« Bài 13: Softmax Regression (/2017/02/17/softmax/) Bài 15: Overfitting » (/2017/03/04/overfitting/)

Bài 14: Multi-layer Perceptron và Backpropagation

Neural-nets (/tags#Neural-nets) Multi-layer (/tags#Multi-layer) Backpropagation (/tags#Backpropagation)

Feb 24, 2017

Trong trang này:

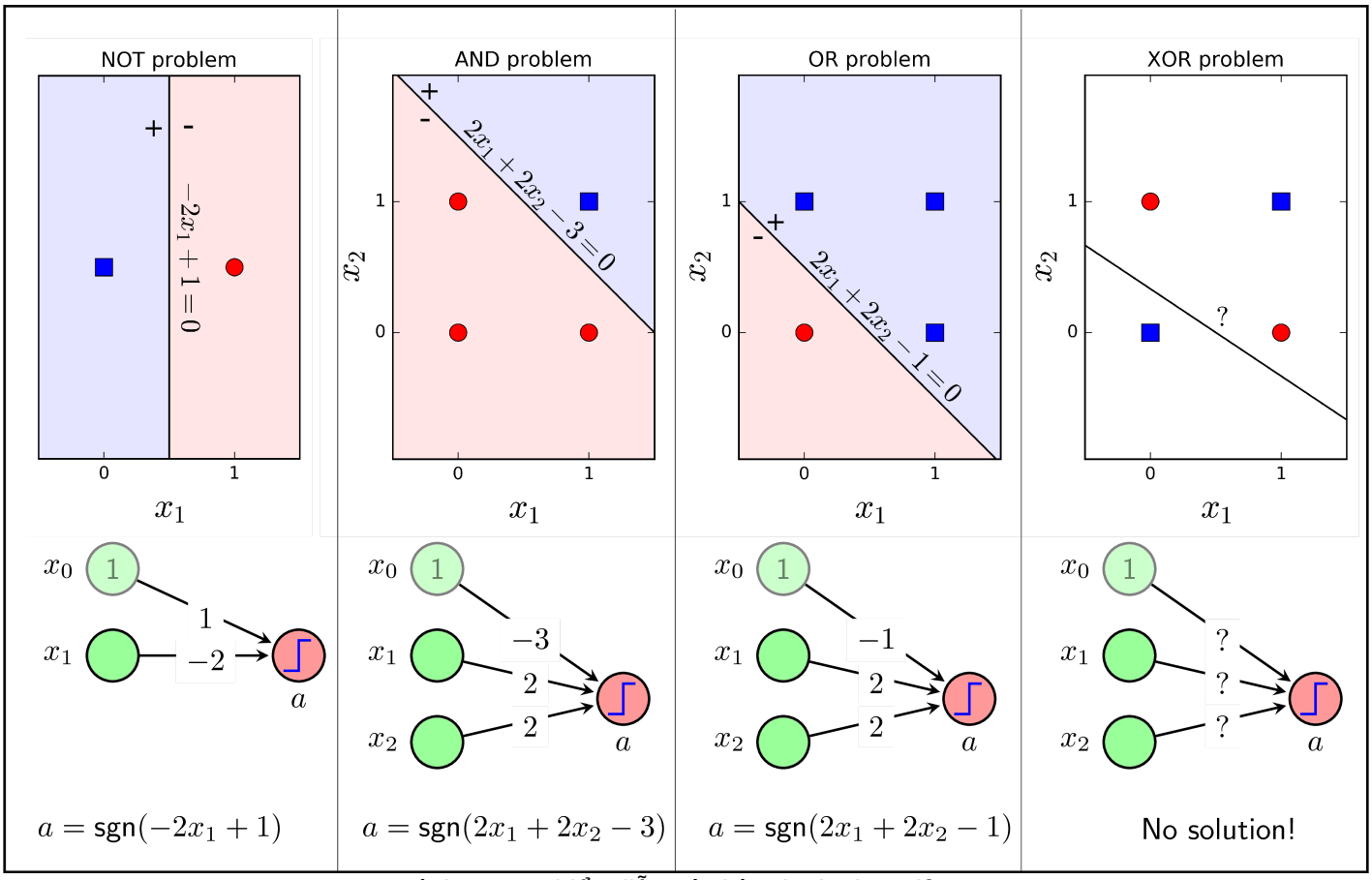
Vì bài này sử dụng khá nhiều công thức toán, bạn đọc được khuyến khích đọc Lưu ý về ký hiệu toán học (/math/#luu-y-ve-ky-hieu).

1. Giới thiệu

Bài toán Supervised Learning (/2016/12/27/categories/#supervised-learning-hoc-co-giam-sat), nói một cách ngắn gọn, là việc đi tìm một hàm số để với mỗi *input*, ta sử dụng hàm số đó để dự đoán *output*. Hàm số này được xây dựng dựa trên các cặp dữ liệu $(\mathrm{x}_i, \mathrm{y}_i)$ trong *training set*. Nếu *đầu ra dự đoán* (predicted output) gần với *đầu ra thực sự* (ground truth (/2017/01/08/knn/#ground-truth)) thì đó được gọi là một thuật toán tốt (nhưng khi *đầu ra dự đoán quá giống với đầu ra thực sự* thì không hẳn đã tốt, tôi sẽ đề cập kỹ về hiện tượng trong bài tiếp theo).

1.1. PLA cho các hàm logic cơ bản

Chúng ta cùng xét khả năng biểu diễn (representation) của Perceptron Learning Algorithm (PLA) (/2017/01/21/perceptron/) cho các bài toán binary vô cùng đơn giản: biểu diễn các hàm số logic NOT, AND, OR, và XOR (https://en.wikipedia.org/wiki/Exclusive_or) (output bằng 1 nếu và chỉ nếu hai input khác nhau). Để có thể sử dụng PLA (output là 1 hoặc -1), chúng ta sẽ thay các giá trị bằng 0 của output của các hàm này bởi -1. Trong hàng trên của Hình 1 dưới đây, các điểm hình vuông màu xanh là các điểm có label bằng 1, các điểm hình tròn màu đỏ là các điểm có label bằng -1. Hàng dưới của Hình 1 là các mô hình perceptron với các hệ số tương ứng.



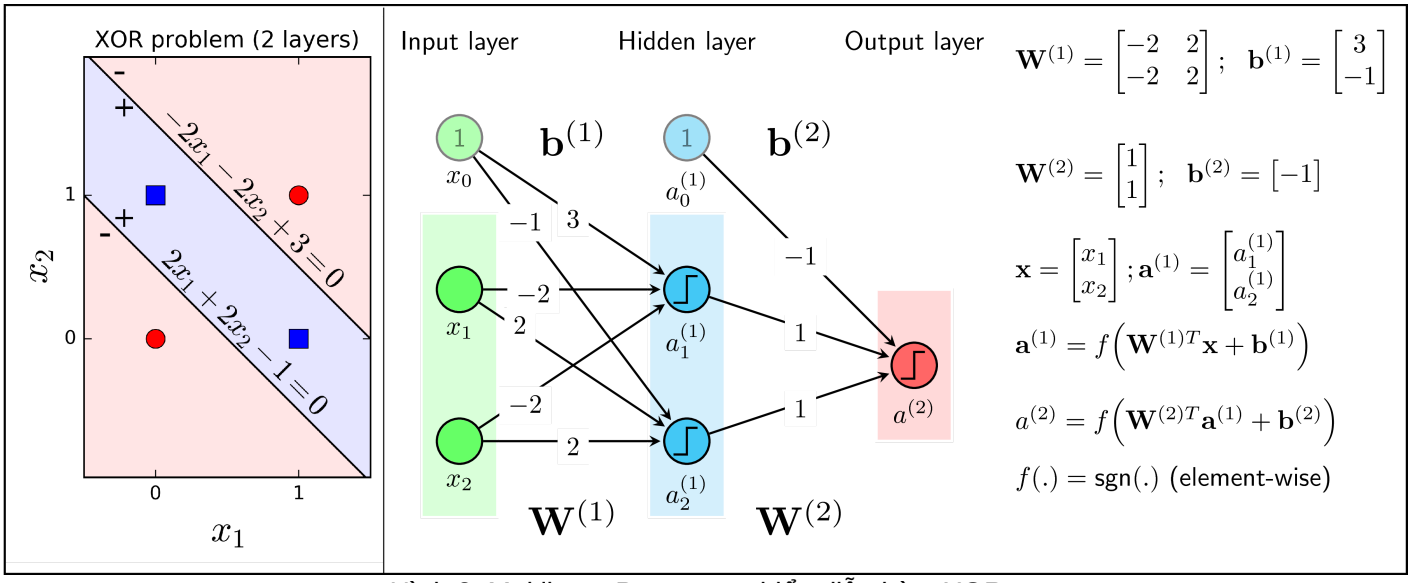
Hình 1: PLA biểu diễn các hàm logic đơn giản.

Nhận thấy rằng với các bài toán OR, AND, và OR, dữ liệu là *linearly separable* (/2017/01/21/perceptron/#bai-toan-perceptron), vì vậy ta có thể tìm được các hệ số cho perceptron giúp biểu diễn chính xác mỗi hàm số. Xem ví dụ 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, predicted output đều giống với ground truth (/2017/01/08/knn/#ground-truth). Bạn đọc có thể tự kiểm chứng các hệ số trong hình với hàm AND và OR.

1.2. Biểu diễn hàm XOR với Neural Network.

Với hàm XOR, vì dữ liệu không *linearly separable*, tức không thể tìm được 1 đường thẳng giúp phân chia hai lớp xanh đỏ, nên bài toán vô nghiệm. Nếu thay PLA bằng Logistic Regression (/2017/01/27/logisticregression/), tức thay hàm activation function từ *sgn* sang *sigmoid* (/2017/01/27/logisticregression/#sigmoid-function), ta cũng không tìm được các hệ số thỏa mãn, vì về bản chất, Logistic Regression cũng chỉ tạo ra các đường biên có dạng tuyến tính (/2017/01/27/logisticregression/#boundary-tao-boi-logistic-regression-co-dang-tuyen-tinh). Như vậy là các mô hình Neural Network chúng ta đã biết không thể biểu diễn được hàm số logic đơn giản này.

Nhận thấy rằng nếu cho phép sử dụng hai đường thẳng, bài toán biểu diễn hàm XOR sẽ được giải quyết như Hình 2 (trái) dưới đây:



Hình 2: Multilayer Perceptron biểu diễn hàm XOR

Các hệ số tương ứng với hai đường thẳng trong Hình 2 (trái) được minh họa trên Hình 2 (phải) tại các node màu xanh (có hai loại màu xanh). Đầu ra $a_1^{(1)}$ bằng 1 với các điểm nằm về phía (+) của đường thẳng $-2x_1 - 2x_2 + 3 = 0$, bằng -1 với các điểm nằm về phía (-) của đường thẳng này. Tương tự, đầu ra $a_2^{(1)}$ bằng 1 với các điểm nằm về phía (+) của đường thẳng $2x_1 + 2x_2 - 1 = 0$.

Như vậy, hai đường thẳng 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 input của một PLA khác. Trong PLA mới này, input là các node màu lam (đừng quên node bias có giá trị bằng 1), output là các node màu đỏ. Các hệ số được cho trên Hình 2 (phải). Kiểm tra lại một chút, với các điểm hình vuông xanh (hình trái), $a_1^{(1)} = a_2^{(1)} = 1$, khi đó $a^{(2)} = \text{sgn}(1 + 1 - 1) = 1$. Với các điểm hình tròn đỏ, $a_1^{(1)} = -a_2^{(1)}$, vậy nên $a^{(2)} = \text{sgn}(a_1^{(1)} + a_2^{(1)} - 1) = \text{sgn}(-1) = -1$. Trong cả hai trường hợp, predicted ouput đều giống với ground truth. Vậy, nếu ta sử dụng 3 PLA tương ứng với các output $a_1^{(1)}, a_2^{(1)}, a^{(2)}$, ta sẽ biểu diễn được hàm XOR.

Ba PLA kể trên được xếp vào hai *layers*. Layer thứ nhất: input - lục, output - lam. Layer thứ hai: input - lam, output - đỏ. Ở đây, output của layer thứ nhất chính là input của layer thứ hai. Tổng hợp lại ta được một mô hình mà ngoài layer input (lục) và output (đỏ), ta còn có một layer nữa (lam). Mô hình này có tên gọi là Multi-layer Perceptron (MLP). Layer trung gian ở giữa còn được gọi là *hidden layer*.

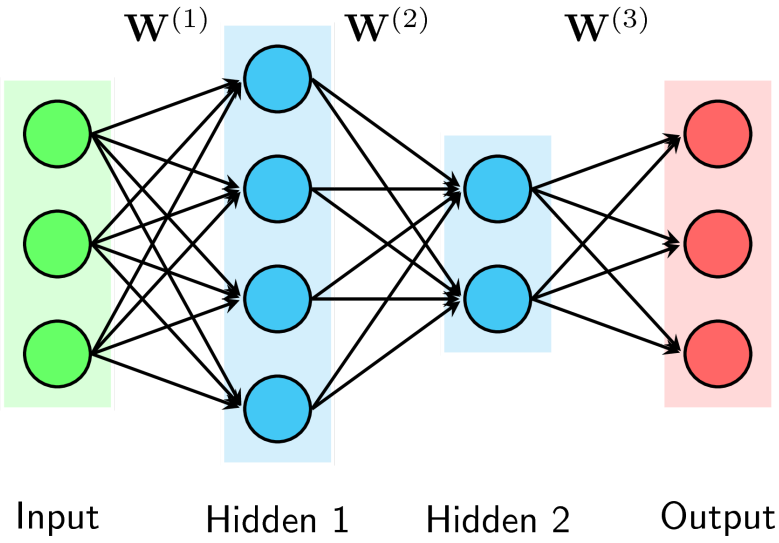
Một vài lưu ý:

- Perceptron Learing Algorithm là một trường hợp của *single-layer neural network* với *activation fucntion* (/2017/01/27/logisticregression/#nhac-lai-hai-mo-hinh-tuyen-tinh) là hàm *sgn*. Trong khi đó, Perceptron là tên chung để chỉ các Neural Network với chỉ một input layer và một output tại output layer, không có hidden layer.
- Các *activation function* có thể là các nonlinear function khác, ví dụ như *sigmoid function* (/2017/01/27/logisticregression/#sigmoid-function) hoặc *tanh function* (/2017/01/27/logisticregression/#tanh-function). Các *activation function* phải là nonlinear (phi tuyến), vì nếu không, nhiều layer hay một layer cũng là như nhau. Ví dụ với hai layer trong Hình 2, nếu *activation function* là một hàm linear (giả sử hàm $f(s) = s$), thì cả hai layer có thể được thay bằng một layer với ma trận hệ số $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ (tạm bỏ qua biases).
- Để cho đơn giản, tôi đã sử dụng ký hiệu $\mathbf{W}^{(l)T}$ để thay cho $(\mathbf{W}^{(l)})^T$ (ma trận chuyển vị). Trong Hình 2 (phải), tôi sử dụng ký hiệu ma trận $\mathbf{W}^{(2)}$, mặc dù đúng ra nó phải là vector, để biểu diễn tổng quát cho trường hợp output layer có thể có nhiều hơn 1 node. Tương tự với bias $\mathbf{b}^{(2)}$.
- Khác với các bài trước về Neural Networks, khi làm việc với MLP, ta nên tách riêng phần biases và ma trận hệ số ra. Điều này đồng nghĩa với việc vector input $\mathbf{x}$ là vector KHÔNG mở rộng.

2. Các ký hiệu và khái niệm

2.1. Layers

Ngoài *Input layers* và *Output layers*, một Multi-layer Perceptron (MLP) có thể có nhiều *Hidden layers* ở giữa. Các *Hidden layers* theo thứ tự từ input layer đến output layer được đánh số thứ thự là *Hidden layer 1*, *Hidden layer 2*, … Hình 3 dưới đây là một ví dụ với 2 Hidden layers.



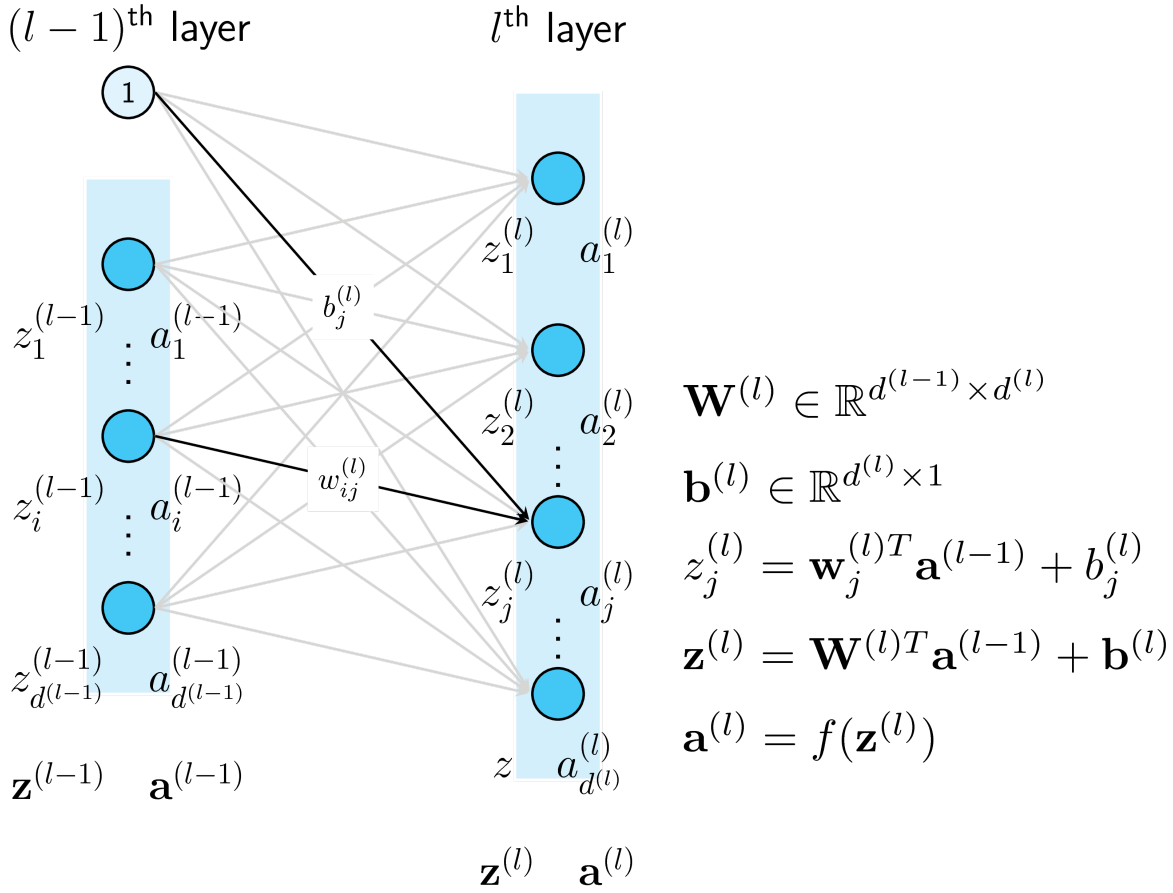
Hình 3: MLP với hai hidden layers (các biases đã bị ẩn).

Số lượng layer trong một MLP được tính bằng số hidden layers cộng với 1. Tức là khi đếm số layers của một MLP, ta không tính input layers. Số lượng layer trong một MLP thường được ký hiệu là L. Trong Hình 3 trên đây, $L = 3$.

2.2. Units

Một *node* hình tròn trong một layer được gọi là một unit. Unit ở các input layer, hidden layers, và output layer được lần lượt gọi là input unit, hidden unit, và output unit. Đầu vào của các hidden layer được ký hiệu bởi z, đầu ra của mỗi unit thường được ký hiệu là a (thể hiện *activation*, tức giá trị của mỗi unit sau khi ta áp dụng activation function lên z). 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 $\mathbf{a}^{(l)} \in \mathbb{R}^{d^{(l)}}$.

Khi làm việc với những Neural Networks phức tạp, cách tốt nhất để hạn chế lỗi là viết cụ thể chiều của mỗi ma trận hay vector ra, bạn sẽ thấy rõ hơn trong phần sau.



Hình 4: Các ký hiệu sử dụng trong MLP.

2.3. Weights và Biases

Có L ma trận trọng số cho một MLP có L layers. Các ma trận này được ký hiệu là $\mathrm{W}^{(l)} \in \mathbb{R}^{d^{(l-1)} \times d^{(l)}}, l = 1, 2, \ldots, L$ trong đó $\mathrm{W}^{(l)}$ 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 biases của layer thứ (l) được ký hiệu là $\mathrm{b}^{(l)} \in \mathbb{R}^{d^{(l)}}$. Các trọng số này được ký hiệu như trên Hình 4. Khi tối ưu một MLP cho một công việc nào đó, chúng ta cần đi tìm các weghts và biases này.

Tập hợp các weights và biases lần lượt được ký hiệu là W và b.

2.4. Activation functions

(Phần này chú yếu được dịch lại từ: http://cs231n.github.io/neural-networks-1/ (CS231n Convolutional Neural Networks for Visual Recognition))

Mỗi output của một unit (trừ các input units) được tính dựa vào công thức:

$$a_i^{(l)} = f(\mathbf{w}_i^{(l)T} \mathbf{a}^{(l-1)} + b_i^{(l)})$$

Trong đó $f(.)$ là một (nonlinear) activation function. Ở dạng vector, biểu thức bên trên được viết là:

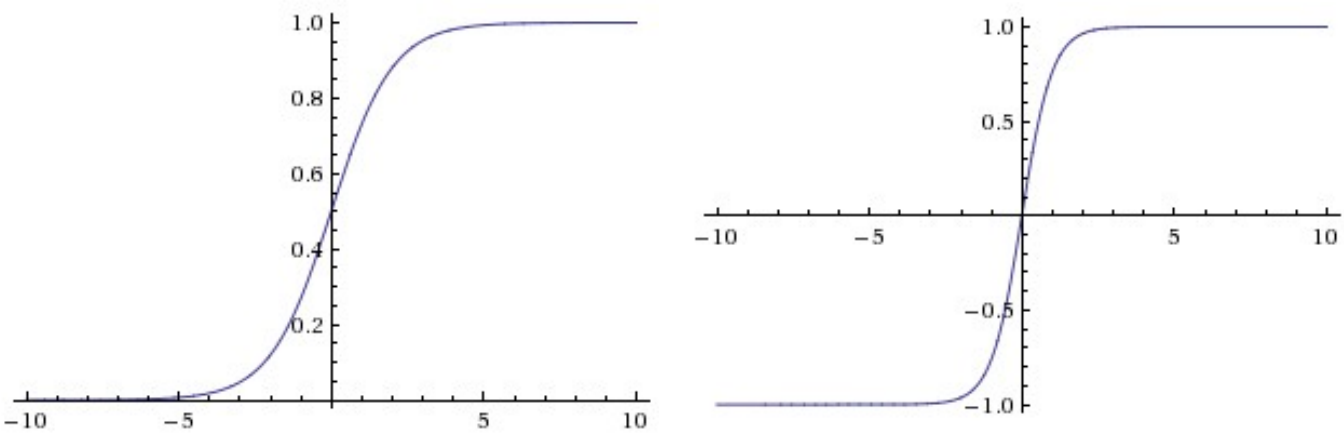
$$\mathrm{a}^{(l)} = f(\mathrm{W}^{(l)T}\mathrm{a}^{(l-1)} + \mathrm{b}^{(l)})$$

Khi activation function $f(.)$ được áp dụng cho một ma trận (hoặc vector), ta hiểu rằng nó được áp dụng cho *từng thành phần của ma trận đó*. Sau đó các thành phần này được sắp xếp lại đúng theo thứ tự để được một ma trận có kích thước bằng với ma trận input. Trong tiếng Anh, việc áp dụng lên từng phần tử như thế này được gọi là *element-wise*.

2.4.1. Hàm *sgn* không được sử dụng trong MLP

Hàm *sgn* (còn gọi là *hard-threshold*) chỉ được sử dụng trong PLA, mang mục đích giáo dục nhiều hơn. Trong thực tế, hàm *sgn* không được sử dụng vì hai lý do: đầu ra là *discrete*, và đạo hàm tại hầu hết các điểm bằng 0 (trừ điểm 0 không có đạo hàm). Việc đạo hàm bằng 0 này khiến cho các thuật toán gradient-based (ví dụ như Gradient Descent (/2017/01/12/gradientdescent/_)) không hoạt động!

2.4.1 Sigmoid và tanh



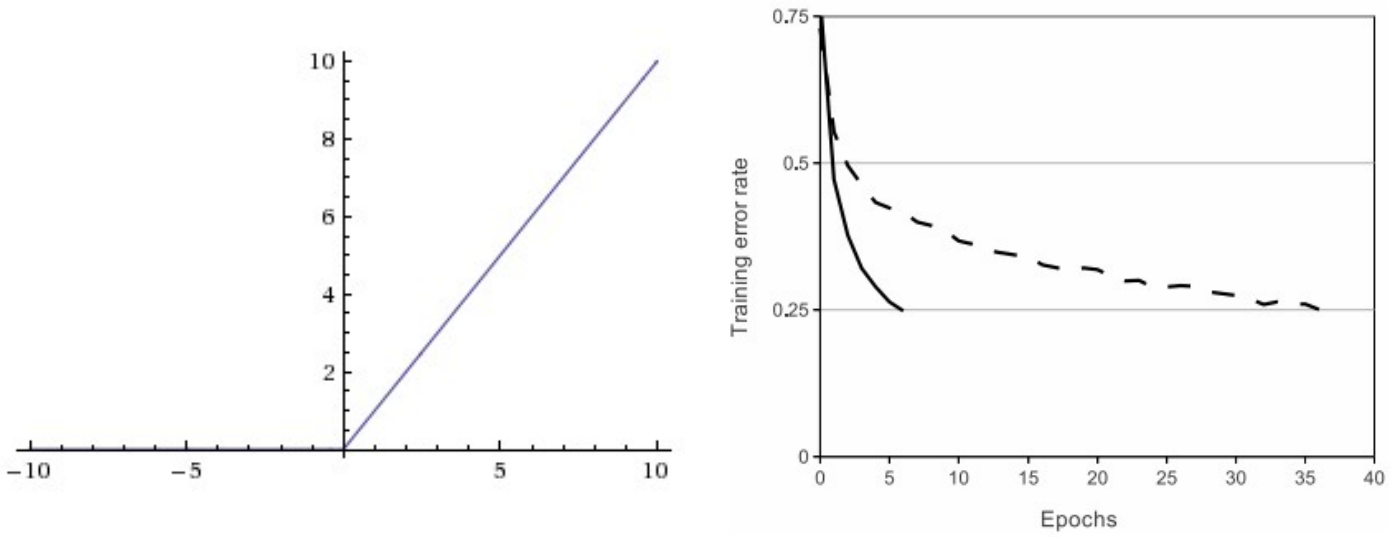
Hình 5: Hàm *sigmoid* (trái) và *tanh* (phải). (Nguồn CS231n Convolutional Neural Networks for Visual Recognition (https://cs231n.github.io/neural-networks-1/))

Hàm *sigmoid* có dạng $f(s) = 1/(1+\exp(-s))$ với đồ thị như trong Hình 5 (trái). 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. Hàm số này được sử dụng nhiều trong quá khứ ví có đạo hàm rất *đẹp*. Những năm gần đây, hàm số này ít khi được sử dụng. Nó có một nhược điểm cơ bản:

- *Sigmoid saturate and kill gradients*: Một nhược điểm dễ nhận thấy là khi đầu vào có trị tuyệt đối lớn (rất âm hoặc rất dương), gradient của hàm số này 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. Bạn đọc sẽ hiểu rõ hơn phần này trong phần Backpropagation.

Hàm *tanh* cũng có nhược điểm tương tự về việc gradient rất nhỏ với các đầu vào có trị tuyệt đối lớn.

2.4.3. ReLU



Hình 5: Hàm ReLU và tốc độ hội tụ khi so sánh với hàm tanh.

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 (trái)). Nó có công thức toán học $f(s) = \max(0, s)$ - rất đơn giản. Ưu điểm chính của nó là:

- ReLU được chứng minh giúp cho việc training các *Deep Networks* nhanh hơn rất nhiều (theo Krizhevsky et al. (http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf)). Hình 5 (phải) so sánh sự hội tụ của SGD khi sử dụng hai activation function khác nhau: 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 với gradient bằng 1 nếu đầu vào lớn hơn 0, bằng 0 nếu đầu vào nhỏ hơn 0.
- Mặc dù hàm ReLU không có đạo hàm tại $s = 0$, trong thực nghiệm, người ta vẫn thường định nghĩa $\mathrm{ReLU}'(0) = 0$ và khẳng định thêm rằng, xác suất để input của một unit bằng 0 là rất nhỏ.

Hàm ReLU có nhiều biến thể khác như Noisy ReLU, Leaky ReLu, ELUs (https://en.wikipedia.org/wiki/Rectifier_(neural_networks)). Tôi xin phép dừng phần này ở đây vì chưa có ý định đi sâu vào Deep Neural Networks.

2.4.4. Một vài lưu ý

- Output layer nhiều khi không có activation function mà sử dụng trực tiếp giá trị đầu vào $z_i^{(l)}$ của mỗi unit. Hoặc nói một cách khác, activation function chính là hàm *identity*, tức đầu ra bằng đầu vào. Với các bài toán classification, output layer thường là một Softmax Regression

(/2017/02/17/softmax/) layer giúp tính xác suất để một điểm dữ liệu rơi vào mỗi class.

- Mặc dù activation function cho mỗi unit có thể khác nhau, trong cùng một network, activation như nhau thường được sử dụng. Điều này giúp cho việc tính toán được đơn giản hơn.

3. Backpropagation

Phần này khá nặng về Đại Số Tuyến Tính, bạn đọc không muốn hiểu backpropagation có thể bỏ qua để đọc tiếp phần Ví dụ với Python.

Phương pháp phổ biến nhất để tối ưu MLP vẫn là Gradient Descent (GD). Để áp dụng GD, chúng ta cần tính được gradient của hàm mất mát theo từng ma trận trọng số $\mathrm{W}^{(l)}$ và vector bias $\mathrm{b}^{(l)}$. Trước hết, chúng ta cần tính *predicted output* $\hat{\mathrm{y}}$ với một input x:

$$
\begin{aligned}
\mathrm{a}^{(0)} &= \mathrm{x} \\
z_i^{(l)} &= \mathrm{w}_i^{(l)T}\mathrm{a}^{(l-1)} + b_i^{(l)} \\
\mathrm{z}^{(l)} &= \mathrm{W}^{(l)T}\mathrm{a}^{(l-1)} + \mathrm{b}^{(l)}, \;\; l = 1, 2, \ldots, L \\
\mathrm{a}^{(l)} &= f(\mathrm{z}^{(l)}), \;\; l = 1, 2, \ldots, L \\
\hat{\mathrm{y}} &= \mathrm{a}^{(L)}
\end{aligned}
$$

Bước này được gói là *feedforward* vì cách tính toán được thực hiện từ đầu đến cuối của network. MLP cũng được gọi

Giả sử $J(\mathrm{W}, \mathrm{b}, \mathrm{X}, \mathrm{Y})$ là một hàm mất mát của bài toán, trong đó W, b là tập hợp tất cả các ma trận trọng số giữa các layers và biases của mỗi layer. X, Y là cặp dữ liệu huấn luyện với mỗi cột tương ứng với một điểm dữ liệu. Để có thể áp dụng các gradient-based methods (mà Gradient Descent là một ví dụ), chúng ta cần tính được:

$$
\frac{\partial J}{\partial \mathrm{W}^{(l)}}; \frac{\partial J}{\partial \mathrm{b}^{(l)}}, \;\; l = 1, 2, \ldots, L
$$

Một ví dụ của hàm mất mát là hàm Mean Square Error (MSE) tức *trung bình của bình phương lỗi*.

$$
\begin{aligned}
J(\mathrm{W}, \mathrm{b}, \mathrm{X}, \mathrm{Y}) &= \frac{1}{N}\sum_{n=1}^{N} ||\mathrm{y}_n - \hat{\mathrm{y}}_n||_2^2 \\
&= \frac{1}{N}\sum_{n=1}^{N} ||\mathrm{y}_n - \mathrm{a}_n^{(L)}||_2^2
\end{aligned}
$$

Với N là số cặp dữ liệu (x, y) trong tập training.

Theo những công thức ở trên, việc tính toán trực tiếp giá trị này là cực kỳ phức tạp vì hàm mất mát không phụ thuộc trực tiếp vào các hệ số. Phương pháp phổ biến nhất được dùng có tên là Backpropagation giúp tính gradient ngược từ layer cuối cùng đến layer đầu tiên. Layer cuối cùng được tính toán trước vì nó *gần gũi* hơn với *predicted outputs* và hàm mất mát. Việc tính toán gradient của các layer trước được thực hiện dựa trên một quy tắc quen thuộc có tên là *chain rule* (https://en.wikipedia.org/wiki/Chain_rule), tức *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 lớp cuối cùng:

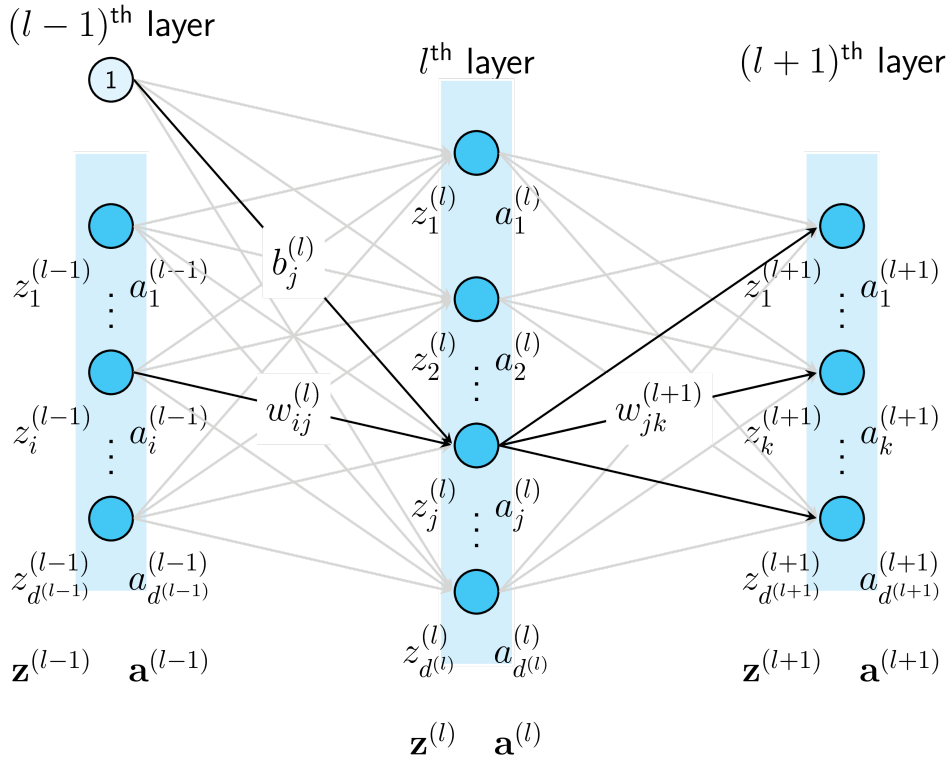
$$\begin{aligned}\frac{\partial J}{\partial w_{ij}^{(L)}} &= \frac{\partial J}{\partial z_j^{(L)}} . \frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} \\ &= e_j^{(L)} a_i^{(L-1)}\end{aligned}$$

Trong đó $e_j^{(L)} = \frac{\partial J}{\partial z_j^{(L)}}$ thường là một đại lượng *dễ 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:

$$\frac{\partial J}{\partial b_j^{(L)}} = \frac{\partial J}{\partial z_j^{(L)}} . \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} = e_j^{(L)}$$

Với đạo hàm theo hệ số ở các lớp l *thấp hơn*, chúng ta hay xem hình dưới đây. Ở đây, tại mỗi unit, tôi đã viết riêng đầu vào z và đầu ra a để các bạn tiện theo dõi.



$$\mathbf{W}^{(l)} \in \mathbb{R}^{d^{(l-1)} \times d^{(l)}}$$

$$\mathbf{b}^{(l)} \in \mathbb{R}^{d^{(l)} \times 1}$$

$$z_j^{(l)} = \mathbf{w}_j^{(l)T}\mathbf{a}^{(l-1)} + b_j^{(l)}$$

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

$$\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)})$$

Hình 6: Mô phỏng cách tính backpropagation. Layer cuối có thể là output layer.

Dựa vào hình trên, ta có thể tính được:

$$\begin{aligned}\frac{\partial J}{\partial w_{ij}^{(l)}} &= \frac{\partial J}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} \\ &= e_j^{(l)} a_i^{(l-1)}\end{aligned}$$

với:

$$\begin{aligned}e_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}} &= \frac{\partial J}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \\ &= \left(\sum_{k=1}^{d^{(l+1)}} \frac{\partial J}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial a_j^{(l)}} \right) f'(z_j^{(l)}) \\ &= \left(\sum_{k=1}^{d^{(l+1)}} e_k^{(l+1)} w_{jk}^{(l+1)} \right) f'(z_j^{(l)}) \\ &= \left(\mathbf{w}_{j:}^{(l+1)} \mathbf{e}^{(l+1)} \right) f'(z_j^{(l)})\end{aligned}$$

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

Dấu sigma tính tổng ở hàng thứ hai trong phép tính trên xuất hiện vì $a_j^{(l)}$ *đóng góp* vào việc tính tất cả các $z_k^{(l+1)}, k = 1, 2, \ldots, d^{(l+1)}$. Biểu thức đạo hàm ngoài dấu ngoặc lớn là vì $a_j^{(l)} = f(z_j^{(l)})$. Tới 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:

$$\frac{\partial J}{\partial b_j^{(l)}} = e_j^{(l)}$$

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:

3.1. Backpropagation cho Stochastic Gradient Descent

3.1.1. Đạo hàm theo từng hệ số $w_{ij}^{(l)}, b_i^{(l)}$

---

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

$$e_j^{(L)} = \frac{\partial J}{\partial z_j^{(L)}}$$

3. Từ đó suy ra:

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

4. Với $l = L-1, L-2, \ldots, 1$, tính:

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

5. Cập nhật đạo hàm cho từng hệ số:

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

3.1.2. Đạo hàm theo ma trận $\mathrm{W}^{(l)}, \mathrm{b}^{(l)}$

Việc tính toán theo từng hệ số như trên chỉ phù hợp cho việc hiểu nguyên lý tính toán, trong khi lập trình, ta cần tìm cách thu gọn chúng về dạng vector và ma trận để tăng tốc độ cho thuật toán. Đặt $\mathrm{e}^{(l)} = [e_1^{(l)}, e_2^{(l)}, \ldots, e_{d^{(l)}}^{(l)}]^T \in \mathbb{R}^{d^{(l)}\times 1}$. Ta sẽ có quy tắc tính như sau:

---

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

$$\mathrm{e}^{(L)} = \frac{\partial J}{\partial \mathrm{z}^{(L)}}$$

3. Từ đó suy ra:

$$\frac{\partial J}{\partial \mathrm{W}^{(L)}} = \mathrm{a}^{(L-1)}\mathrm{e}^{(L)T}$$
$$\frac{\partial J}{\partial \mathrm{b}^{(L)}} = \mathrm{e}^{(L)}$$

4. Với $l = L-1, L-2, \ldots, 1$, tính:

$$\mathrm{e}^{(l)} = \left(\mathrm{W}^{(l+1)}\mathrm{e}^{(l+1)}\right) \odot f'(\mathrm{z}^{(l)})$$

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

$$\frac{\partial J}{\partial \mathrm{W}^{(l)}} = \mathrm{a}^{(l-1)}\mathrm{e}^{(l)T}$$
$$\frac{\partial J}{\partial \mathrm{b}^{(l)}} = \mathrm{e}^{(l)}$$

Chú ý: Biểu thức tính đạo hàm trong dòng trên của bước 3 có thể khiến bạn đặt câu hỏi: tại sao lại là $\mathrm{a}^{(L-1)}\mathrm{e}^{(L)T}$ mà không phải là $\mathrm{a}^{(L-1)T}\mathrm{e}^{(L)}$, $\mathrm{e}^{(L)T}\mathrm{a}^{(L-1)}$, hay $\mathrm{e}^{(L)}\mathrm{a}^{(L-1)T}$? *Quy tắc bỏ túi* cần nhớ là **chiều của hai ma trận ở hai vế phải như nhau**. Thử một chút, vế trái là đạo hàm theo $\mathrm{W}^{(L)}$ là một đại lượng có chiều (*dimension*, not *afternoon*) bằng chiều của ma trận này, tức chiều là $\mathbb{R}^{d^{(L-1)}\times d^{(L)}}$. Vế phải, $\mathrm{e}^{(L)} \in \mathrm{R}^{d^{(L)}\times 1}$, $\mathrm{a}^{(L-1)} \in \mathbb{R}^{d^{(L-1)}\times 1}$. Để hai vế có chiều bằng nhau thì ta phải lấy $\mathrm{a}^{(L-1)}\mathrm{e}^{(L)T}$. Cũng chú ý thêm **rằng đạo hàm theo một ma trận của một hàm số nhận giá trị thực (scalar) sẽ có chiều bằng với chiều của ma trận đó!!**

3.2. Backpropagation cho Batch (mini-batch) Gradient Descent

Nếu chúng ta muốn thực hiện Batch hoặc mini-batch Gradient Descent thì sao? Trong thực tế, mini-batch GD (/2017/01/16/gradientdescent2/#-mini-batch-gradient-descent) được sử dụng nhiều nhất. Nếu lượng dữ liệu là nhỏ, Batch GD (/2017/01/16/gradientdescent2/#-batch-gradient-descent) trực tiếp được sử dụng.

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

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

---

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

$$\mathrm{E}^{(L)} = \frac{\partial J}{\partial \mathrm{Z}^{(L)}}$$

3. Từ đó suy ra:

$$\frac{\partial J}{\partial \mathbf{W}^{(L)}} = \mathbf{A}^{(L-1)}\mathbf{E}^{(L)T}$$

$$\frac{\partial J}{\partial \mathbf{b}^{(L)}} = \sum_{n=1}^{N} \mathbf{e}_n^{(L)}$$

4. Với $l = L-1, L-2, \ldots, 1$, tính:

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

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

$$\frac{\partial J}{\partial \mathbf{W}^{(l)}} = \mathbf{A}^{(l-1)}\mathbf{E}^{(l)T}$$

$$\frac{\partial J}{\partial \mathbf{b}^{(l)}} = \sum_{n=1}^{N} \mathbf{e}_n^{(l)}$$

Mặc dù khi làm thực nghiệm, các công cụ có hỗ trợ việc tự động tính Backpropagation, tôi vẫn không muốn bỏ qua phần này. Hiểu backpropagation rất quan trọng! Xem thêm Yes you should understand backprop (https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b#.g76s9xxzc).

4. Ví dụ trên Python

Source code cho ví dụ này có thể được xem tại đây (https://github.com/tiepvupsu/tiepvupsu.github.io/blob/master/assets/14_mlp/Example%20.ipynb).

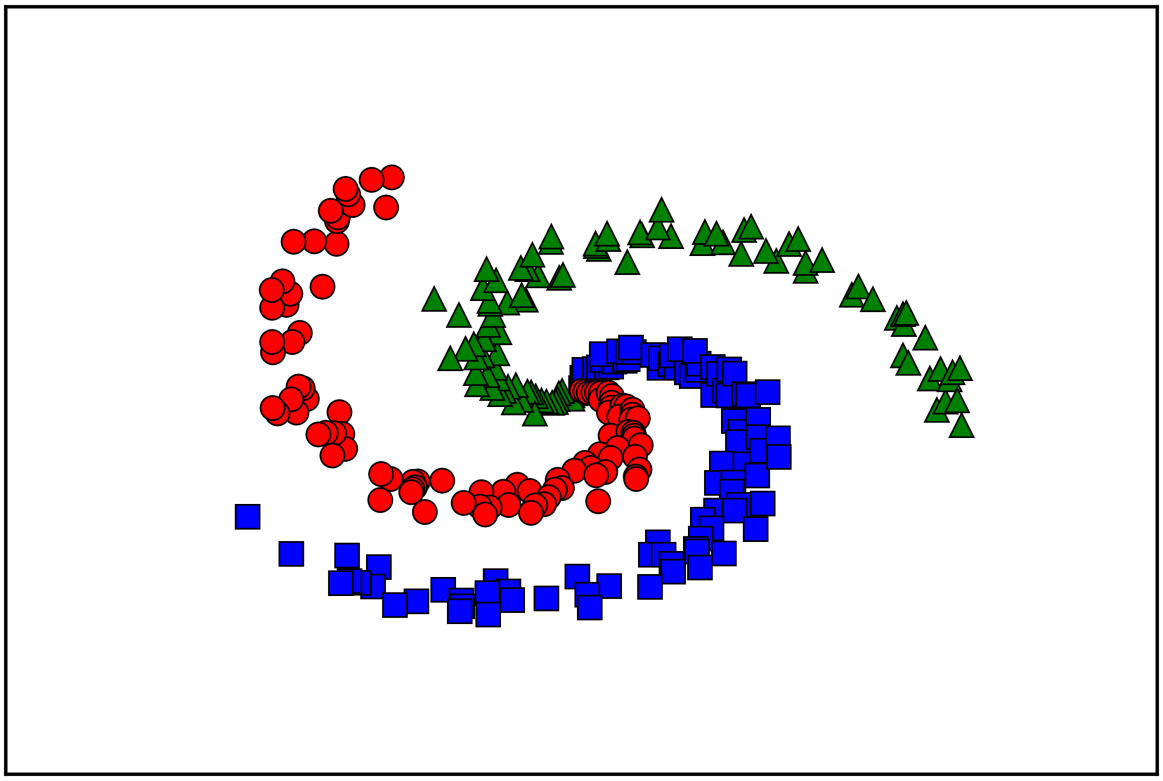
Ví dụ tôi nêu trong mục này mang mục đích giúp các bạn hiểu thực sự cách lập trình cho backpropagation. Khi làm thực nghiệm, chúng ta sử dụng các thư viện sẵn có giúp tính backpropagation. Ví dụ như Sklearn cho MLP (http://scikit-learn.org/stable/modules/neural_networks_supervised.html).

Để kiểm chứng lại những gì tôi viết trên đây có đúng không, chúng ta cùng xem một ví dụ. Ý tưởng trong ví dụ này được lấy từ CS231n Convolutional Neural Networks for Visual Recognition (https://cs231n.github.io/neural-networks-case-study/), phần code dưới đây tôi viết lại cho phù hợp với những tính toán và ký hiệu phía trên.

4.1. Tạo dữ liệu giả

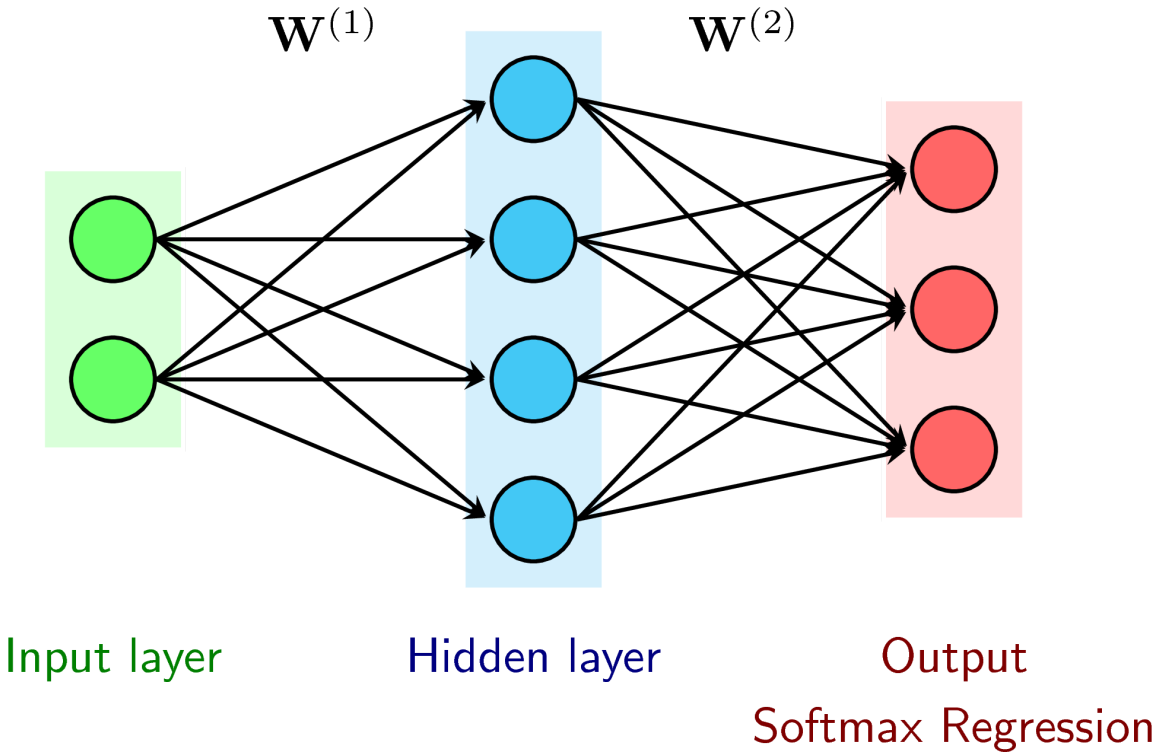
Trước hết, ta tạo dữ liệu cho 3 classes mà không có hai class nào là *linearly separable*:

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

N = 100 # number of points per class
d0 = 2 # dimensionality
C = 3 # number of classes
X = np.zeros((d0, N*C)) # data matrix (each row = single example)
y = np.zeros(N*C, dtype='uint8') # class labels

for j in xrange(C):
  ix = range(N*j,N*(j+1))
  r = np.linspace(0.0,1,N) # radius
  t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
  X[:,ix] = np.c_[r*np.sin(t), r*np.cos(t)].T
  y[ix] = j
# lets visualize the data:
# plt.scatter(X[:N, 0], X[:N, 1], c=y[:N], s=40, cmap=plt.cm.Spectral)

plt.plot(X[0, :N], X[1, :N], 'bs', markersize = 7);
plt.plot(X[0, N:2*N], X[1, N:2*N], 'ro', markersize = 7);
plt.plot(X[0, 2*N:], X[1, 2*N:], 'g^', markersize = 7);
# plt.axis('off')
plt.xlim([-1.5, 1.5])
plt.ylim([-1.5, 1.5])
cur_axes = plt.gca()
cur_axes.axes.get_xaxis().set_ticks([])
cur_axes.axes.get_yaxis().set_ticks([])

plt.savefig('EX.png', bbox_inches='tight', dpi = 600)
plt.show()
```



Hình 7: Phân bố dữ liệu theo class.

Với dữ liệu được phân bố thế này, Softmax Regression không thể thực hiện được vì Bounray giữa các class tạo bởi Softmax Regression có dạng linear (/2017/02/17/softmax/#-boundary-tao-boi-softmax-regression-la-linear). Chúng ta hãy làm một thí nghiệm nhỏ bằng cách thêm một *Hidden layer* vào giữa Input layer vả output layer của Softmax Regression. Activation function của Hidden layer là hàm ReLU: $f(s) = \max(s, 0)$, $f'(s) = 0 \text{ if } s \leq 0$, $f'(s) = 1 \text{ otherwise}$.



Hình 8: 2-layer Neural Networks.

Bây giờ chúng ta sẽ áp dụng Batch Gradient Descent cho bài toán này (vì lượng dữ liệu là nhỏ). Trước hết cần thực tìm công thức tính các activation và output.

4.2. Tính toán Feedforward

$$\begin{aligned}\mathrm{Z}^{(1)} &= \mathrm{W}^{(1)T}\mathrm{X} \\ \mathrm{A}^{(1)} &= \max(\mathrm{Z}^{(1)}, 0) \\ \mathrm{Z}^{(2)} &= \mathrm{W}^{(2)T}\mathrm{A}^{(1)} \\ \hat{\mathrm{Y}} = \mathrm{A}^{(2)} &= \mathrm{softmax}(\mathrm{Z}^{(2)})\end{aligned}$$

Hàm mất mát được tính như sau: (/2017/02/17/softmax/#-ham-mat-mat-cho-softmax-regression)

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

Ở đây, tôi đã cho thêm thừa số $\frac{1}{N}$ để tránh hiện tượng tổng quá lớn với Batch GD. Về mặt toán học, thừa số này không làm thay đổi nghiệm của bài toán.

4.3. Tính toán Backpropagation

Áp dụng quy tắc như đã trình bày ở trên và và đạo hàm theo ma trận trọng số của Softmax Regression (/2017/02/17/softmax/#-toi-uu-ham-mat-mat), ta có:

$$\mathrm{E}^{(2)} = \frac{\partial J}{\partial \mathrm{Z}^{(2)}} = \frac{1}{N}(\hat{\mathrm{Y}} - \mathrm{Y})$$

$$\frac{\partial J}{\partial \mathrm{W}^{(2)}} = \mathrm{A}^{(1)}\mathrm{E}^{(2)T}$$

$$\frac{\partial J}{\partial \mathrm{b}^{(2)}} = \sum_{n=1}^{N} \mathrm{e}_n^{(2)}$$

$$\mathrm{E}^{(1)} = \left(\mathrm{W}^{(2)}\mathrm{E}^{(2)}\right) \odot f'(\mathrm{Z}^{(1)})$$

$$\frac{\partial J}{\partial \mathrm{W}^{(1)}} = \mathrm{A}^{(0)}\mathrm{E}^{(1)T} = \mathrm{X}\mathrm{E}^{(1)T}$$

$$\frac{\partial J}{\partial \mathrm{b}^{(1)}} = \sum_{n=1}^{N} \mathrm{e}_n^{(1)}$$

Từ đó ta có thể bắt đầu lập trình như sau:

4.4. Một số hàm phụ trợ

```
def softmax(V):
    e_V = np.exp(V - np.max(V, axis = 0, keepdims = True))
    Z = e_V / e_V.sum(axis = 0)
    return Z

## One-hot coding
from scipy import sparse
def convert_labels(y, C = 3):
    Y = sparse.coo_matrix((np.ones_like(y),
        (y, np.arange(len(y)))), shape = (C, len(y))).toarray()
    return Y

# cost or loss function
def cost(Y, Yhat):
    return -np.sum(Y*np.log(Yhat))/Y.shape[1]
```

4.4. Phần chương trình chính

```
d0 = 2
d1 = h = 100 # size of hidden layer
d2 = C = 3
# initialize parameters randomly
W1 = 0.01*np.random.randn(d0, d1)
b1 = np.zeros((d1, 1))
W2 = 0.01*np.random.randn(d1, d2)
b2 = np.zeros((d2, 1))

Y = convert_labels(y, C)
N = X.shape[1]
eta = 1 # learning rate
for i in xrange(10000):
    ## Feedforward
    Z1 = np.dot(W1.T, X) + b1
    A1 = np.maximum(Z1, 0)
    Z2 = np.dot(W2.T, A1) + b2
    Yhat = softmax(Z2)

    # print loss after each 1000 iterations
    if i %1000 == 0:
        # compute the loss: average cross-entropy loss
        loss = cost(Y, Yhat)
        print("iter %d, loss: %f" %(i, loss))

    # backpropagation
    E2 = (Yhat - Y )/N
    dW2 = np.dot(A1, E2.T)
    db2 = np.sum(E2, axis = 1, keepdims = True)
    E1 = np.dot(W2, E2)
    E1[Z1 <= 0] = 0 # gradient of ReLU
    dW1 = np.dot(X, E1.T)
    db1 = np.sum(E1, axis = 1, keepdims = True)

    # Gradient Descent update
    W1 += -eta*dW1
    b1 += -eta*db1
    W2 += -eta*dW2
    b2 += -eta*db2
```

```
iter 0, loss: 1.098815
iter 1000, loss: 0.150974
iter 2000, loss: 0.057996
iter 3000, loss: 0.039621
iter 4000, loss: 0.032148
iter 5000, loss: 0.028054
iter 6000, loss: 0.025346
iter 7000, loss: 0.023311
iter 8000, loss: 0.021727
iter 9000, loss: 0.020585
```

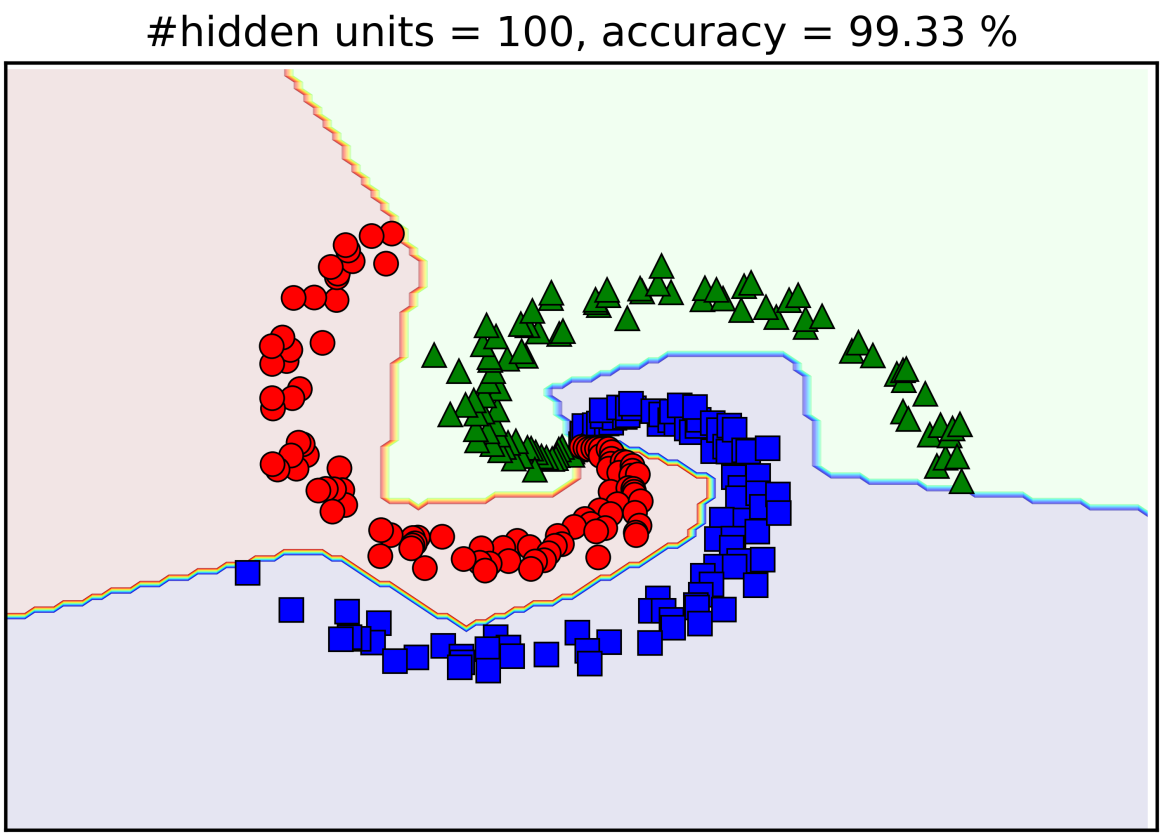
4.5. Kết quả

Như vậy hàm mất mát giảm dần khi số vòng lặp tăng lên. Bây giờ chúng ta cùng áp dụng ngược network này vào phân loại *dữ liệu training*:

```
Z1 = np.dot(W1.T, X) + b1
A1 = np.maximum(Z1, 0)
Z2 = np.dot(W2.T, A1) + b2
predicted_class = np.argmax(Z2, axis=0)
print('training accuracy: %.2f %%' % (100*np.mean(predicted_class == y)))
```

```
training accuracy: 99.33 %
```

Vậy là trong 300 điểm, chỉ có 2 điểm bị phân loại sai! Dưới đây là hình minh hoạ *khu vực* của mỗi class:

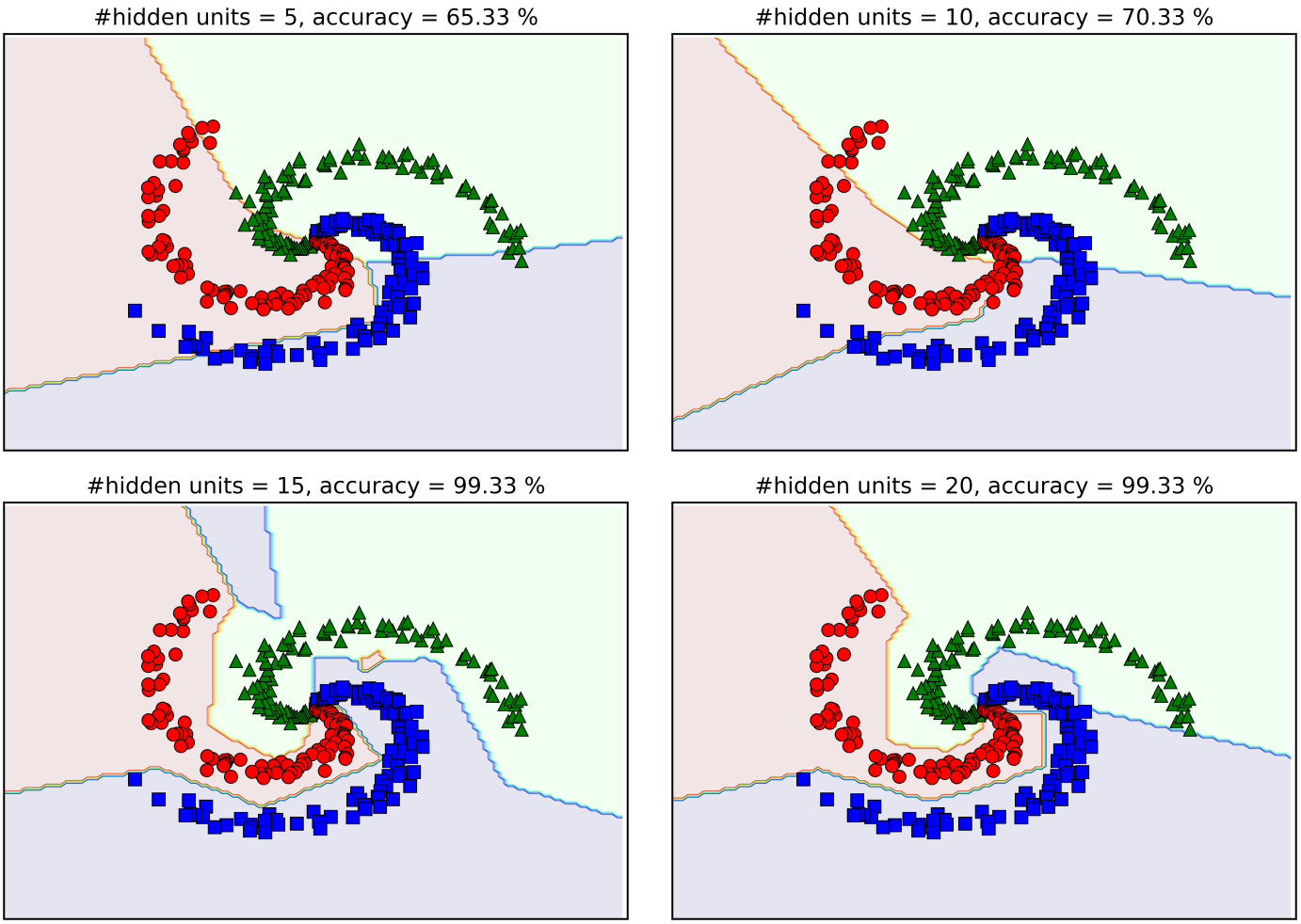


Hình 9: Kết quả khi sử dụng 1 hidden layer với 100 units.

Hai điểm bị phân lớp sai có lẽ nằm gần khu vực trung tâm.

Vậy là chỉ thêm 1 hidden layer, Neural Network đã có thể xây dựng được boundary *phi tuyến*. Kết luận đầu tiên ở đây là khả năng biểu diễn của MLP tốt hơn rất nhiều so với 1-layer Neural Network.

Kết quả bên trên được thực hiện khi số lượng units trong hidden layer là `d1 = 100`. Chúng ta thử thay đổi giá trị này bởi `d1 = 5, 10, 15, 20` xem kết quả khác nhau như thế nào. Dưới đây là hình mình họa:



Hình 10: Kết quả với số lượng units trong hidden layer là khác nhau.

Có một vài nhận xét như sau:

- Khi số lượng hidden units tăng lên, độ chính xác của mô hình tạo được cũng tăng lên.
- Với `d1 = 5`, đường phân định giữa ba classes gần như là đường thẳng.
- Với `d1 = 15`, mặc dù kết quả đã đạt 99.33%, vẫn có một vùng đỏ nhỏ nằm giữa nhánh màu lục và màu lam, và một vùng màu lam khá lớn giữa màu đỏ và lục. Khi một điểm dữ liệu test rơi vào những vùng này, nó sẽ bị phân loại sai.
- Với `d1 = 20`, kết quả nhận được đã tương đối giống với `d1 = 100`. Mặc dù các đường boundary không được trơn tru cho lắm.

5. Thảo luận

- Người ta đã chứng minh được rằng (https://www.dartmouth.edu/~gvc/Cybenko_MCSS.pdf), với một hàm số liên tục bất kỳ $f(x)$ và một số $\varepsilon > 0$, luôn luôn tồn tại một Neural Network với predicted output có dạng $g(x)$ với một hidden layer (với số hidden units đủ lớn và *nonlinear* activation function phù hợp) sao cho với mọi $x, |f(x) - g(x)| < \varepsilon$. Nói một cách khác, Neural Network có khả năng xấp xỉ hầu hết các hàm liên tục.

- Trên thực tế, việc tìm ra số lượng hidden units và *nonlinear* activation function nói trên nhiều khi bất khả thi. Thay vào đó, thực nghiệm chứng minh rằng Neural Networks với nhiều hidden layers kết hợp với các *nonlinear* activation function (đơn giản như ReLU) có khả năng xấp xỉ (khả năng biểu diễn) training data tốt hơn.

- Khi số lượng hidden layers 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 khia 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 training data, nhưng lại không biểu diễn tốt test data. Hiện tượng này gọi là Overfitting (https://en.wikipedia.org/wiki/Overfitting), tôi sẽ trình bày trong bài sau.

- Nếu mọi units của một layer được kết nối với mọi unit của layer tiếp theo (như chúng ta đang xét trong baì này), ta gọi đó là fully connected layer (kết nối hoàn toàn). Neural Networks với toàn fully connected layer ít được sử dụng trong thực tế. Thay vào đó, có nhiều phương pháp giúp làm giảm độ phức tạp của mô hình bằng cách giảm số lượng kết nối bằng cách cho nhiều kết nối bằng 0 (ví dụ, sparse autoencoder (https://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf)), hoặc các hệ số được ràng buộc giống nhau (để giảm số hệ số cần tối ưu) (ví dụ, Convolutional Neural Networks (CNNs / ConvNets) (https://cs231n.github.io/convolutional-networks/)). Bạn đọc muốn tìm hiểu thêm có thể bắt đầu tại đây (http://deeplearning.net).

- Đây là bài cuối cùng trong chuỗi bài về Neural Networks. Viết một bài về Deep Learning sẽ tốn thời gian hơn rất nhiều, trong 1 tuần tôi không đủ khả năng hoàn thành được. Bài tiếp theo tôi sẽ nói về Overfitting (https://en.wikipedia.org/wiki/Overfitting), sau đó chuyển sang một phương pháp classification rất phổ biến khác: Support Vector Machine (https://en.wikipedia.org/wiki/Support_vector_machine).

- Về backpropagation, có rất nhiều điều phải nói nữa. Nếu có thể, tôi xin phép được trình bày sau. Bài này cũng đã đủ dài.

6. Tài liệu tham khảo

[1] Neural Networks Part 1: Setting up the Architecture - Andrej Karpathy (https://cs231n.github.io/neural-networks-1/)

[2] Neural Networks, Case study - Andrej Karpathy (https://cs231n.github.io/neural-networks-case-study/)

[3] Lecture Notes on Sparse Autoencoders - Andrew Ng (https://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf)

[4] Yes you should understand backprop (https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b#.g76s9xxzc)

[5] Backpropagation, Intuitions - Andrej Karpathy (https://cs231n.github.io/optimization-2/)

[6] How the backpropagation algorithm works - Michael Nielsen (http://neuralnetworksanddeeplearning.com/chap2.html)

Nếu có câu hỏi, Bạn có thể để lại comment bên dưới hoặc trên Forum (https://www.facebook.com/groups/257768141347267/) để nhận được câu trả lời sớm hơn.
Bạn đọc có thể ủng hộ blog qua 'Buy me a cofee' (/buymeacoffee/) ở góc trên bên trái của blog.
Tôi vừa hoàn thành cuốn ebook 'Machine Learning cơ bản', bạn có thể đặt sách tại đây (/ebook/). Cảm ơn bạn.

« Bài 13: Softmax Regression (/2017/02/17/softmax/) Bài 15: Overfitting » (/2017/03/04/overfitting/)
