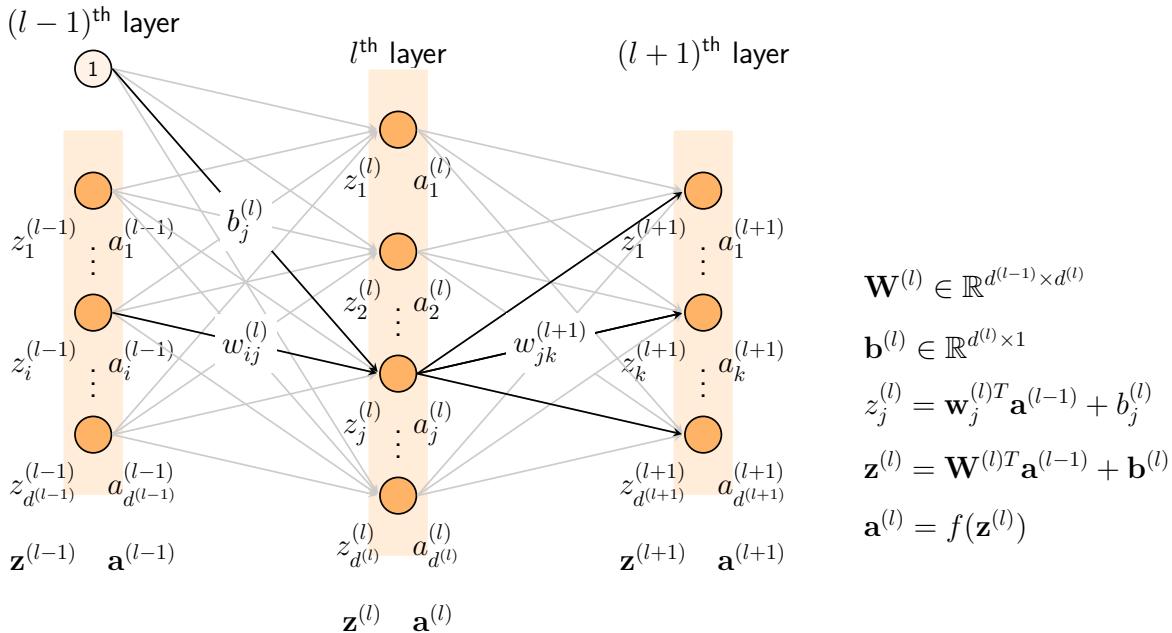
với

$$e_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}} = \frac{\partial J}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \quad (5.17)$$

$$= \left( \sum_{k=1}^{d^{(l+1)}} \frac{\partial J}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial a_j^{(l)}} \right) f^{(l)'}(z_j^{(l)}) = \left( \sum_{k=1}^{d^{(l+1)}} e_k^{(l+1)} w_{jk}^{(l+1)} \right) f^{(l)'}(z_j^{(l)}) \quad (5.18)$$

trong đó  $\mathbf{e}^{(l+1)} = [e_1^{(l+1)}, e_2^{(l+1)}, \dots, e_{d^{(l+1)}}^{(l+1)}]^T \in \mathbb{R}^{d^{(l+1)} \times 1}$  và  $\mathbf{w}_{j:}^{(l+1)}$  được hiểu là **hàng** thứ  $j$  của ma trận  $\mathbf{W}^{(l+1)}$  (chú ý dấu hai chấm, khi không có dấu này, chúng ta mặc định dùng nó để ký hiệu cho vector *cột*).

Dấu  $\sum$  tính tổng ở dòng thứ hai trong phép tính trên xuất hiện vì  $a_j^{(l)}$  đóng góp vào việc tính tất cả các  $z_k^{(l+1)}, k = 1, 2, \dots, d^{(l+1)}$ . Biểu thức đạo hàm ngoài dấu ngoặc lớn là vì  $a_j^{(l)} = f^{(l)}(z_j^{(l)})$ . Ở đây, ta có thể thấy rằng việc activation function có đạo hàm đơn giản sẽ có ích rất nhiều trong việc tính toán. Với cách làm tương tự, bạn đọc có thể suy ra  $\frac{\partial J}{\partial b_j^{(l)}} = e_j^{(l)}$ .



**Hình 5.7:** Mô phỏng cách tính backpropagation. Layer cuối có thể là output layer.

Nhận thấy rằng trong các công thức trên đây, việc tính các  $e_j^{(l)}$  đóng một vài trò quan trọng. Hơn nữa, để tính được giá trị này, ta cần tính được các  $e_j^{(l+1)}$ . Nói cách khác, ta cần tính *ngược* các giá trị này từ cuối. Cái tên *backpropagation* cũng xuất phát từ việc này.

Việc tính toán các đạo hàm khi sử dụng SGD có thể tóm tắt như sau

### Thuật toán 5.1: Backpropagation tới $w_{ij}^{(l)}, b_i^{(l)}$

1. *Bước feedforward:* Với 1 giá trị đầu vào  $\mathbf{x}$ , tính giá trị đầu ra của network, trong quá trình tính toán, lưu lại các giá trị activation  $\mathbf{a}^{(l)}$  tại mỗi layer.
2. *Với mỗi unit j ở output layer, tính*

$$e_j^{(L)} = \frac{\partial J}{\partial z_j^{(L)}}; \quad \frac{\partial J}{\partial w_{ij}^{(L)}} = a_i^{(L-1)} e_j^{(L)}; \quad \frac{\partial J}{\partial b_j^{(L)}} = e_j^{(L)} \quad (5.19)$$

3. *Với  $l = L-1, L-2, \dots, 1$ , tính:*

$$e_j^{(l)} = \left( \mathbf{w}_{j:}^{(l+1)} \mathbf{e}^{(l+1)} \right) f'(z_j^{(l)}) \quad (5.20)$$

4. *Cập nhật đạo hàm cho từng hẽ số*

$$\frac{\partial J}{\partial w_{ij}^{(l)}} = a_i^{(l-1)} e_j^{(l)}; \quad \frac{\partial J}{\partial b_j^{(l)}} = e_j^{(l)} \quad (5.21)$$

Phiên bản *vectorization* của thuật toán trên có thể được thực hiện như sau.

### Thuật toán 5.2: Backpropagation tối $\mathbf{W}^{(l)}$ và vector bias $\mathbf{b}^{(l)}$

1. *Bước feedforward:* Với một giá trị đầu vào  $\mathbf{x}$ , tính giá trị đầu ra của network, trong quá trình tính toán, lưu lại các activation  $\mathbf{a}^{(l)}$  tại mỗi layer.
2. *Với output layer, tính*

$$\mathbf{e}^{(L)} = \frac{\partial J}{\partial \mathbf{z}^{(L)}} \in \mathbb{R}^{d^{(L)}}; \quad \frac{\partial J}{\partial \mathbf{W}^{(L)}} = \mathbf{a}^{(L-1)} \mathbf{e}^{(L)T} \in \mathbb{R}^{d^{(L-1)} \times d^{(L)}}; \quad \frac{\partial J}{\partial \mathbf{b}^{(L)}} = \mathbf{e}^{(L)}$$

3. *Với  $l = L - 1, L - 2, \dots, 1$ , tính:*

$$\mathbf{e}^{(l)} = (\mathbf{W}^{(l+1)} \mathbf{e}^{(l+1)}) \odot f'(\mathbf{z}^{(l)}) \in \mathbb{R}^{d^{(l)}} \quad (5.22)$$

trong đó  $\odot$  là element-wise product hay Hadamard product tức lấy từng thành phần của hai vector nhân với nhau để được vector kết quả.

4. *Cập nhật đạo hàm cho ma trận trọng số và vector biases:*

$$\frac{\partial J}{\partial \mathbf{W}^{(l)}} = \mathbf{a}^{(l-1)} \mathbf{e}^{(l)T} \in \mathbb{R}^{d^{(l-1)} \times d^{(l)}}; \quad \frac{\partial J}{\partial \mathbf{b}^{(l)}} = \mathbf{e}^{(l)} \quad (5.23)$$

Khi làm việc với các phép tính đạo hàm phức tạp, ta luôn cần nhớ hai điều sau.

1. Đạo hàm của một hàm có đầu ra là một số vô hướng theo một vector hoặc ma trận là một đại lượng có cùng chiều với vector hoặc ma trận đó.
2. Để các phép nhân các ma trận, vector thực hiện được, ta cần đảm bảo chiều của chúng phù hợp.

Trong công thức  $\frac{\partial J}{\partial \mathbf{W}^{(L)}} = \mathbf{a}^{(L-1)} \mathbf{e}^{(L)T}$ , về trái là một ma trận thuộc  $\mathbb{R}^{d^{(L-1)} \times d^{(L)}}$ , vậy về phải cũng phải là một đại lượng có chiều tương tự. Từ đó bạn đọc có thể thấy tại sao về phải phải là  $\mathbf{a}^{(L-1)} \mathbf{e}^{(L)T}$  mà không thể là  $\mathbf{a}^{(L-1)} \mathbf{e}^{(L)}$  hay  $\mathbf{e}^{(L)} \mathbf{a}^{(L-1)}$ .

#### 5.4.1 Backpropagation cho Batch (mini-batch) Gradient Descent

Nếu ta muốn thực hiện batch hoặc mini-batch gradient descent thì thế nào? Trong thực tế, mini-batch gradient descent được sử dụng nhiều nhất với các bài toán mà tập huấn luyện lớn. Nếu lượng dữ liệu là nhỏ, batch gradient descent trực tiếp được sử dụng.

Khi đó, cặp (input, output) sẽ ở dạng ma trận  $(\mathbf{X}, \mathbf{Y})$ . Giả sử rằng mỗi lần tính toán, ta lấy  $N$  dữ liệu để tính toán. Khi đó,  $\mathbf{X} \in \mathbb{R}^{d^{(0)} \times N}$ ,  $\mathbf{Y} \in \mathbb{R}^{d^{(L)} \times N}$ . Với  $d^{(0)} = d$  là chiều của dữ liệu đầu vào (không tính bias).

Khi đó các activation sau mỗi layer sẽ có dạng  $\mathbf{A}^{(l)} \in \mathbb{R}^{d^{(l)} \times N}$ . Tương tự thế,  $\mathbf{E}^{(l)} \in \mathbb{R}^{d^{(l)} \times N}$ . Và ta cũng có thể suy ra công thức cập nhật như sau.

### Thuật toán 5.3: Backpropagation tối $\mathbf{W}^{(l)}$ và bias $\mathbf{b}^{(l)}$ (mini-batch)

1. *Bước feedforward:* Với toàn bộ dữ liệu (batch) hoặc một nhóm dữ liệu (mini-batch) đầu vào  $\mathbf{X}$ , tính giá trị đầu ra của network, trong quá trình tính toán, lưu lại các activation  $\mathbf{A}^{(l)}$  tại mỗi layer. Mỗi cột của  $\mathbf{A}^{(l)}$  tương ứng với một cột của  $\mathbf{X}$ , tức một điểm dữ liệu đầu vào.
2. *Với output layer, tính*

$$\mathbf{E}^{(L)} = \frac{\partial J}{\partial \mathbf{Z}^{(L)}}; \quad \frac{\partial J}{\partial \mathbf{W}^{(L)}} = \mathbf{A}^{(L-1)} \mathbf{E}^{(L)T}; \quad \frac{\partial J}{\partial \mathbf{b}^{(L)}} = \sum_{n=1}^N \mathbf{e}_n^{(L)} \quad (5.24)$$

3. *Với  $l = L - 1, L - 2, \dots, 1$ , tính:*

$$\mathbf{E}^{(l)} = (\mathbf{W}^{(l+1)} \mathbf{E}^{(l+1)}) \odot f'(\mathbf{Z}^{(l)}) \quad (5.25)$$

trong đó  $\odot$  là element-wise product hay Hadamard product tức lấy từng thành phần của hai ma trận nhân với nhau để được ma trận kết quả.

4. *Cập nhật đạo hàm cho ma trận trọng số và vector biases:*

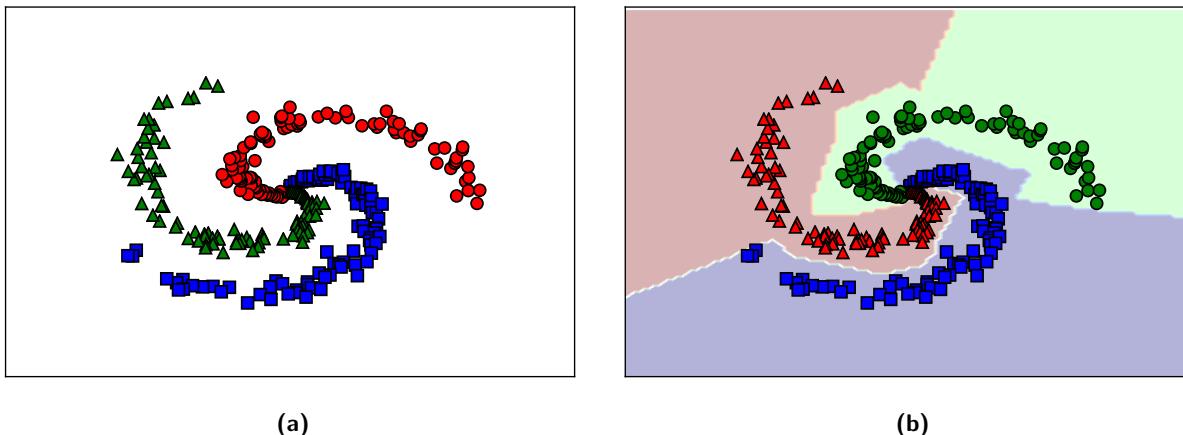
$$\frac{\partial J}{\partial \mathbf{W}^{(l)}} = \mathbf{A}^{(l-1)} \mathbf{E}^{(l)T}; \quad \frac{\partial J}{\partial \mathbf{b}^{(l)}} = \sum_{n=1}^N \mathbf{e}_n^{(l)} \quad (5.26)$$

## 5.5 Ví dụ trên Python

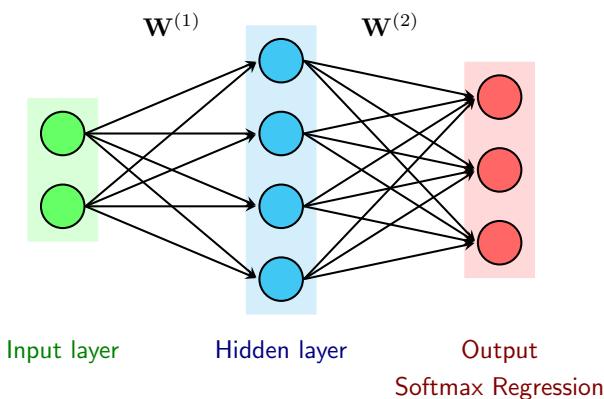
Trong mục này, chúng ta sẽ tạo dữ liệu giả trong không gian hai chiều sao cho đường ranh giới giữa các lớp này không có dạng tuyến tính. Điều này khiến cho softmax regression không làm việc được. Tuy nhiên, bằng cách thêm một hidden layer, chúng ta sẽ thấy rằng neural network này làm việc rất hiệu quả.

### 5.5.1 Tạo dữ liệu giả

Các điểm dữ liệu giả của ba lớp được tạo và minh họa bởi các màu khác nhau trên Hình 5.8a. Ta thấy rõ ràng rằng đường ranh giới giữa các lớp dữ liệu không thể là các đường thẳng. Hình 5.8b là một ví dụ về các đường ranh giới được coi là tốt với hầu hết các điểm dữ liệu nằm đúng vào khu vực có màu nền tương ứng. Các đường biên này được tạo sử dụng multilayer neural network với một hidden layer sử dụng ReLU làm hàm kích hoạt và output layer là một softmax regression như trên Hình 5.9. Chúng ta cùng đi sâu vào xây dựng bộ phân lớp dựa trên dữ liệu huấn luyện này.



**Hình 5.8:** Dữ liệu giả trong không gian hai chiều và ví dụ về các ranh giới tốt.



**Hình 5.9:** Multilayer neural network với input layer có hai unit (bias đã được ẩn), một hidden layer với hàm kích hoạt ReLU (có thể có số lượng hidden unit tùy ý), và output layer là một softmax regression với ba phần tử đại diện cho ba lớp dữ liệu.

Nhắc lại hàm ReLU  $f(z) = \max(z, 0)$ , với đạo hàm

$$f'(z) = \begin{cases} 0 & \text{néu } z \leq 0 \\ 1 & \text{o.w} \end{cases} \quad (5.27)$$

Vì lượng dữ liệu huấn luyện là nhỏ với 100 điểm cho mỗi lớp, ta có thể dùng batch gradient descent để cập nhật các ma trận hệ số và vector bias. Trước hết, ta cần tính đạo hàm của hàm mất mát theo các ma trận và vector này bằng cách áp dụng backpropagation.

### 5.5.2 Tính toán Feedforward

Giả sử các cặp dữ liệu huấn luyện là  $(\mathbf{x}_i, \mathbf{y}_i)$  với  $\mathbf{y}_i$  là một vector ở dạng one-hot. Các điểm dữ liệu này xếp cạnh nhau tạo thành các ma trận đầu vào  $\mathbf{X}$  và ma trận đầu ra  $\mathbf{Y}$ . Bước feedforward của neural network này được thực hiện như sau.

$$\mathbf{Z}^{(1)} = \mathbf{W}^{(1)T} \mathbf{X} + \mathbf{B}^{(1)} \quad (5.28)$$

$$\mathbf{A}^{(1)} = \max(\mathbf{Z}^{(1)}, \mathbf{0}) \quad (5.29)$$

$$\mathbf{Z}^{(2)} = \mathbf{W}^{(2)T} \mathbf{A}^{(1)} + \mathbf{B}^{(2)} \quad (5.30)$$

$$\hat{\mathbf{Y}} = \mathbf{A}^{(2)} = \text{softmax}(\mathbf{Z}^{(2)}) \quad (5.31)$$

Trong đó  $\mathbf{B}^{(1)}, \mathbf{B}^{(2)}$  là các ma trận bias với tất cả các cột bằng nhau và lần lượt bằng  $\mathbf{b}^{(1)}$  và  $\mathbf{b}^{(2)}$ <sup>5</sup>. Hàm mất mát được tính dựa trên hàm cross-entropy

$$J \triangleq J(\mathbf{W}, \mathbf{b}; \mathbf{X}, \mathbf{Y}) = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^C y_{ji} \log(\hat{y}_{ji}) \quad (5.32)$$

### 5.5.3 Tính toán Backpropagation

Áp dụng Thuật toán 5.3, ta có

$$\mathbf{E}^{(2)} = \frac{\partial J}{\partial \mathbf{Z}^{(2)}} = \frac{1}{N} (\mathbf{A}^{(2)} - \mathbf{Y}) \quad (5.33)$$

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \mathbf{A}^{(1)} \mathbf{E}^{(2)T}; \quad \frac{\partial J}{\partial \mathbf{b}^{(2)}} = \sum_{n=1}^N \mathbf{e}_n^{(2)} \quad (5.34)$$

$$\mathbf{E}^{(1)} = (\mathbf{W}^{(2)} \mathbf{E}^{(2)}) \odot f'(\mathbf{Z}^{(1)}) \quad (5.35)$$

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \mathbf{A}^{(0)} \mathbf{E}^{(1)T} = \mathbf{X} \mathbf{E}^{(1)T}; \quad \frac{\partial J}{\partial \mathbf{b}^{(1)}} = \sum_{n=1}^N \mathbf{e}_n^{(1)} \quad (5.36)$$

Các công thức toán học phức tạp này sẽ được lập trình một cách đơn giản hơn trên numpy.

### 5.5.4 Triển khai thuật toán trên numpy

Trước hết, ta viết lại hàm softmax và cross-entropy.

```
def softmax_stable(Z):
    """
    Compute softmax values for each sets of scores in Z.
    each ROW of Z is a set of scores.
    """
    e_Z = np.exp(Z - np.max(Z, axis = 1, keepdims = True))
    A = e_Z / e_Z.sum(axis = 1, keepdims = True)
    return A

def crossentropy_loss(Yhat, y):
    """
    Yhat: a numpy array of shape (Npoints, nClasses) -- predicted output
    y: a numpy array of shape (Npoints) -- ground truth.
    NOTE: We don't need to use the one-hot vector here since most of elements
    are zeros. When programming in numpy, in each row of Yhat, we need to access
    to the corresponding indexes only.
    """
    id0 = range(Yhat.shape[0])
    return -np.mean(np.log(Yhat[id0, y]))
```

<sup>5</sup> Ta cần xếp các vector bias giống nhau để tạo thành các ma trận bias vì trong toán học, không có định nghĩa tổng của một ma trận và một vector. Khi lập trình, việc này là khả thi.

Tiếp theo là các hàm phục vụ việc khởi tạo, dự đoán và huấn luyện network.

```

def mlp_init(d0, d1, d2):
    """
    Initialize W1, b1, W2, b2
    d0: dimension of input data
    d1: number of hidden unit
    d2: number of output unit = number of classes
    """
    W1 = 0.01*np.random.randn(d0, d1)
    b1 = np.zeros(d1)
    W2 = 0.01*np.random.randn(d1, d2)
    b2 = np.zeros(d2)
    return (W1, b1, W2, b2)

def mlp_predict(X, W1, b1, W2, b2):
    """
    Suppose that the network has been trained, predict class of new points.
    X: data matrix, each ROW is one data point.
    W1, b1, W2, b2: learned weight matrices and biases
    """
    Z1 = X.dot(W1) + b1      # shape (N, d1)
    A1 = np.maximum(Z1, 0)    # shape (N, d1)
    Z2 = A1.dot(W2) + b2     # shape (N, d2)
    return np.argmax(Z2, axis=1)

def mlp_fit(X, y, W1, b1, W2, b2, eta):
    loss_hist = []
    for i in xrange(10000): # number of epoches
        # feedforward
        Z1 = X.dot(W1) + b1      # shape (N, d1)
        A1 = np.maximum(Z1, 0)    # shape (N, d1)
        Z2 = A1.dot(W2) + b2     # shape (N, d2)
        Yhat = softmax_stable(Z2) # shape (N, d2)

        if i %1000 == 0: # print loss after each 1000 iterations
            loss = crossentropy_loss(Yhat, y)
            print("iter %d, loss: %f" %(i, loss))
            loss_hist.append(loss)

        # back propagation
        id0 = range(Yhat.shape[0])
        Yhat[id0, y] -=1
        E2 = Yhat/N             # shape (N, d2)
        dW2 = np.dot(A1.T, E2)   # shape (d1, d2)
        db2 = np.sum(E2, axis = 0) # shape (d2,)
        E1 = np.dot(E2, W2.T)    # shape (N, d1)
        E1[Z1 <= 0] = 0         # gradient of ReLU, shape (N, d1)
        dW1 = np.dot(X.T, E1)    # shape (d0, d1)
        db1 = np.sum(E1, axis = 0) # shape (d1,)

        # Gradient Descent update
        W1 += -eta*dW1
        b1 += -eta*db1
        W2 += -eta*dW2
        b2 += -eta*db2
    return (W1, b1, W2, b2, loss_hist)

```

Sau khi đã hoàn thành các hàm chính của multilayer neural network này, chúng ta đưa dữ liệu vào, xác định số hidden unit, và huấn luyện network.

```
# suppose X, y are training input and output, respectively
d0 = 2          # data dimension
d1 = h = 100    # number of hidden units
d2 = C = 3      # number of classes
eta = 1          # learning rate
(W1, b1, W2, b2) = mlp_init(d0, d1, d2)
(W1, b1, W2, b2, loss_hist) = mlp_fit(X, y, W1, b1, W2, b2, eta)
y_pred = mlp_predict(X, W1, b1, W2, b2)
acc = 100*np.mean(y_pred == y)
print('training accuracy: %.2f %%' % acc)
```

Kết quả:

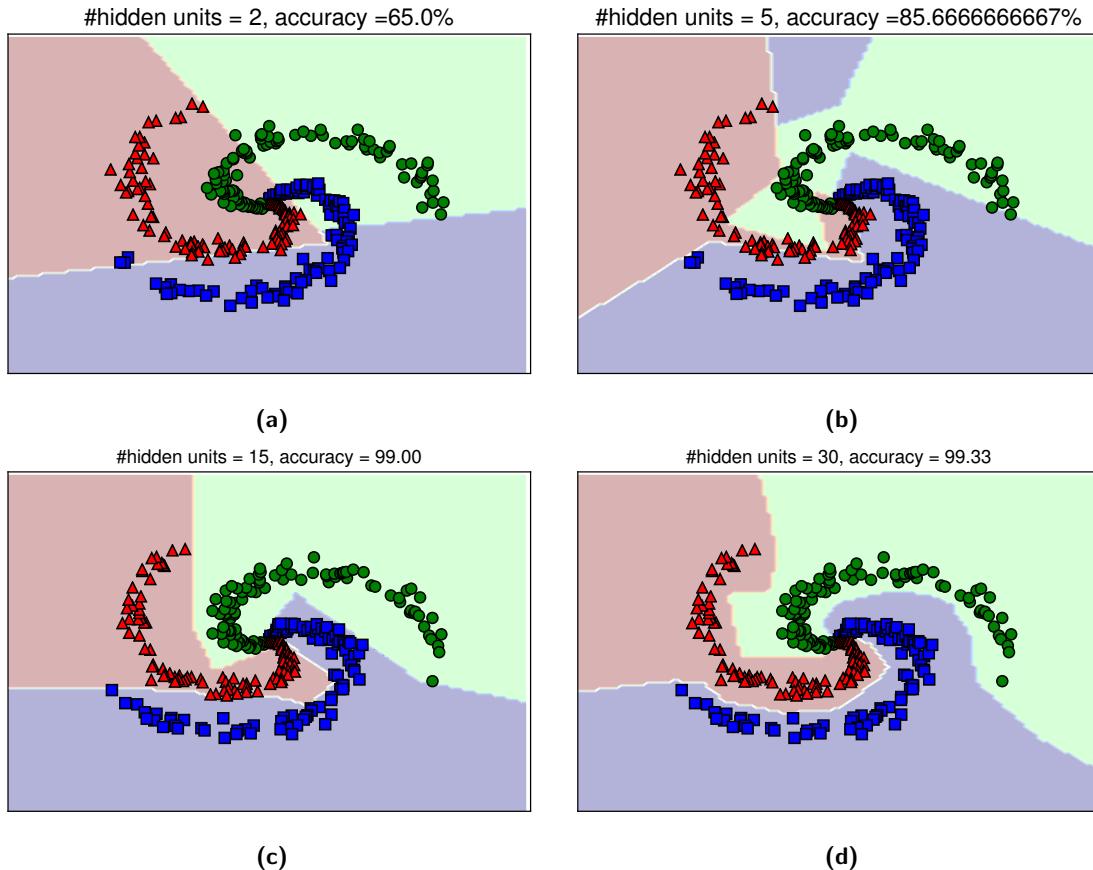
```
iter 0, loss: 1.098628
iter 1000, loss: 0.094922
iter 2000, loss: 0.030014
iter 3000, loss: 0.023423
iter 4000, loss: 0.021071
iter 5000, loss: 0.019116
iter 6000, loss: 0.018158
iter 7000, loss: 0.017541
iter 8000, loss: 0.016914
iter 9000, loss: 0.016671
training accuracy: 99.33 %
```

Ta có thể thấy rằng hàm mất mát giảm dần và hội tụ. Kết quả phân lớp trên tập huấn luyện rất tốt, chỉ một vài điểm bị phân lớp lỗi, nhiều khả năng chúng nằm ở khu vực trung tâm. Với chỉ một hidden layer, network đã thực hiện công việc gần như hoàn hảo.

Bằng cách thay đổi số lượng hidden unit (biến **d1**) và huấn luyện lại các network, minh họa ranh giới giữa các lớp dữ liệu, chúng ta thu được các kết quả như trên Hình 5.10. Khi chỉ có hai hidden unit, các đường ranh giới vẫn gần như thẳng, kết quả là có tới 35% số điểm dữ liệu trong tập huấn luyện bị phân lớp lỗi. Khi số lượng hidden unit là 5, độ chính xác được cải thiện thêm khoảng 20%, tuy nhiên, các đường ranh giới vẫn chưa thực sự tốt. Thậm chí lớp đỗ và lam còn bị chia cắt một cách không tự nhiên. Nếu tiếp tục tăng số lượng hidden unit, ta thấy rằng các đường ranh giới tương đối hoàn hảo.

Có thể chứng minh được rằng với một hàm số liên tục bất kỳ  $f(x)$  và một số  $\varepsilon > 0$ , luôn tồn tại một neural network với đầu ra có dạng  $g(x)$  với một hidden layer (với số hidden unit đủ lớn và hàm kích hoạt phi tuyến phù hợp) sao cho với mọi  $x$ ,  $|f(x) - g(x)| < \varepsilon$ . Nói cách khác, neural network có khả năng xấp xỉ bất kỳ hàm liên tục nào [Cyb89].

Trên thực tế, việc tìm ra số lượng hidden unit và hàm kích hoạt nói trên hầu như bất khả thi. Thay vào đó, thực nghiệm chứng minh rằng neural networks với nhiều hidden layer kết hợp với các hàm kích hoạt đơn giản, ví dụ ReLU, có khả năng xấp xỉ dữ liệu tốt hơn.



**Hình 5.10:** Kết quả với số lượng units trong hidden layer là khác nhau.

hơn. Tuy nhiên, khi số lượng hidden layer lớn lên, số lượng hệ số cần tối ưu cũng lớn lên và mô hình sẽ trở nên phức tạp. Sự phức tạp này ảnh hưởng tới hai khía cạnh. Thứ nhất, tốc độ tính toán sẽ bị chậm đi rất nhiều. Thứ hai, nếu mô hình quá phức tạp, nó có thể biểu diễn rất tốt dữ liệu huấn luyện, nhưng có thể không biểu diễn tốt dữ liệu kiểm thử. Đây chính là hiện tượng overfitting.

Vậy có các kỹ thuật nào giúp tránh overfitting cho multilayer neural network? Ngoài kỹ thuật toán cross-validation, chúng ta quan tâm hơn tới các phương pháp regularization. Các neural network với regularization được gọi là *regularized neural network*. Kỹ thuật phổ biến nhất được dùng để tránh overfitting là *weight decay*.

## 5.6 Tránh overfitting cho neural network bằng weight decay

Với weight decay, hàm mất mát sẽ được cộng thêm một đại lượng regularization có dạng

$$\lambda R(\mathbf{W}) = \lambda \sum_{l=1}^L \|\mathbf{W}^{(l)}\|_F^2$$

tức tổng bình phương Frobenius norm của tất cả các ma trận hệ số. Chú ý rằng khi làm việc với multilayer neural network, bias hiếm khi được *regularized*. Đây cũng là lý do vì sao ta nên tách rời ma trận hệ số và vector bias khi làm việc với multilayer neural network. Việc tối thiểu hàm mất mát mới (với số hạng regularization) sẽ khiến cho các thành phần của các vector hệ số  $\mathbf{W}^{(l)}$  không quá lớn, thậm chí nhiều thành phần sẽ gần với không. Điều này khiến cho việc có nhiều hidden unit vẫn an toàn vì nhiều trong số chúng gần với không.

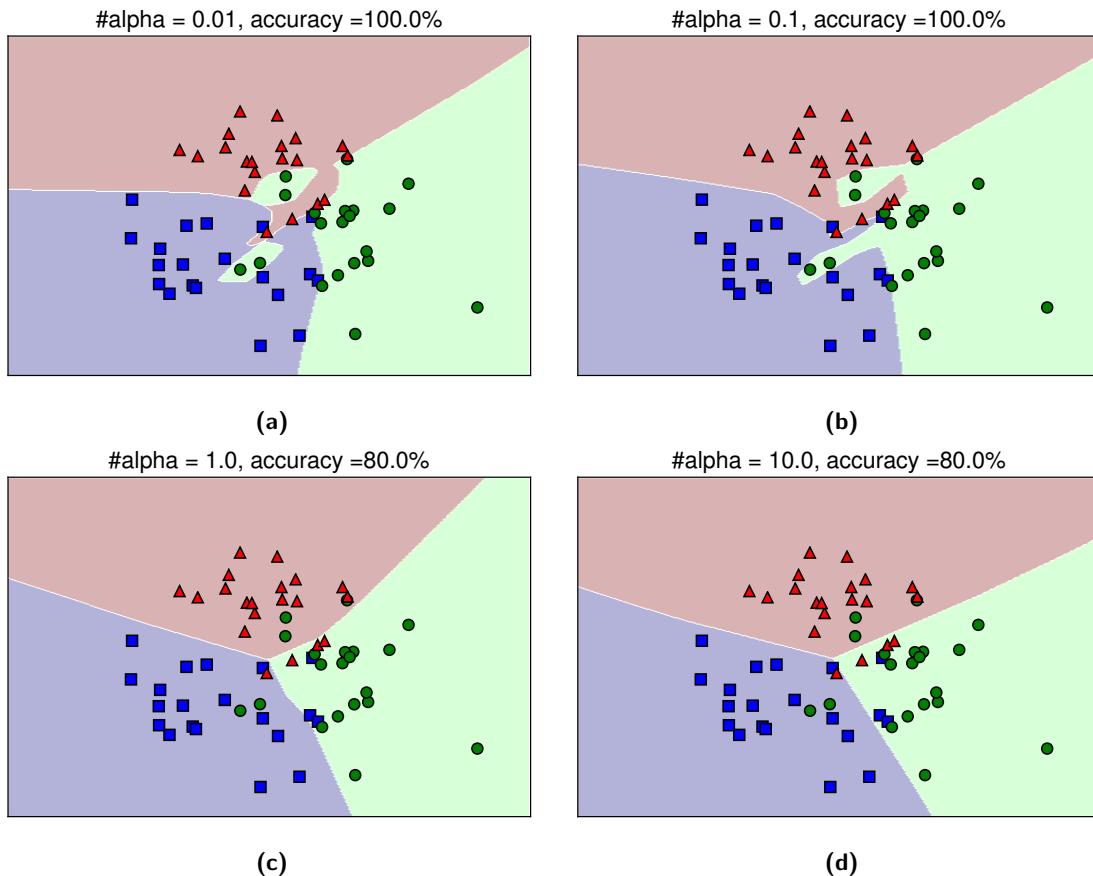
Tiếp theo, chúng ta sẽ làm một ví dụ nữa trong không gian hai chiều. Lần này, chúng ta sẽ sử dụng thư viện scikit-learn.

```
from __future__ import print_function
import numpy as np
from sklearn.neural_network import MLPClassifier
means = [[-1, -1], [1, -1], [0, 1]]
cov = [[1, 0], [0, 1]]
N = 20
X0 = np.random.multivariate_normal(means[0], cov, N)
X1 = np.random.multivariate_normal(means[1], cov, N)
X2 = np.random.multivariate_normal(means[2], cov, N)

X = np.concatenate((X0, X1, X2), axis = 0)
y = np.asarray([0]*N + [1]*N + [2]*N)

alpha = 1e-1 # regularization parameter
clf = MLPClassifier(solver='lbfgs', alpha=alpha, hidden_layer_sizes=(100))
clf.fit(X, y)
y_pred = clf.predict(X)
acc = 100*np.mean(y_pred == y)
print('training accuracy: %.2f %%' % acc)
```

Trong đoạn code trên, thuộc tính `alpha` chính là tham số regularization  $\lambda$ . `alpha` càng lớn sẽ khiến các thành phần trong các ma trận hệ số càng nhỏ. Thuộc tính `hidden_layer_sizes` chính là số lượng hidden unit trong mỗi hidden layer. Nếu có nhiều hidden layer, chẳng hạn hai với số lượng hidden unit lần lượt là 10 và 100, ta cần khai báo `hidden_layer_sizes=(10, 100)`. Hình 5.11 minh họa ranh giới giữa các lớp tìm được với các giá trị `alpha` khác nhau, tức mức độ regularization khác nhau. Khi `alpha` nhỏ cỡ 0.01, các ranh giới tìm được trông không được tự nhiên và vùng xác định lớp màu lục không được liên tục. Mặc dù độ chính xác trên tập huấn luyện này là 100%, ta có thể quan sát thấy rằng overfitting đã xảy ra. Với `alpha = 0.1`, kết quả cho thấy *lãnh thổ* của các lớp đã liên tục, nhưng overfitting vẫn xảy ra. Khi `alpha` cao hơn, độ chính xác đã giảm xuống nhưng các đường ranh giới tự nhiên hơn. Bạn đọc có thể thay đổi các giá trị `alpha` trong source code (<https://goo.gl/czxrSf>) và quan sát các hiện tượng xảy ra. Đặc biệt, khi `alpha = 100`, độ chính xác còn 33.33%. Tại sao lại như vậy? Hy vọng bạn đọc có thể tự trả lời được.



**Hình 5.11:** Kết quả với số lượng units trong hidden layer là khác nhau.

## 5.7 Đọc thêm

1. *Neural Networks: Setting up the Architecture*, Andrej Karpathy (<https://goo.gl/rfzCVK>).
  2. *Neural Networks, Case study*, Andrej Karpathy (<https://goo.gl/3ihCxL>).
  3. *Lecture Notes on Sparse Autoencoders*, Andrew Ng (<https://goo.gl/yTgtLe>).
  4. *Yes you should understand backprop* (<https://goo.gl/8B3h1b>).
  5. *Backpropagation, Intuitions*, Andrej Karpathy (<https://goo.gl/fjHzNV>).
  6. *How the backpropagation algorithm works*, Michael Nielsen (<https://goo.gl/mwz2kU>).

---

## References

- AMMIL12. Yaser S Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning from data*, volume 4. AML-Book New York, NY, USA:, 2012.
- Cyb89. George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 1989.
- DHS11. John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.
- KB14. Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- KSH12. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- Nes07. Yurii Nesterov. Gradient methods for minimizing composite objective function, 2007.
- Ros57. F Rosemblat. The perceptron: A perceiving and recognizing automation. *Cornell Aeronautical Laboratory Report*, 1957.
- Rud16. Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.
- TH12. Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.
- XWCL15. Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

---

# Index

- activation fuction
  - sigmoid fuction, 32
  - tanh fuction, 32
- activation function – hàm kích hoạt, 63
- cross entropy, 49
- element-wise, 63
- epoch, 18
- GD, *xem* gradient descent, 4
- gradient descent, 4
  - stopping criteria – điều kiện dừng, 19
  - batch gradient descent, 18
  - mini-batch gradient descent, 19
  - momentum, 13
  - Nesterov accelerated gradient, 16
  - stochastic gradient descent, 18
- hierarchical, 41
- level sets – đường đồng mức, 12
- linearly separable, 21
- local minimum – cực tiểu, 4
- one-vs-one, 41
- one-vs-rest, 42
- online learning, 18
- perceptron learning algorithm, 21
- PLA, 21
- regularized neural network, 74
- ReLU, 64
- SGD, *xem* gradient descent, 18
- sigmoid, 63
- softmax function, 46
- softmax regression, 45
- tanh, 63