dịch từ: <https://www.analyticsvidhya.com/blog/2018/10/introduction-neural-networks-deep-learning/>

An Introductory Guide to Deep Learning and Neural Networks

Deep Learning (DL)

# 1. Giới thiệu

Với sự bùng nổ công nghệ trong xã hội hiện nay, sẽ thật tuyệt nếu bạn được trang bị đầy đủ kiến thức về Deep Learning. Từ việc phân loại hình ảnh, dịch thuật hay thậm chí là trong lĩnh vực xe tự lái, tất cả các công việc này máy móc đang đều dần có khả năng thay thế con người, thậm chí là làm còn tốt hơn. DL đang dần thâm nhập sâu hơn vào từng ngành công nghiệp, và thậm chí công nghệ này đang cải tiến hàng ngày.

Nói một cách dễ hiểu, cả tấn người bỗng dưng quan tâm tới vấn đề này. Nhưng chúng ta phải bắt đầu như thế nào? Ta nên học cái gì? Cái cốt lõi tạo nên công nghệ đầy hấp dẫn này là gì?

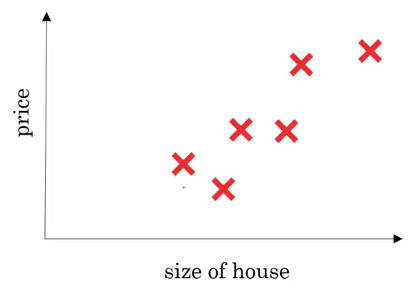
# 2. Mạng Neural và Deep Learning

## 2.1. Giới thiệu về Deep Learning

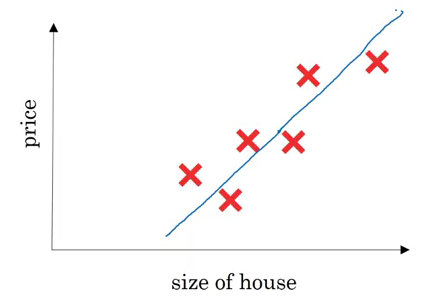
### Mạng Neural là gì?

Ta cùng bắt đầu với vấn đề cơ bản nhất: mạng neural là gì?

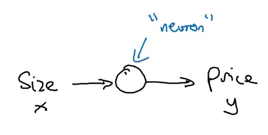
Giả sử ta có 1 bài toán là dự đoán giá của 1 căn nhà bất kỳ, khi ta đã biết được kích thước và giá tiền của 1 số căn nhà khác – giả sử là 6 căn. Đầu tiên ta sẽ vẽ 1 biểu đồ cho dễ hình dung:



Ở trục ngang, ta có kích thước của căn nhà, ở trục dọc, ta có giá tương ứng của nó. Một mô hình hồi quy tuyến tính sẽ giải quyết bài toán này bằng cách vẽ 1 đường thẳng sau cho nó “khớp” với dữ liệu:



Vì vậy, đầu vào ở đây là kích thước căn nhà và đầu ra chính là giá của nó. Bây giờ thử xem ta sẽ giải quyết bài toán như nào khi sử dụng 1 mô hình mạng neural đơn giản:

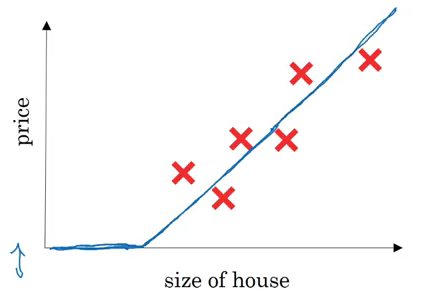


Ở đây, một neural sẽ lấy 1 giá trị đầu vào, biến đổi nó một chút bằng một số hàm toán học, rồi sau đó xuất ra 1 giá trị đầu ra tương ứng. Một trong những hàm thông dụng nhất được sử dụng là ReLU (Rectified Linear Unit):

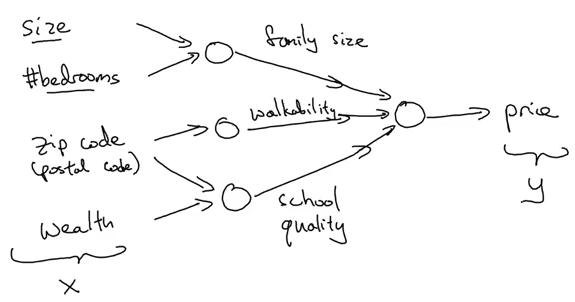


ReLU sẽ lấy 1 số thực z làm đầu vào, rồi sau đó trả về max(0, z). Vì vậy, đầu vào là 10, thì đầu ra sẽ là 10; ngược lại nếu đầu vào là -10, thì đầu ra sẽ là 0. Tạm vậy đã, ta sẽ bàn chi tiết về hàm này sau.

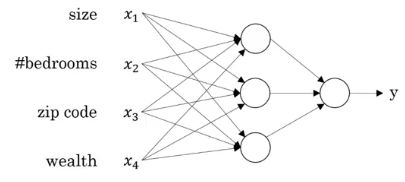
Giờ ta sẽ quay lại với ví dụ trước, nếu ta sử dụng hàm ReLU để dự đoán giá dựa trên kích thước của căn nhà, thì sự dự đoán có thể trông sẽ như thế này:



Ví dụ trên cho thấy một mạng neural với 1 neural duy nhất – nghĩa là khi ta chỉ có 1 đặc tính để dự đoán (kích thước của căn nhà). Nhưng trong thực tế, ta cần xem xét nhiều đặc tính hơn, như là số lượng phòng ngủ, vị trí, … và nhiều đặc điểm khác nữa. Vậy ta cần phải định nghĩa mạng neural như thế nào trong trường hợp đầy rắc rối này nhỉ?



Vấn đề đã bắt đầu trở nên rắc rối dần rồi. Như trên hình vẽ bên trên, ta nhập 4 giá trị vào mạng, tạm gọi là x, mạng sẽ tự động định nghĩa 1 vài đặc tính khác từ đầu vào x, và từ đó xuất gia giá trị y. Đây chính là 1 mô hình mạng neural với 4 đầu vào và 1 đầu ra, và trông nó sẽ như thế này:



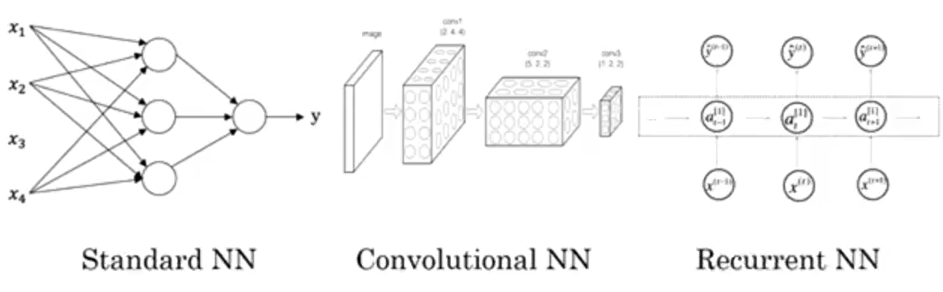
Giờ ta đã hiểu 1 cách trực quan về mạng neural, ta sẽ xem xem chúng ta có thể dùng nó như thế nào cho bài toán học có giám sát.

### Học có giám sát với Mạng Neural

Học có giám sát là 1 bài toán khi ta cần tìm 1 hàm để liên kết giữa đầu vào và đầu ra tương ứng (đã được cho sẵn các cặp đầu vào – đầu ra tương ứng). Ta cần tạo ra 1 đầu ra với mỗi đầu vào cho trước và ta sẽ dạy mô hình bằng những ví dụ có sẵn. Dưới đây là 1 bảng khá dễ hiểu về sự khác nhau giữa ứng dụng của Học có giám sát và sự khác nhau giữa các loại mạng neural để giải quyết các bài toán này:

///table

Dưới đây là là 1 số loại Mạng Neural thường dùng:



Trong khuôn khổ bài báo cáo này, ta sẽ tập trung vào SNN – Standard Neural Network.

Bạn có thể nhận ra, học có giám sát có thể được sử dụng trên cả dữ liệu có cấu trúc và không có cấu trúc.

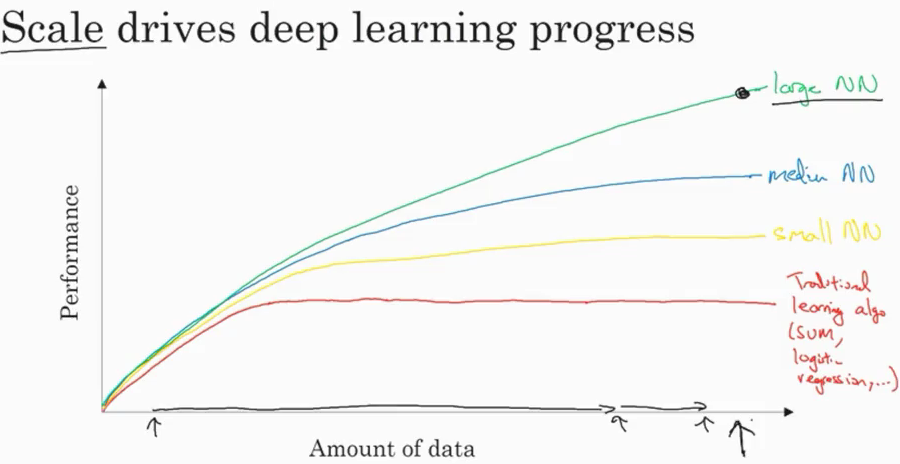
Trong ví dụ trên, dữ liệu đầu cho ta biết lượng phòng ngủ và kích thước căn nhà. Đây chính là dữ liệu có cấu trúc, có nghĩa là với mỗi đặc tính đều có định nghĩa rất rõ ràng.

Ngược với nó, dữ liệu không có cấu trúc thì lại không như vậy. Ví dụ âm thanh, hình ảnh, hay 1 đoạn văn bản, khi mà các đặc tính ta nhận được chỉ là giá trị của các pixel, hoặc từng từ riêng biệt trong cả đoạn văn. Thật khó để nói giá trị của pixel đấy nó đại diện cho bất cứ thứ gì.

Một thuật toán ML hoạt động hiệu quả với dữ liệu có cấu trúc. Nhưng khi gặp bài toán với dữ liệu không có cấu trúc, thì hiệu suất của nó bị giảm đi khá nhiều. Đây chính là sân khấu để Mạng Neural thể hiện được những ưu điểm của mình. Nó hoạt động thật sự hiệu quả với dữ liệu khoogn cấu trúc. Hầu hết những nghiên cứu đột phá ngày nay đều có lõi là 1 mạng neural.

### Tại sao DL lại phát triển như vậy?

Để trả lời được câu hỏi này, ta thử nhìn vào biểu đồ ở dưới:



Khi lượng dữ liệu tăng, hiệu suất của những thuật toán học truyền thống, như là SVM và hồi quy logic, đều không tăng. Thực tế là nó sẽ chững lại tại 1 thời điểm nào đó. Trong trường hợp của mạng neural, nếu bạn có càng nhiều dữ liệu, thì nếu ta có càng nhiều dữ liệu, nó sẽ càng tăng hiệu suất.

Có 3 thang đo để đánh giá 1 quá trình DL:

1. Dữ liệu

2. Thời gian tính toán

3. Thuật toán

Để cải thiện thời gian tính toán của mô hình, hàm toán học đóng vai trò quan trọng. Nếu ta sử dụng hàm sigmoid, thì ta sẽ thu được:



Độ dốc của hàm này ở điểm kết thúc nhận giá trị tiệm cận 0. Vì vậy, các tham số được cập nhật vô cùng chậm, cho ra kết quả là việc học của máy diễn ra rất chậm. Vì thế nên, việc chuyển từ hàm sigmoid sang ReLU là 1 trong những đột phá lớn nhất trong mạng neural. ReLU cập nhật tham số nhanh hơn nhiều khi x > 0. Đây là cách tiếp cận quan trọng nhất để giảm thiểu thời gian tính toán của mô hình.

## 2.2 Giới thiệu về Deep Learning

### 2.2.1. Hồi quy logic dưới dạng 1 Mạng Neural

### Phân loại nhị phân

Trong bài toán phân loại nhị phân, ta có đầu vào x, ví dụ 1 ảnh, và ta cần phải phân loại nó có phải là ảnh 1 con mèo hay không, ta sẽ định dạng đầu ra là 1 hoặc 0. Vì vậy ta chỉ có 2 giá trị đầu ra, đây là 1 ví dụ về bài toán phân loại nhị phân.

Tất nhiên, ta có thể dùng kỹ thuật phân loại phổ biến nhất: hồi quy logic để giải quyết bài toán này.

Hồi quy logic (Logistic Regression)

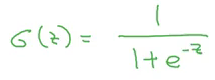
Ta có đầu vào X (ảnh) và muốn tính ra xác suất ảnh đó thuộc lớp 1 (ví dụ con mèo). Với vector X cho trước, đầu ra sẽ là:

y = w(transpose)X + b

Ở đây, ta có w và b là 2 tham số. Vì đầu ra y là 1 giá trị xác suất, nên giá trị của nó nằm giữa 0 và 1. Vì vậy nên Hồi quy logic cũng sử dụng hàm sigmoid để xuất ra giá trị xác suất của y:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/10/Screenshot-from-2018-10-12-15-17-08.png

Với giá trị input bất kỳ, nó chỉ trả về giá trị nằm giữa 0 và 1. Công thức cho hàm sigmoid là:



Vì vậy, nếu z rất lớn, exp(-z) sẽ tiệm cận 0, và vì vậy đầu ra của hàm sigmoid sẽ là 1. Tương tự, nếu z rất nhỏ, ta sẽ được giá trị 0 ở đầu ra.

Lưu ý rằng tham số w là vector n chiều, và b là 1 giá trị thực. Giờ ta thử tính xem độ phức tạp của hàm hồi quy logic.

### Độ phức tạp hàm Hồi quy logic

Để train tham số w và b, ta cần hàm tính độ phức tạp. Ta cần tính tham số w và b để khi kết thúc quá trình train, ta xuất ra giá trị y^ càng gần với y càng tốt.

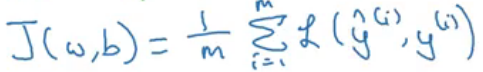
Ta dùng hàm mất mát được định nghĩa như sau:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/10/Screenshot-from-2018-10-12-15-22-09.png

Có 1 vấn đề với hàm này là việc tối ưu hóa trở trên không-lồi, dẫn tới nhiều kết quả tối ưu cục bộ. Vì vậy, độ giảm dốc sẽ không hoạt động tốt với hàm này. Bởi thế nên ta sẽ sử dụng 1 hàm mất mát khác có chức năng tương tự hàm mất mát này và đồng thời giải quyết được vấn đề bằng việc sử dụng hàm lồi:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/10/Screenshot-from-2018-10-12-15-26-05.png

Hàm mất mát được định nghĩa với mỗi ví dụ training cho chúng ta biết hiệu suất hoạt động trong từng trường hợp cụ thể. Mặc khác, hàm độ phức tạp được tính trên toàn bộ tập training. Hàm phức tạp của Hồi quy logic là:



Ta cần hàm phức tạp càng nhỏ càng tốt. Vì thế, ta cần phải tối ưu tham số w và b.

### Dốc giảm dần (Gradient Descent - GD)

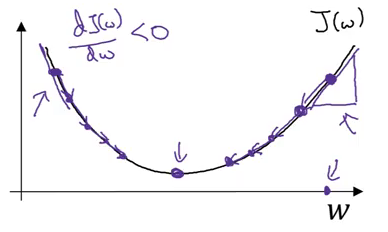
Đây là kỹ thuật giúp ta train được tham số w và b theo cách mà hàm phức tạp đã được tối ưu. Hàm phức tạp ở đây đã lồi tự nhiên – nghĩa là chỉ có 1 cực tiểu, và đó là lí do ta chọn hàm này thay cho hàm lỗi vuông (có nhiều cực tiểu địa phương).

Cùng xem các bước của Gradient Descent:

1. Khởi tạo w và b (thường = 0)

2. Bước 1 bước xuống hướng dốc nhất.

3. Lặp lại bước 2 tới khi đạt cực tiểu.



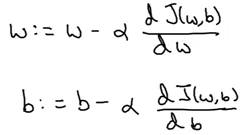
Phương trình cập nhật của GD trở thành:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/10/Screenshot-from-2018-10-15-12-15-19.png

Ở đây, ⍺ là tốc độ học tập điểu khiển độ lớn mà ta nên thực hiện sau mỗi lần lặp.

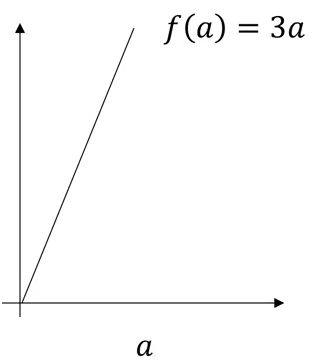
Nếu ta ở bên phải đồ thị trên, độ dốc mang giá trị dương. Sử dụng phương trình cập nhật, ta sẽ đi sang bên trái (hướng xuống dưới) tới khi đạt cực tiểu; và nếu ta ở bên trái thì chỉ cần làm ngược lại. Khá là trực quan phải không?

Phương trình cập nhật cho các tham số của hồi quy logic là:



### Đạo hàm

Giả sử ta có 1 hàm : f(a) = 3a như sau:



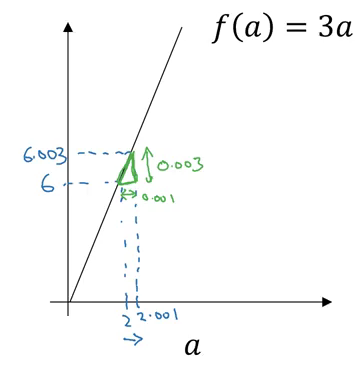
Đạo hàm của hàm này tại điểm bất kỳ đều biểu diễn cho độ đốc của điểm đó.

f(a=2) = 3\*2 = 6

f(a=2.001) = 3\*2.001 = 6.003

Đạo hàm (độ dốc) của hàm tại a = 2 là:

Độ dốc = Chiều cao / chiều rộng



Slope = 0.003 / 0.001 = 3

Đây là cách tính đạo hàm / độ dốc của 1 hàm. Ta thử xem xét thêm vài ví dụ khác về đạo hàm.

Xét 3 hàm dưới đây và đạo hàm tương ứng của nó:

f(a) = a2 , d(f(a))/d(a) = 2a

f(a) = a3, d(f(a))/d(a) = 3a2

f(a) = log(a), d(f(a))/d(a) = 1/a

Tất cả các ví dụ trên, đạo hàm là 1 hàm của a, nghĩa là độ dốc của hàm khác nhau tại các điểm.

### Đồ thị tính toán

Những đồ thị này biểu thị mức tính toán của từng hàm cụ thể. Xem xét ví dụ sau:

J(a,b,c) = 3(a+bc)

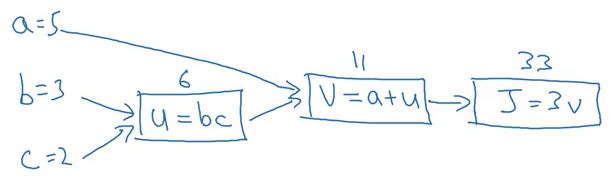
Ta cần phải tính J theo a, b, c. Ta có thể làm theo 3 bước:

u = bc

v = a+u

J = 3v

Sơ đồ hóa cho nó, ví dụ a = 5, b = 3, c = 2:



### Đạo hàm và Đồ thị tính toán

Giờ ta sẽ tìm hiểu tại sao ta có thể tính đạo hàm bằng cách sử dụng đồ thị tính toán. Giả sử ta cần tính dJ/da, ta cần:

1. Vì J là hàm của v, tính dJ/dv:

dJ/dv = d(3v)/dv = 3

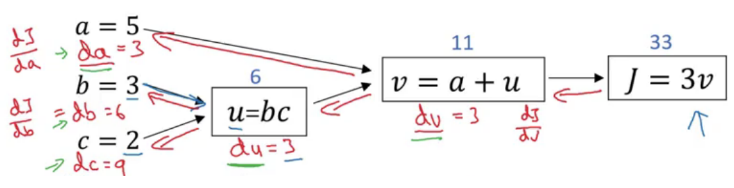
2. Vì v là hàm của a và u, tính dv/da

dv/da = d(a+u)/da = 1

3. Tính dJ/da

dJ/da = (dJ/dv)\*(dv/da) = 3\*1 = 3

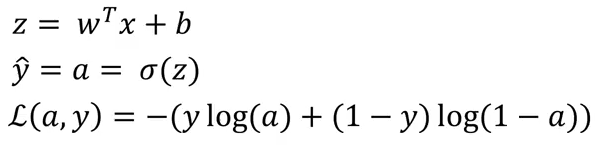
Tương tự, ta có thể tính dJ/db và dJ/dc:



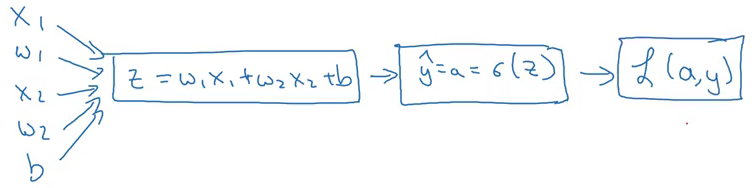
Giờ ta sẽ mang lấy khái niệm của biểu đồ tính toán và GD cùng với nhau và xem tham số của hồi quy logic được cập nhật như thế nào.

### Logistic Regression Gradient Descent (Dốc hồi quy tuyến tính)

Phương trình hồi quy tuyến tính là:



Ở đây L là hàm mất mát. Giờ, với 2 đặc tính (x1, x2), đồ thị tính toán để tính toán mất mát sẽ là:



Bây giờ thì w1, w2 là tham số cần cập nhật. Để làm được điều này ta cần theo các bước sau:

1. Tính da

da = dL/da = (-y/a) + (1-y)/(1-a)

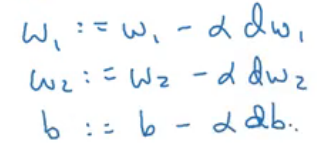
2. Tính dz

dz = (dL/da)\*(da/dz) = [(-y/a) + (1-y)/(1-a)]\*[a(1-a)] = a-y

3. Tính dw1

dw1 = [(dL/da)\*(da/dz)]\*dz/dw1 = (a-y)\*dz/dw1

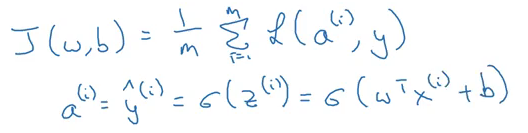
Tương tự, ta có thể tính được dw2 và db. Cuối cùng, trọng số có thể cập nhật dựa trên phương trình sau:



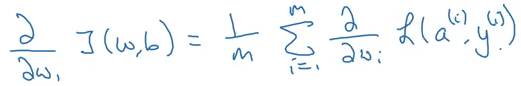
Hãy nhớ rằng đây chỉ là 1 ví dụ training đơn. Ta sẽ xem xét ví dụ đa training. Vì vậy, ta thử tính GD với 1 ví dụ đa training.

### Dốc trên đa training

Ta có thể định nghĩa hàm dự đoán và hàm phức tạp cho ví dụ m-training là:



Đạo hàm của hàm mất mát w.r.t có thể được viết dưới dạng:



Ta có thể áp dụng hồi quy logic cho ví dụ m-training:

J = 0; dw1 = 0; dw2 =0; db = 0;

w1 = 0; w2 = 0; b=0;

for i = 1 to m

    # Forward pass

    z(i) = W1\*x1(i) + W2\*x2(i) + b

    a(i) = Sigmoid(z(i))

    J += (Y(i)\*log(a(i)) + (1-Y(i))\*log(1-a(i)))

    # Backward pass

    dz(i) = a(i) - Y(i)

    dw1 += dz(i) \* x1(i)

    dw2 += dz(i) \* x2(i)

    db  += dz(i)

J /= m

dw1/= m

dw2/= m

db/= m

# Gradient descent

w1 = w1 - alpa \* dw1

w2 = w2 - alpa \* dw2

b = b - alpa \* db

Những vòng lặp này làm cho việc tính toán trở nên rất chậm. Có 1 cách thay thế các vòng lặp cho nó chạy tuần tự sẽ làm cho việc tính toán hiệu quả hơn nhiều. Ta sẽ xem xét nó trong phần sau.

# Phần II. Python và Vector hóa

Thay vì sử dụng những vòng lặp như ví dụ trên, ta có thể sử dụng Vector hóa – là 1 phương pháp rất hiệu quả.

### Vector hóa

Vector hóa về căn bản là 1 cách loại bỏ vòng lặp trong code. Nó thực hiện tất cả tác vụ cùng nhau cho ví dụ m-training thay vì tính toán chúng riêng biệt. Ta thử xem xét biểu diễn Vector hóa và không được Vector hóa của hồi quy logic:

Dạng không Vector hóa:

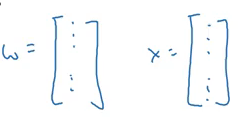
z = 0

for i in range(nx):

   z += w[i] \* x[i]

z +=b

Giờ cùng xem xét dạng Vector hóa. Ta có thể viết w và x dưới dạng Vector:



Ta có thể tính Z cho tất cả các ví dụ traning bằng:

Z = np.dot(W,X)+b (numpy is imported as np)

Hàm dot của thư viện NumPy mặc định sử dụng Vector hóa. Đây là cách ta Vector hóa. Bây giờ ta sẽ tìm cách Vector hóa cả thuật toán hồi quy logic.

### Vector hóa Hồi quy logic

Tiếp tục với ví dụ m-training, bước đầu tiên, ta tính Z:

Z = np.dot(W.T, X) + b

Trong phương trình trên, X là ma trận đặc tính, W là ma trận hệ số. Bước tiếp theo, tính output(A), làm hàm sigmoid của Z:

A = 1 / 1 + np.exp(-Z)

Giờ ta tính mất mát và tối thiểu hóa mất mát:

dz = A – Y

Cuối cùng, ta sẽ tính đạo hàm của tham số và cập nhật chúng:

dw = np.dot(X, dz.T) / m

db = dz.sum() / m

W = W – ⍺dw

b = b – ⍺db

### Khai triển trên Python

Xem xét các ví dụ:

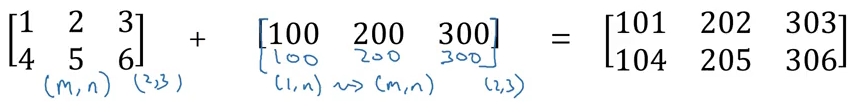
obj.sum(axis = 0): tính tổng các cột

obj.sum(axis = 1): tính tổng các hàng

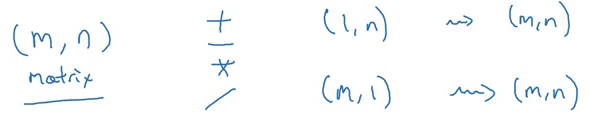
obj.reshape(1,4): thay đổi kích thước ma trận bằng cách truyền giá trị



Nếu ta cộng 100 cho ma trận (4x1), Python sẽ tạo ra ma trận toàn 100 với kích thước (4x1). Tương tự, với ví dụ sau, ma trận 1x3 sẽ được sao chép sang ma trận 2x3:



Nguyên tắc chung sẽ là:

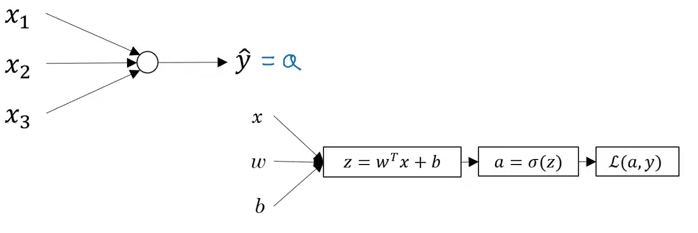


Nếu ta cộng, trừ, nhân hay chia 1 ma trận mxn với 1 ma trận 1xn, nó sẽ sao chép m lần để được ma trận mxn. Điều này làm cho việc tính toán trở nên nhanh hơn rất nhiều.

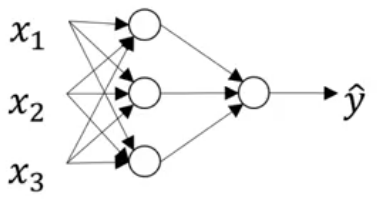
### 2.3 Mạng Neural Nông

### Tổng quan mạng Neural

Trong Hồi quy logic, để tính đầu ra y = a, ta cần sử dụng đồ thị tính toán sau:



Trong trường hợp mạng neural với 1 lớp ẩn, cấu trúc sẽ được thiết kế theo mô hình:



Và đồ thị tính toán để tính toán đầu ra sẽ là:

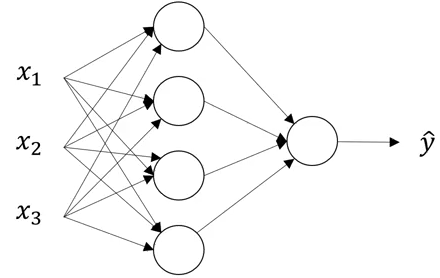
X1  \

X2   => z1 = XW1 + B1 => a1 = Sigmoid(z1) => z2 = a1W2 + B2 => a2 = Sigmoid(z2) => l(a2,Y)

X3  /

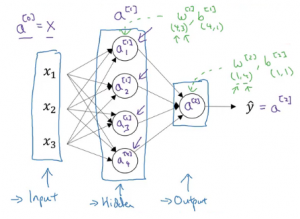
Đại diện mạng lưới Neural

Xem xét đại diện sau của 1 mạng neural:



Bạn có thể tính được số lớp của mạng neural trên không? Nhớ rằng khi đếm số lớp của một mạng neural, ta không tính lớp đầu vào. Vì vậy, có 2 lớp trong mạng neural trên, ví dụ 1 lớp ẩn (giữa) và lớp đầu ra.

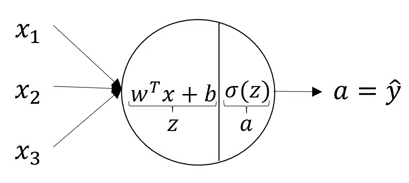
Lớp đầu tiên được định nghĩa là a0, lớp thứ 2 là a1, lớp cuối cùng là a2. Các tham số tương ứng là w1, b1 và w1, b2:



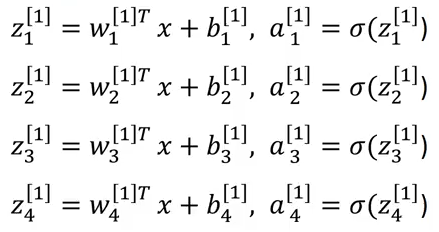
Đây chính là mô hình của 1 mạng neural. Tiếp theo ta sẽ học cách tính đầu ra bằng cách sử dụng 1 mạng neural.

### Tính đầu ra 1 Mạng Neural

Ta cùng tìm hiểu chi tiết cách mà một mạng Neural hoạt động. Mỗi neural lấy 1 input, thực hiện vài tính toán trên chúng (tính z = w[T] + b) sau đó áp dụng hàm sigmoid:



Bước này thực hiện bởi từng neural riêng biệt, phương trình cho lớp ẩn đầu tiên với 4 neural sẽ là:



Vì vậy, với input X, ouput của mỗi neural sẽ là:

z[1] = W[1]x + b[1]

a[1] = 𝛔(z[1])

z[2] = W[2]x + b[2]

a[2] = 𝛔(z[2])

Để tính những ouput này, ta cần chạy 1 vòng lặp tính từng giá trị riêng của từng neural. Nhắc lại rằng gọi vòng lặp sẽ khiến việc tính toán chậm hơn, vì thế ta cần tối ưu hóa code để tránh điều này.

### Vector hóa trên nhiều ví dụ

Dạng không vector hóa của output của mạng neural:

for i=1 to m:

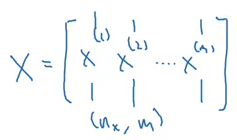
z[1](i) = W[1](i)x + b[1]

a[1](i) = 𝛔(z[1](i))

z[2](i) = W[2](i)x + b[2]

a[2](i) = 𝛔(z[2](i))

Sử dụng vòng lặp, ta đang tính z và a của mỗi ví dụ training riêng biệt. Giờ ta sẽ xem nó được vector hóa như thế nào. Tất cả các ví dụ training sẽ được ghép vào 1 ma trận X duy nhất:



Tại đây, nx là số đặc tính và m là số ví dụ training. Dạng vector hóa của tính toán đầu ra là:

Z[1] = W[1]X + b[1]

A[1] = 𝛔(Z[1])

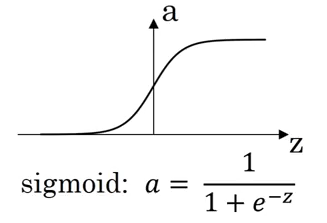
Z[2] = W[2]X + b[2]

A[2] = 𝛔(Z[2])

Điều này sẽ giảm thời gian tính toán (trong hầu hết các trường hợp).

### Hàm hoạt động (Activation Function)

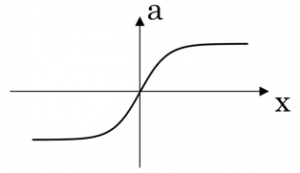
Khi tính toán đầu ra, một AF được áp dụng. Việc chọn hàm ảnh hưởng rất nhiều tới hiệu suất của mô hình. Vì vậy, chúng ta sử dụng hàm kích hoạt sigmoid:



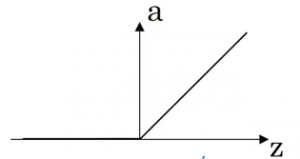
Tuy nhiên, đây không phải là lựa chọn tối ưu trong một số trường hợp. Bởi vì khi z tăng thì đạo hàm sẽ tiến dần tới 0, và vì vậy các tham số sẽ được cập nhật rất chậm.

May thay, ta có một vài hàm khác để thay thế:

- tanh:



- ReLU:



///bảng

Ta có thể chọn các hàm hoạt động khác nhau phụ thuộc vào bài toán đang cần giải.

### Tại sao ta lại cần hàm hoạt động phi tuyết tính?

Nếu ta sử dụng hàm tuyến tính trên đầu ra của lớp, nó sẽ coi đầu ra là 1 hàm tuyến tính của đầu vào:

Z = WX + b

Trong trường hợp hàm hoạt động tuyến tính, đầu ra sẽ bằng Z:

A = Z

Sử dụng một hàm tuyến tính về cơ bản là vô dụng. Thành phần của 2 hàm tuyến tính, tự bản thân nó đã là 1 hàm tuyến tính, và trừ khi ta sử dụng 1 hàm phi tuyến tính thì ta mới tạo ra 1 vài hàm thú vị hơn. Đó là lí do tại sao hầu hết các chuyên gia đều sử dụng hàm hoạt động phi tuyến tính.

Vẫn có 1 vài trường hợp ta cần sử dụng hàm hoạt động tuyến tính. Giả sử ta cần dự đoán giá của 1 căn nhà, đầu ra sẽ nằm trong (0, 1) và (-1, 1) tương ứng. Nhưng giá căn nhà cũng có thể lớn hơn 1. Trong trường hợp này, ta sẽ sử dụng hàm hoạt động tuyến tính tại lớp đầu ra.

Một khi ta đã có đầu ra, ta sẽ làm gì tiếp theo? Ta cần thực hiện truyền theo thứ tự để cập nhật các tham số bằng việc sử dụng dốc giảm dần (gradient descent).

### Dốc giảm dần và Mạng Neural

Các tham số ta cần cập nhật trong 2 lớp mạng là w[1], b[1], w[2] và b[2], và hàm phức tạp sẽ được tối ưu là:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/10/Screenshot-from-2018-10-15-19-26-58.png

Các bước đề giảm dần độ dốc được khai triển như sau:

Repeat:

Compute predictions (y'(i), i = 1,...m)

    Get derivatives: dW[1], db[1], dW[2], db[2]

    Update: W[1] = W[1] - ⍺ \* dW[1]

            b[1] = b[1] - ⍺ \* db[1]

            W[2] = W[2] - ⍺ \* dW[2]

            b[2] = b[2] - ⍺ \* db[2]

Ta thử nhìn nhanh các bước cần thực hiện trên mạng 2 lớp:

Chuyển tiếp:

Z[1] = W[1]\*A[0] + b[1]    # A[0] is X

A[1] = g[1](Z[1])

Z[2] = W[2]\*A[1] + b[2]

A[2] = g[2](Z[2])

Truyền:

dZ[2] = A[2] - Y

dW[2] = (dZ[2] \* A[1].T) / m

db[2] = Sum(dZ[2]) / m

dZ[1] = (W[2].T \* dZ[2]) \* g'[1](Z[1])  # element wise product (\*)

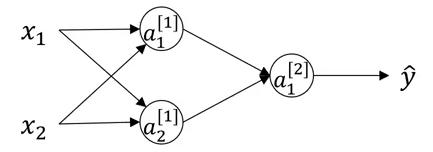
dW[1] = (dZ[1] \* A[0].T) / m   # A[0] = X

db[1] = Sum(dZ[1]) / m

Trên đây là những bước hoàn chỉnh để mạng neural tính toán đầu ra. Lưu ý rằng ta phải khởi trọng số W ở thời điểm bắt đầu và cập nhật trong các bước nền. Vậy nên ta thử xem các trọng số này được khởi tạo như thế nào nhé.

### Khởi tạo ngẫu nhiên

Phần trước ta thấy trọng số được khởi tạo = 0 trong trường hợp hồi quy logic. Nhưng câu hỏi đặt ra là ta có nên khởi tạo như vậy? Xem xét ví dụ sau:



Nếu trọng số được khởi tạo bằng 0. Ma trận W sẽ là:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/10/Screenshot-from-2018-10-15-19-41-52.png

Sử dụng điều kiện:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/10/Screenshot-from-2018-10-15-19-42-44.png

Và cuối cùng ở bước truyền:

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2018/10/Screenshot-from-2018-10-15-19-43-54.png

Dù ta có sử dụng bao nhiêu đơn vị trong 1 lớp đi chăng nữa, ta vẫn sẽ luôn nhận được kết quả đầu ra giống nhau và bằng với việc sử dụng 1 đơn vị. Vì vậy, thay vì khởi tạo trọng số = 0, ta khởi tạo chúng 1 cách ngẫu nhiên:

w[1] = np.random.randn((2,2)) \* 0.01

b[1] = np.zero((2,1))

Ta nhân trọng số với 0.01 để khởi tạo trọng số nhỏ. Nếu ta khởi tạo trọng số lớn, hàm hoạt động sẽ lớn theo, kéo theo độ dốc bằng 0 (trong trường hợp sử dụng hàm sigmoid và hàm tanh). Vì vậy, việc học sẽ bị kéo chậm lại; và ta thì chẳng muốn điều đó xảy ra một chút nào.

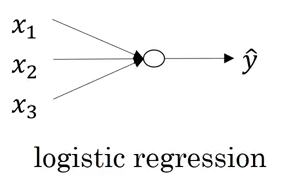
### 2.4 Mạng Neural Sâu (Deep Neural Networks - DNNs)

### Deep L-Layer Neural Network

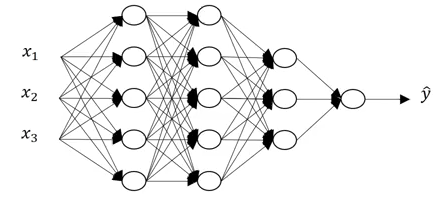
Trong phần này, ta sẽ xem xét việc chuyển tiếp và truyền được áp dụng trong DNNs như thế nào.

Nhưng đầu tiên, ta sẽ giới thiệu khái niệm Mạng Neural Sâu trước đã.

Thế nào là nông, thế nào là sâu? Một Hồi quy logic là một mô hình rất nông và nó chỉ có 1 lớp.



Một mạng neural sâu hơn sẽ có nhiều lớp ẩn hơn:



Một số kí hiệu liên quan lớp DNNs:

L là số lớp.

n[l] là số unit trong lớp l.

a[l] là số hàm hoạt động trong lớp l.

w[l] là trọng số của z[l]

### Truyền chuyển tiếp trong DNNs

Một ví dụ đơn giản, bước truyền chuyển tiếp sẽ được viết dưới dạng:

z[l] = W[l]a[l-1] + b[l]

a[l] = g[l](z[l])

Ta có thể vector hóa những bước này cho m-training như sau:

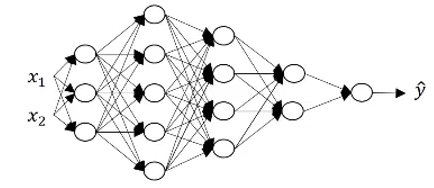
Z[l] = W[l]A[l-1] + B[l]

A[l] = g[l](Z[l])

Những đầu ra này từ một lớp được coi như đầu vào của lớp sau. Ta không thể tính Truyền chuyển tiếp cho tất cả các lớp mà không dùng lớp 1 vòng lặp, nên việc dùng 1 vòng lặp là điều có thể chấp nhận được.

### Bắt đúng kích thước của ma trận

Phân tích kích thước của ma trận là 1 trong những công cụ gỡ rối tốt nhất để kiểm tra xem code của chúng ta có đúng không. Xem xét ví dụ sau:



Có 5 lớp trong mạng neural trên. Trong đó có 4 lớp ẩn và 1 lớp đầu ra. Số unit trong mỗi lớp là:

n[0] = 2, n[1] = 3, n[2] = 5, n[3] = 4, n[4] = 2, and n[5] = 1

Ta khởi tạo kích thước của W, b:

W[l] = (n[l], n[l-1])

b[l] = (n[l], 1)

dW[l] = (n[l], n[l-1])

db[l] = (n[l],1)

Z[l], A[l], dZ[l], dA[l] = (n[l],m)

Nhưng tại sao ta lại cần tới DNNs trong khi ta đã có những giải pháp đơn giản hơn? Ta cùng thử tìm hiểu nhé.

### Tại sao lại cần DNNs?

Trong DNNs, ta có rất nhiều lượng lớp ẩn. Nhưng những lớp ẩn này có tác dụng gì? Để trả lời câu hỏi đó, ta xét hình ảnh sau:



DNNs tìm những mối liên hệ giữa dữ liệu (từ những liên hệ đơn giản cho tới phức tạp). Lớp ẩn đầu tiên sẽ cố gắng tìm những hàm đơn giản như xác định các cạnh trong ảnh trên. Càng vào sâu trong các lớp mạng, những hàm đơn giản này sẽ hợp thành những hàm phức tạp hơn như nhận dạng gương mặt. Một vài ví dụ nữa có thể được liệt kê ra như:

Nhận dạng gương mặt:

Hình ảnh >>> Đường viền >>> Phần mặt >>> Mặt >>> Khuôn mặt mong muốn

Nhận dạng âm thanh:

Âm thanh >>> Các đặc tính cấp thấp (sss, bb) >>> Âm vị >>> Từ >>> Câu

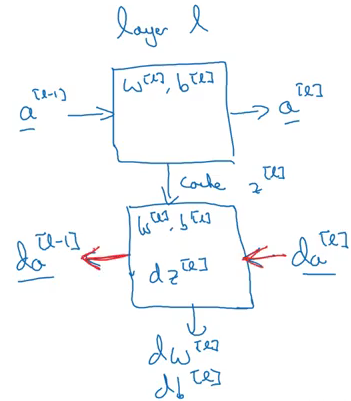
### Xây dựng khối DNNs

Xét 1 lớp bất kỳ trong DNNs. Đầu vào của lớp sẽ là đầu ra của lớp trước; và đầu ra của nó sẽ là kết quả của hàm hoạt động của chính lớp đó.

Input: a[l-1]

Output: a[l]

Lớp này đầu tiên sẽ tính z[l] – số lượng hàm hoạt động sau đó lưu lại. Sau đó với bước truyền chuyển tiếp, nó sẽ tính da[l], ví dụ, đạo hàm của hàm hoạt động ở lớp l, đạo hàm của trọng số weights dw[l], db[l], dz[l], và cuối cùng là da[l-1]:



Đây là cách mỗi khối của DNNs hoạt động. Tiếp theo, ta sẽ xem thử làm sao để khai triển các khối này:

Truyền trước và truyền sau

Đầu vào của 1 bước truyền sau là a[l-1] và đầu ra là a[l], bộ nhớ đệm là z[l], là 1 hàm của w[l] và b[l]. Vì vậy, vector hóa để tính Z[l] và A[l] là:

Z[l] = W[l] \* A[l-1] + b[l]

A[l] = g[l](Z[l])

Ta sẽ tính Z và A trong từng lớp. Sau khi tính hàm hoạt động, bước tiếp theo sẽ là truyền trước, ta sẽ cập nhật trọng số sử dụng đạo hàm. Đầu vào của truyền sau là da[l], và đầu ra là da[l-1], dW[l] và db[l].

Xét phương trình vector hóa của truyền trước:

dZ[l] = dA[l] \* g'[l](Z[l])

dW[l] = 1/m \* (dZ[l] \* A[l-1].T)

db[l] = 1/m \* np.sum(dZ[l], axis = 1, keepdims = True)

dA[l-1] = w[l].T \* dZ[l]

Đây chính là cách ta triển khai DNNs.

DNNs hoạt động hiệu quả 1 cách đáng ngạc nhiên. Chỉ cần chạy 1 vài dòng lệnh và nó sẽ cho ta 1 kết quả như mong đợi. Đó là bởi vì ta đã cấp khối lượng lớn dữ liệu cho mạng và mạng đã học từ dữ liệu đó bằng cách sử dụng các lớp ẩn.