Những Vấn đề Cơ bản về Mạng Neuron

Khanh Nguyen
University of Maryland
College Park, MD 20742
kxnguyen@cs.umd.edu

Hieu Pham
Stanford University
Stanford, CA 94305
hyhieu@cs.stanford.edu

Lời nói đầu

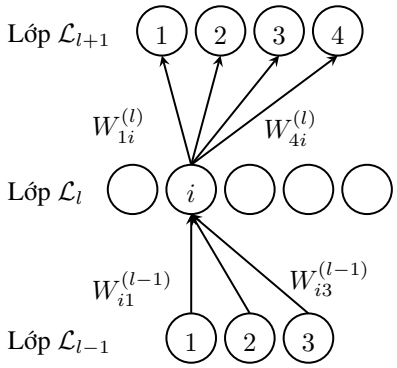
Mạng neuron (neural network) là một mô hình học máy (machine learning) có khả năng mô phỏng bất kỳ hàm số hay quan hệ nào [6]. Tuy không phải là mô hình duy nhất có khả năng này (một ví dụ khác là cây quyết định - decision tree), mạng neuron là mô hình duy nhất hiện nay làm được điều đó với một số lượng tham số vừa phải mà máy tính có khả năng tính toán ra được trong thời gian hợp lý. Tính chất này khiến cho mạng neuron được sử dụng rộng rãi và đạt được nhiều thành quả trong nhiều lĩnh vực khác nhau của trí tuệ nhân tạo (artificial intelligence). Bài viết này nhằm mục đích giới thiệu với các bạn sinh viên và nghiên cứu sinh trong lĩnh vực trí tuệ nhân tạo, đặc biệt là học máy, những kiến thức cơ bản về mạng neuron. Bằng cách sử dụng ngôn ngữ đơn giản nhất có thể và yêu cầu hiểu biết tối thiểu về đại số tuyến tính và giải tích hàm nhiều biến, tác giả hy vọng rằng sau khi đọc xong bài viết, các bạn có thể lập trình được một vài mạng neuron cơ bản để giải quyết một vài bài toán của mình.

1 Giới thiệu

Dù đã ra đời từ khoảng 60 năm trước, thập niên 2006-2015 chứng kiến sự hồi sinh mạnh mẽ của mạng neuron. Hiện này, mô hình này được ứng dụng rộng rãi và đạt được nhiều kết quả tốt trong hầu như mọi lĩnh vực của trí tuệ nhân tạo, bao gồm *thị giác máy tính* (computer vision) [14, 7, 8], *xử lý ngôn ngữ tự nhiên* (natural language processing) [1, 13, 12], *nhận diện giọng nói* (speech recognition) [4, 9].

Tuỳ vào ứng dụng cụ thể, mạng neuron có thể mang các kiến trúc khác nhau, cho phép thông tin giữa các neuron trong mạng được lan truyền theo nhiều phương pháp và định hướng thích hợp. Bài viết này giới thiệu kiến trúc đơn giản nhất của mạng neuron: *mạng kết nối đầy đủ* (fully connected network). Ở kiến trúc này, mạng neuron gồm nhiều *lớp neuron* (neuron layer) được sắp xếp theo thứ tự tuyến tính. Các neuron trong cùng một lớp không được kết nối với nhau. Các neuron giữa hai lớp liên tiếp được kết nối với nhau tạo thành một đồ thị hai phía đầy đủ với các cạnh có trọng số được biểu diễn bởi một ma trận trọng số. Kết cấu này gợi liên tưởng đến mô hình neuron trong bộ não con người với các neuron trong mạng có vai trò như các neuron trong não người, còn các cạnh nối ứng với các đường truyền synapse [1]. Từ đây trở về sau, nếu không nói gì thêm, thuật ngữ "mạng neuron" được hiểu là "mạng neuron kết nối đầy đủ".

[1] Tuy nhiên, vì hoạt động của bộ não người vẫn còn là một ẩn số đối với khoa học, không nên nói rằng hoạt động của mạng neuron mô phỏng hoạt động của bộ não.



Hình 1: Minh hoạ cho kết nối giữa các lớp trong một mạng neuron. Chú ý quy ước về ký hiệu: trọng số giữa neuron i thuộc lớp $\mathcal{L}_{l+1}$ và neuron j thuộc lớp $\mathcal{L}_l$ được ký hiệu là $W_{ij}^{(l)}$.

Có hai con đường lan truyền thông tin trong mạng neuron kết nối đầy đủ. Trong bước *lan truyền tới* (feed-forwarding), thông tin được truyền từ *lớp dữ liệu vào* (input layer), qua các *lớp ẩn* (hidden layer) rồi đến *lớp dữ liệu ra* (output layer). Lớp dữ liệu ra chính là kết quả của mạng, thể hiện giá trị của hàm mà mạng đang mô phỏng tại điểm dữ liệu nhận được ở lớp dữ liệu vào. Tất nhiên, mạng neuron có thể cho kết quả không chính xác, tạo ra các lỗi sai lệch. Trong bước *lan truyền ngược* (back-propagation), các lỗi này sẽ được truyền qua các lớp của mạng theo trình tự ngược lại với bước lan truyền tới, cho phép mạng neuron tính được đạo hàm theo các tham số của nó, từ đó điều chỉnh được các tham số này bằng một thuật toán tối ưu hàm số.

Bài viết này sẽ giúp bạn hiểu về các khái niệm trên, đồng thời đưa bạn đi qua những kiến thức cơ bản để huấn luyện một mạng neuron. Tiếp sau đây, phần 2 của bài viết sẽ giới thiệu về kiến trúc của mạng neuron kết nối đầy đủ. Phần 3 mô tả thuật toán lan truyền tới dùng để suy luận thông tin trên mạng. Phần 4 giới thiệu về hàm kích hoạt. Phần 5 thảo luận cách áp dụng mạng neuron cho các bài toán phân loại. Phần 6 mô tả phương pháp để tìm ra cấu hình tham số tối ưu cho mạng neuron. Phần 7 thảo luận về một vấn đề thường gặp khi sử dụng mạng neuron, overfitting, và các cách khắc phục. Cuối cùng, phần 8 sẽ gợi mở những hướng mới để tìm tòi thêm những kiến thức nâng cao vượt ra ngoài phạm vi của bài viết này.

2 Kiến trúc của mạng neuron kết nối đầy đủ

Như đã nói ở phần giới thiệu, các neuron trong một mạng neuron kết nối đầy đủ được phân chia thành nhiều lớp. Mỗi neuron trong một lớp nhận giá trị trả ra từ các neuron ở lớp liền trước, kết hợp các giá trị này thành một giá trị trung gian, và sau cùng truyền giá trị trung gian qua một *hàm kích hoạt* để trả về kết quả cho neuron ở lớp tiếp theo.

Cụ thể hơn, xét một mạng neuron gồm $L-1$ lớp ẩn. Ta sẽ ký hiệu $\mathcal{L}^{(l)}$ là tập hợp các lớp neuron nằm trong lớp thứ l, với $l = 0, 1, ..., L$. Lớp $\mathcal{L}^{(0)}$ là *lớp dữ liệu vào*. Lớp $\mathcal{L}^{(L)}$ là *lớp dữ liệu ra*. Các lớp còn lại được gọi là các *lớp ẩn*. Neuron trong lớp thứ l chỉ nhận thông tin từ các neuron thuộc lớp thứ $l-1$ và chỉ truyền thông tin cho các neuron thuộc lớp thứ $l+1$. Tất nhiên, các neuron thuộc lớp $\mathcal{L}^{(0)}$ không nhận dữ liệu vào từ các neuron khác và các neuron thuộc lớp $\mathcal{L}^{(L)}$ không truyền dữ liệu ra cho các neuron khác. Hình 1 minh hoạ liên kết xung quanh một neuron mẫu trong một mạng neuron.

Algorithm 1 Thuật toán lan truyền tới.

1: **function** FEEDFORWARD($x^{(0)} \in \mathbb{R}^{|\mathcal{L}_0|}$)
2: **for** $l = 1$ to L **do**
3: $z^{(l)} \leftarrow W^{(l-1)} \cdot x^{(l-1)}$
4: $x^{(l)} \leftarrow f\left(z^{(l)}\right)$
5: **end for**
6: **return** $x^{(L)}$, $Loss(z^{(L)})$
7: **end function**

Giữa hai lớp liên tiếp $\mathcal{L}_l$ và $\mathcal{L}_{l+1}$ trong mạng kết nối đầy đủ, ta thiết lập một ma trận trọng số $W^{(l)}$ với kích thước là $|\mathcal{L}_{l+1}| \times |\mathcal{L}_l|$ [2]. Phần tử $W_{ij}^{(l)}$ của ma trận này thể hiện độ ảnh hưởng của neuron j trong lớp l lên neuron i trong lớp $l+1$. Tập hợp các ma trận trọng số $W = \{W^{(0)}, W^{(1)}, \cdots, W^{(L-1)}\}$ được gọi là tập hợp các tham số của mạng neuron. Việc xác định giá trị của tập tham số được biết đến như việc *học* (learn) hay *huấn luyện* (train) mạng neuron.

3 Phương thức suy luận thông tin của mạng neuron

Giả sử rằng một khi các tham số của một mạng neuron được xác định, làm thế nào để sử dụng mạng neuron này như một hàm số thông thường? Thuật toán *lan truyền tới* (feed-forwarding) (thuật toán 1) cho phép mạng neuron nhận một điểm dữ liệu vào và tính toán điểm dữ liệu ra tương ứng.

Trong thuật toán 1, tham số $x^{(0)}$ là vector dữ liệu vào của mạng. Ở dòng 3, $W^{(l-1)} \cdot x^{(l-1)}$ chỉ phép nhân giữa ma trận $W^{(l-1)}$ với vector $x^{(l-1)}$. Ở dòng 4, hàm $f : \mathbb{R} \to \mathbb{R}$ là một hàm kích hoạt mà ta ta sẽ tìm hiểu ở ngay phần 4. f nhận một số thực và trả về một số thực. Tuy nhiên, trong thuật toán 1, ta sử dụng ký hiệu f trên một vector và được hiểu là áp dụng f lên từng thành phần của vector. Tức là:

$$f\left(\begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{l-1} \end{bmatrix}\right) = \begin{bmatrix} f(v_0) \\ f(v_1) \\ \vdots \\ f(v_{l-1}) \end{bmatrix} \tag{1}$$

Ngoài giá trị của hàm số được mô phỏng, $x^{(L)}$, thuật toán lan truyền tới còn trả về giá trị của hàm mất mát $Loss$, thể hiện độ tốt của tập tham số hiện tại. Hàm này sẽ được nói đến ở phần 6.1.

Lưu ý: các bạn sẽ thấy các tài liệu khác giới thiệu thêm một vector $b^{(l)}$, gọi là *thành kiến* (bias), cho mỗi lớp, và viết dòng 3 của thuật toán 1 như sau:

$$z^{(l)} \leftarrow W^{(l-1)} \cdot x^{(l-1)} + b^{(l-1)} \tag{2}$$

Tuy nhiên, vế phải của phương trình 2 cũng có thể được biến đổi lại thành một phép nhân ma trận duy nhất. Vì thế, tác giả tạm thời bỏ đi vector thành kiến để các công thức trở nên ngắn gọn và dễ hiểu hơn.

4 Hàm kích hoạt

Hàm f trong thuật toán 1 được gọi là *hàm kích hoạt*. Hàm kích hoạt có vai trò vô cùng quan trọng đối với mạng neuron. Trên thực tế, những tiến bộ gần đây nhất trong các nghiên cứu về mạng neuron

[2] Ký hiệu $|S|$ chỉ số phần tử của tập hợp S.

[5, 11] chính là những công thức mới cho f, giúp tăng khả năng mô phỏng của mạng neuron cũng như đơn giản hoá quá trình huấn luyện mạng. Phần này sẽ giải thích vai trò của hàm kích hoạt cũng như giới thiệu một số hàm kích hoạt thường dùng.

Hàm kích hoạt được sử dụng để loại bỏ khả năng tuyến tính hoá của mạng neuron. Để hiểu rõ điều này, ta thử bỏ đi hàm f ở trong thuật toán 1. Chú ý rằng điều này tương đương với việc sử dụng hàm kích hoạt $f(x) = x$. Khi đó, ta nhận thấy kết quả suy luận của mạng neuron chỉ là một ánh xạ tuyến tính của dữ liệu vào. Thật vậy, ta có:

$$x^{(L)} = W^{(L-1)} \cdot x^{(L-1)} = \cdots = \underbrace{W^{(L-1)} \cdots W^{(0)}}_{W} \cdot x^{(0)} = W \cdot x^{(0)} \tag{3}$$

Trong trường hợp này, việc học tất cả L ma trận $W^{(l)}$ là không cần thiết bởi vì tập hợp tất cả các hàm mà mạng biểu diễn được chỉ là các ánh xạ tuyến tính, và có thể được biểu diễn thông qua một phép nhân ma trận duy nhất với:

$$W = W^{(L-1)} \cdot W^{(L-2)} \cdots W^{(0)} \tag{4}$$

Điều này làm suy giảm rất nhiều khả năng mô hình hóa của mạng neuron. Để biểu diễn được nhiều hàm số hơn, ta phải phi tuyến hoá mạng neuron bằng cách đưa kết quả của mỗi phép nhân ma trận-vector $W^{(l-1)} \cdot x^{(l-1)}$ qua một hàm *không* tuyến tính f. Một số hàm kích hoạt thường được sử dụng là:

1. Hàm *sigmoid*: $f(x) = \mathrm{sigm}(x) = \dfrac{1}{1 + \exp(-x)}$;
2. Hàm *tanh*: $f(x) = \tanh(x)$;
3. Hàm *đơn vị tuyến tính đứng* (rectified linear unit – ReLU) ([2]): $f(x) = \max(0, x)$;
4. Hàm *đơn vị tuyến tính đứng có mất mát* (leaky rectified linear unit – leaky ReLU) ([10]): $f(x) = \begin{cases} x & \text{nếu } x > 0 \\ kx & \text{nếu } x \leq 0 \end{cases}$, với k là một hằng số chọn trước. Thông thường $k \approx 0.01$;
5. Hàm *maxout* ([3]): $f(x_1, \cdots, x_n) = \max\limits_{1 \leq i \leq n} x_i$.

5 Mô phỏng hàm xác suất và hàm phân loại

Mạng neuron được ứng dụng rộng rãi để giải các bài toán phân loại, tức là xác định xem dữ liệu vào thuộc loại gì trong một tập các lựa chọn cho trước. Để giải bài toán này, ta dùng mạng neuron để mô phỏng một phân bố xác suất trên tập các lựa chọn. Ví dụ ta muốn dùng mạng neuron để giải bài toán xác nhận gương mặt (face verification). Tập các lựa chọn chỉ gồm hai phần tử: với một cặp ảnh chân dung bất kì, ta yêu cầu mạng neuron trả lời "có" hoặc "không" cho câu hỏi rằng hai bức ảnh đó có phải cùng một người hay không. Mạng neuron đưa ra câu trả lời dựa vào việc tính toán xác suất xảy ra của từng đáp án rồi chọn câu trả lời có xác suất cao hơn. Trong trường hợp này, giả sử rằng tổng xác suất của hai đáp án là 1, vậy thì ta chỉ cần tính xác suất cho một đáp án và suy ra xác suất của đáp án còn lại. Một mạng neuron sử dụng hàm sigmoid kích hoạt ở lớp cuối rất phù hợp để làm điều này, vì hàm sigmoid nhận vào một số thực trong khoảng $(-\infty, +\infty)$ và trả về một số thực trong khoảng $(0, 1)$.

Tổng quát hơn, khi tập phương án lựa chọn có nhiều hơn hai phần tử, ta cần biến mạng neuron thành một phân bố xác suất $P(x)$ thỏa mãn hai điều kiện sau:

1. $P(x) \geq 0 \ \ \forall x \in \Omega$ (Ω là tập lựa chọn);
2. $\sum_x P(x) = 1$.

Xét vector trước khi kích hoạt ở lớp cuối, $z^{(L)} = \left(z_0^{(L)}, z_1^{(L)}, .., z_{|\mathcal{L}_L|-1}^{(L)}\right)$. Thay vì sử dụng hàm sigmoid, ta dùng hàm *softmax* để đưa vector này thành một phân bố xác suất. Hàm softmax có dạng như sau:

$$\text{softmax}(z^{(l)}) = \left(p_0, p_1, \cdots, p_{|\mathcal{L}_L|-1} \right) \tag{5}$$

trong đó

$$p_i = \frac{\exp\left(z_i^{(L)}\right)}{\sum_{j=0}^{|\mathcal{L}_L|-1} \exp\left(z_j^{(L)}\right)}$$

với $\exp(.)$ là hàm lũy thừa theo cơ số tự nhiên e và $0 \leq i \leq |\mathcal{L}_L| - 1$. Lưu ý là số lượng neuron ở lớp cuối, $|\mathcal{L}_L|$, phải bằng với số các phương án lựa chọn.

Dễ thấy là kết quả của hàm softmax thỏa mãn hai điều kiện của một phân bố xác suất và hàm sigmoid là một trường hợp đặc biệt của hàm softmax.

6 Phương pháp ước lượng tham số của mạng neuron

Khi suy luận thông tin trên mạng neuron, ta giả sử rằng các tham số (các ma trận $W^{(l)}$) đều được cho sẵn. Điều này dĩ nhiên là không thực tế; ta cần phải đi tìm các giá trị của tham số sao cho mạng neuron suy luận càng chính xác càng tốt. Như đã nói ở trên, công việc này được gọi là *ước lượng tham số* (parameter estimation), còn được biết đến như quá trình *huấn luyện* (train) hay *học* (learn) của mạng neuron.

Ta gọi $h(x; W)$ và $g(x)$ lần lượt là hàm biểu diễn bởi mạng neuron (với tập tham số W) và hàm mục tiêu cần mô phỏng. Việc tìm ra công thức để tính ngay ra giá trị của tập số tham số rất khó khăn. Ta chọn một cách tiếp cận khác, giảm thiểu dần khoảng cách giữa $h(x; W)$ và $g(x)$ bằng cách lặp lại hai bước sau:

1. Đo độ sai lệch của suy luận của mạng neuron trên một tập điểm dữ liệu mẫu $\{(x_d, g(x_d))\}$, gọi là *tập huấn luyện* (training set).
2. Cập nhật tham số của mạng W để giảm thiểu độ sai lệch trên.

Cách tính độ sai lệch sẽ được đề cập ở phần 6.1. Cách tính giá trị mới của tham số sau mỗi lần cập nhật sẽ được nói đến ở phần 6.2 và 6.3.

6.1 Hàm mất mát

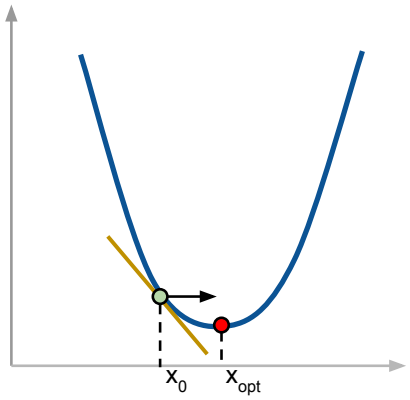
Tổng của các độ sai lệch giữa dữ liệu ra của mạng neuron, $h(x_d; W)$, và dữ liệu ra cần đạt được, $g(x_d)$, thể hiện độ tốt của tập tham số hiện tại. Nếu tập huấn luyện là cố định, tổng này về bản chất là một hàm số chỉ phụ thuộc vào tập tham số W, thường được biết đến với cái tên *hàm mất mát*:

$$Loss(W) \equiv \sum_{d \in D} dist(h(x_d; W), g(x_d)) \tag{6}$$

với D là tập huấn luyện, $dist$ là một hàm tính độ chênh lệch giữa hai điểm dữ liệu ra [3].

Tất nhiên, với một mạng neuron tối ưu, giá trị của hàm mất mát sẽ bằng 0. Trong thực tế, ta muốn tìm ra giá trị của tham số để giá trị hàm mất mát càng nhỏ càng tốt. Vì thế, bài toán ước lượng tham

[3] Có nhiều cách định nghĩa độ chệnh lệch khác nhau. Người ta thường chọn những hàm liên tục, có đạo hàm ở (gần như) mọi nơi, và dễ tính để tính độ chênh lệch.



Hình 2: Ví dụ minh họa cho việc tối ưu một hàm số. Đường thẳng màu vàng là đạo hàm tại điểm x_0. Mũi tên chỉ hướng x_0 cần được dịch chuyển để đến gần hơn với x_{opt}.

số của mạng neuron về bản chất chính là bài toán tìm giá trị của biến W để cực tiểu hóa hàm số $Loss(W)$.

Để hiểu rõ hơn, ta xem xét ví dụ sau đây: sử dụng mạng neuron để mô phỏng hàm XOR của hai bit nhị phân [4]. Ta thiết kế một mạng neuron đơn giản chỉ có hai lớp dữ liệu vào và ra (không có lớp ẩn), và sử dùng hàm sigmoid để kích hoạt. Lớp dữ liệu vào gồm 2 neuron và lớp dữ liệu ra gồm 1 neuron; vì thế, kết nối giữa hai lớp này được thể hiện chỉ bằng một vector tham số $w = (w_1, w_2)$. Vì hàm sigmoid trả về một số thực trong khoảng $(0, 1)$, để mô phỏng hàm XOR, ta cần thực hiện một phép biến đổi đơn giản để chuyển một số thực thành một bit nhị phân: nếu giá trị của lớp cuối cùng lớn hơn 0.5, xuất ra 1, ngược lại xuất ra 0. Đến đây ta được một mạng neuron có định dạng dữ liệu vào và ra giống hệt như hàm XOR.

Việc còn lại là đặt giá trị tham số w thích hợp để mạng neuron của ta hoạt động như một hàm XOR thực thụ. Trong ví dụ này, ta định nghĩa độ chênh lệch giữa dữ liệu ra của mạng neuron p [5] và dữ liệu ra mẫu y bằng bình phương của hiệu giữa chúng:

$$dist(p, y) \equiv \frac{1}{2}(p - y)^2 \tag{7}$$

Hàm mất mát trên tập huấn luyện được định nghĩa như sau:

$$Loss(w) \equiv \frac{1}{2}\sum_{d\in D}(p_d - y_d)^2 = \frac{1}{2}\sum_{d\in D}\left(\text{sigm}(w^\top x_d) - y_d\right)^2 \tag{8}$$

với $D = \{(x_d, y_d)\} = \{((0, 0), 0), ((0, 1), 1), ((1, 0), 1), ((1, 1), 0)\}$.

6.2 Thuật toán gradient descent

Tiếp theo, ta cần một thuật toán để có thể cực tiểu hóa hàm mất mát. Quay lại với những kiến thức về cực tiểu hóa hàm số được dạy ở trung học phổ thông, hẳn ta đã quen thuộc với nhận xét rằng giá

[4] Bit nhị phân là một biến chỉ có thể mang giá trị 0 hoặc 1. Hàm XOR(x, y) nhận vào hai bit nhị phân x và y, trả về giá trị 1 nếu x khác y, và trả về giá trị 0 nếu x bằng y.

[5] p ở đây là một số thực trong khoảng $(0, 1)$ trả ra từ hàm sigmoid. Việc biến đổi số thực này thành bit nhị phân chỉ xảy ra khi ta dùng mạng neuron để suy luận. Ở đây ta đang nói về việc ước lượng tham số.

Algorithm 2 Thuật toán gradient descent.

1: **function** GRADDESC($\nabla_x f$, α)
2: Khởi tạo x_0 tùy ý.
3: **while** $\|\nabla_x f(x_0)\| > \epsilon$ **do**
4: $x_0 \leftarrow x_0 - \alpha \nabla_x f(x_0)$
5: Tùy chọn: cập nhật α.
6: **end while**
7: **return** x_0
8: **end function**

trị đạo hàm của một hàm số bằng 0 tại các điểm cực trị cục bộ. Vậy ở những điểm không phải là cực trị, giá trị này nói với ta điều gì?

Xem xét việc cực tiểu hóa một hàm số $f(x)$ (hình 2). Nhiệm vụ của ta là di từ chuyển giá trị của biến hiện thời x_0 (điểm xanh) đến giá trị của biến tại điểm cực tiểu của f, x_{opt} (điểm đỏ). Dễ thấy rằng nếu $x_0 < x_{opt}$ thì đạo hàm tại x_0 sẽ mang dấu âm. Ngược lại, đạo hàm tại các điểm $x_0 > x_{opt}$ sẽ cho dấu dương. Trong cả hai trường hợp, ta thấy cần đi *ngược lại* với dấu của đạo hàm để đến gần hơn với x_{opt}. Cụ thể hơn, nếu $f'(x_0) < 0$ ta phải tăng x_0, nếu $f'(x_0) > 0$ ta phải giảm x_0.

Khi input x không còn là một số thực mà là một vector l chiều $x = (x_0, x_1, ..., x_{l-1})$, $f(x)$ trở thành một hàm nhiều biến. Đạo hàm riêng (partial derivative) của f theo chiều x_i được kí hiệu là $\frac{\partial f}{\partial x_i}$. *Gradient* của f theo vector x, ký hiệu là $\nabla_x f$, là vector bao gồm đạo hàm riêng theo mỗi chiều của x:

$$\nabla_x f = \left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, ..., \frac{\partial f}{\partial x_{l-1}} \right)$$

Gradient đóng vai trò như dấu của đạo hàm trong trường hợp một chiều. Hướng ngược lại với gradient vector tại một điểm là hướng mà hàm số tăng/giảm nhanh nhất nếu di chuyển một đoạn cực nhỏ ra khỏi điểm đó. Thuật toán *gradient descent* tìm đến điểm cực tiểu bằng cách dịch chuyển tham số theo hướng ngược lại với vector gradient một đoạn nhỏ vừa đủ trong mỗi lần cập nhật.

Thuật toán 2 mô tả các bước của gradient descent. Tại mỗi bước trong vòng lặp *while*, thuật toán này dịch chuyển x_0 đi một lượng phụ thuộc vào gradient của f tại x_0, $\nabla_x f(x_0)$. Dấu trừ thể hiện việc di chuyển ngược lại với hướng của gradient vector. α được dùng để kiểm soát độ lớn của mỗi bước dịch chuyển x_0, được gọi là *độ dịch*. Độ dịch rất quan trọng vì nó ảnh hưởng đến tốc độ hội tụ của x_0 đến x_{opt}. Nếu độ dịch quá lớn, ta có khả năng di chuyển vượt quá khỏi giá trị tối ưu và không bao giờ hội tụ. Ngược lại, nếu di chuyển với độ dịch quá nhỏ, ta phải tốn nhiều bước hơn mới đến được đích, khiến tốc độ hội tụ giảm.

Có một số điểm cần lưu ý khi cài đặt thuật toán 2. Thứ nhất, ở dòng 4, phép tính $x_0 - \alpha \nabla_x f(x_0)$ là một phép trừ giữa hai vector cùng chiều. Thứ hai, phép nhân $\alpha \nabla_x f(x_0)$ được hiểu tùy theo ngữ cảnh vì trong thực tế ta có thể dùng cùng một độ dịch cho tất cả các chiều của $\nabla_x f(x_0)$ (α là số thực), hoặc dùng độ dịch khác nhau cho từng chiều (α là vector). Cuối cùng, điều kiện dừng của thuật toán (dòng 3) là khi ta tìm được một điểm cực trị cục bộ, tức là khi *norm* của gradient vector, kí hiệu là $\|\nabla_x f(x_0)\|$, xấp xỉ 0 với sai số là ϵ [6].

[6] Có nhiều loại norm, thường người ta dùng norm-2, tức là độ dài của vector.

Algorithm 3 Thuật toán lan truyền ngược.

function BACKPROP($x^{(0)} \in \mathbb{R}^{|\mathcal{L}_0|}$, $\{W^{(l)}\}$)
    Áp dụng thuật toán 1 với $x^{(0)}$ để tính các giá trị $z^{(1)}, \cdots, z^{(L)}$, $x^{(1)}, \cdots, x^{(L)}$ và hàm mất mát $Loss(z^{(L)})$.
    $\delta^{(L)} \leftarrow \frac{\partial}{\partial z^{(L)}} Loss$
    for $l = L - 1$ to 0 **do**
        $\frac{\partial}{\partial z^{(l)}} Loss \leftarrow f'(z^{(l)}) \circ \left(W^{(l)\top} \cdot \delta^{(l+1)}\right)$
        $\frac{\partial}{\partial W^{(l)}} Loss \leftarrow \delta^{(l+1)} \cdot x^{(l)\top}$
    end for
    return $\left\{\frac{\partial}{\partial W^{(0)}} Loss, \cdots, \frac{\partial}{\partial W^{(L-1)}} Loss\right\}$
end function

Ta minh họa thuật toán gradient descent với ví dụ mô phỏng hàm XOR ở phần 6.1. Gradient của hàm mất mát theo w sẽ là:

$$\begin{aligned}
\nabla_w Loss(w) &= \nabla_w \frac{1}{2} \sum_{d \in D} (p_d - y_d)^2 \\
&= \nabla_w \frac{1}{2} \sum_{d \in D} \left(\text{sigm}(w^\top x_d) - y_d\right)^2 \\
&= \sum_{d \in D} \left(\text{sigm}(w^\top x_d) - y_d\right) \nabla_w \text{sigm}(w^\top x_d) \\
&= \sum_{d \in D} \left(\text{sigm}(w^\top x_d) - y_d\right) \text{sigm}(w^\top x_d) \left(1 - \text{sigm}(w^\top x_d)\right) x_d \\
&= \sum_{d \in D} (p_d - y_d)\, p_d\, (1 - p_d)\, x_d
\end{aligned} \tag{9}$$

Sau đó ta chỉ việc áp dụng thuật toán 2 với $\nabla_x f(x) \equiv \nabla_w Loss(w)$ để huấn luyện mạng neuron.

6.3 Cách tính đạo hàm trên mạng neuron: thuật toán back-propagation

Khi áp dụng thuật toán gradient descent trên một mạng neuron có nhiều lớp, ta cần tính đạo hàm của hàm mất mát theo tham số của từng lớp neuron một. Việc tính đạo hàm theo từng tham số trở nên phức tạp và khó kiểm soát khi mạng neuron có nhiều lớp hơn và nhiều liên kết phức tạp hơn giữa các neuron. Tuy nhiên, ta có thể tận dụng sự lặp lại cấu trúc giữa các lớp và chỉ cần xem xét cách tính đạo hàm ở một lớp bất kì. Thuật toán *lan truyền ngược* (back-propagation) (thuật toán 3) khái quát hóa cách tính đạo hàm ở một lớp neuron và cách truyền đạo hàm cho lớp liền trước.

Ta sẽ giải thích dòng 5 và dòng 6 của thuật toán 3. Ở dòng 5, chú ý rằng gradient của hàm mất mát theo $z^{(l)}$, $\frac{\partial}{\partial z^{(l)}} Loss$, là một vector có $|\mathcal{L}_l|$ dòng, trong đó dòng thứ i là:

$$\begin{aligned}
\frac{\partial}{\partial z_i^{(l)}} Loss &= \sum_{j=1}^{|\mathcal{L}^{(l+1)}|} \frac{\partial Loss}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}} \\
&= \sum_{j=1}^{|\mathcal{L}^{(l+1)}|} \delta_j^{(l+1)} \frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}} \quad \text{(định nghĩa } \delta_j\text{)} \\
&= \sum_{j=1}^{|\mathcal{L}^{(l+1)}|} \delta_j^{(l+1)} \frac{\partial}{\partial z_i^{(l)}} \sum_{k=1}^{|\mathcal{L}^{(l)}|} W_{jk}^{(l)} f(z_k^{(l)}) \\
&= \sum_{j=1}^{|\mathcal{L}^{(l+1)}|} \delta_j^{(l+1)} W_{ji}^{(l)} f'(z_i^{(l)}) \\
&= f'(z_i^{(l)}) \sum_{j=1}^{|\mathcal{L}^{(l+1)}|} (W^{(l)})_{ij}^\top \delta_j^{(l+1)}
\end{aligned} \tag{10}$$

Ta thấy $\sum_{j=1}^{|\mathcal{L}^{(l+1)}|} (W^{(l)})_{ij}^\top \delta_j^{(l+1)}$ chính là dòng thứ i của vector nhận được từ phép nhân $W^{(l+1)\top} \cdot \delta^{(l+1)}$ [7]. Vì thế, $\frac{\partial}{\partial z^{(l)}} Loss = f'(z^{(l)}) \circ \left(W^{(l)\top} \cdot \delta^{(l+1)}\right)$, trong đó $\circ$ là phép nhân các phần tử có chiều tương ứng của hai vector [8].

Ở dòng 6, gradient theo hàm mất mát theo $W^{(l)}$, $\frac{\partial}{\partial W^{(l)}} Loss$, là một ma trận cùng kích thước với $W^{(l)}$, tức là $|\mathcal{L}^{(l+1)}| \times |\mathcal{L}^{(l)}|$. Phần tử thuộc dòng i và cột j của ma trận này là:

$$\begin{aligned}
\frac{\partial}{\partial W_{ij}^{(l)}} Loss &= \sum_{k=1}^{|\mathcal{L}^{(l+1)}|} \frac{\partial Loss}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial W_{ij}^{(l)}} \\
&= \sum_{k=1}^{|\mathcal{L}^{(l+1)}|} \delta_k^{(l+1)} \frac{\partial}{\partial W_{ij}^{(l)}} \sum_{t=1}^{|\mathcal{L}^{(l)}|} W_{kt}^{(l)} x_t^{(l)} \\
&= \delta_i^{(l+1)} \frac{\partial}{\partial W_{ij}^{(l)}} W_{ij}^{(l)} x_j^{(l)} \quad \left(\frac{\partial}{\partial W_{ij}^{(l)}} W_{kt}^{(l)} = 0 \quad \text{chỉ khi } k \neq i \text{ hoặc } t \neq j \right) \\
&= \delta_i^{(l+1)} x_j^{(l)}
\end{aligned} \tag{11}$$

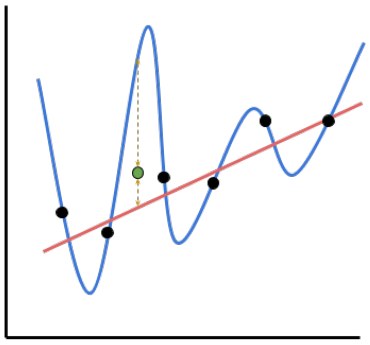
Vì thế, $\frac{\partial}{\partial W^{(l)}} Loss = \delta^{(l+1)} \cdot x^{(l)\top}$.

7 Overfitting

Trong ứng dụng thực tế, ta thường sử dụng mạng neuron để mô phỏng những hàm số hàm số mà cấu trúc của chúng vẫn chưa được xác định. Khi đó, ta chỉ có thể thu nhập được các bộ mẫu dữ liệu ra (vào) được sinh ra từ hàm số, nhưng lại không thể đặc tả quá trình sinh ra các bộ mẫu đó. Một ví dụ kinh điển đó là quá trình bộ não con người thu nhận thông tin từ hình ảnh của chữ viết tay, rồi suy luận ra chữ viết. Cơ chế bộ não biểu diễn hình ảnh và suy luận ra thông tin từ đó là một ẩn số đối với khoa học. Tuy nhiên, ta có thể dùng các bức ảnh cùng với nhãn đúng của chúng để huấn luyện mạng neuron mô phỏng xấp xỉ được quá trình xử lý hình ảnh của bộ não. Cho dù cấu trúc giữa bộ não và

[7] $W^{(l+1)\top}$ là chuyển vị của ma trận $W^{(l+1)}$.

[8] Ví dụ: (1, 2, 3) ∘ (4, 5, 6) = (1 × 2, 2 × 4, 3 × 6) = (2, 8, 18).



Hình 3: Một ví dụ về overfitting. Đa thức có bậc cao hơn (xanh dương) vì quá chú trọng vào việc phải đi qua tất cả các điểm trong tập huấn luyện (đen) nên có hình dạng phức tạp, không "bình thường". Đa thức bậc thấp hơn (đỏ) cho giá trị hàm mất mát cao hơn trên tập huấn luyện nhưng lại phù hợp hơn với phân bố dữ liệu trong thực tế. Điều này thể hiện bằng việc đa thức bậc thấp ước lượng một điểm không có trong tập huấn luyện (xanh) chính xác hơn đa thức bậc cao.

mạng neuron (có thể) khác nhau, với một thuật toán huấn luyện tốt, chúng sẽ đưa ra kết luận giống nhau với cùng một điểm dữ liệu vào.

Đối với bài toán dự đoán, vì mục tiêu cuối cùng của ta là mô phỏng một hàm số ẩn, ta không nên cực tiểu hóa hàm mất mát trên tập huấn luyện. Nếu ta làm như vậy sẽ dẫn đến hiện tượng *overfitting*, tức là mạng neuron sẽ học được một hàm phức tạp để mô phỏng hoàn hảo nhất tập huấn luyện. Tuy nhiên, cũng do cấu trúc phức tạp, hàm này không có tính tổng quát hóa cao, tức là nó rất dễ sai khi gặp một điểm dữ liệu *không có* trong tập huấn luyện (hình 3). Khi ấy, mạng neuron giống như một con người chỉ biết học tủ mà không biết cách vận dụng kiến thức để giải quyết những thứ chưa từng gặp phải. Overfitting là một vấn đề nghiêm trọng đối với mạng neuron vì khả năng mô hình hóa của chúng quá cao, dễ dàng học được các hàm phức tạp. Ta sẽ tìm hiểu một số phương pháp thông dụng để chẩn đoán và ngăn ngừa overfitting cho mạng neuron.

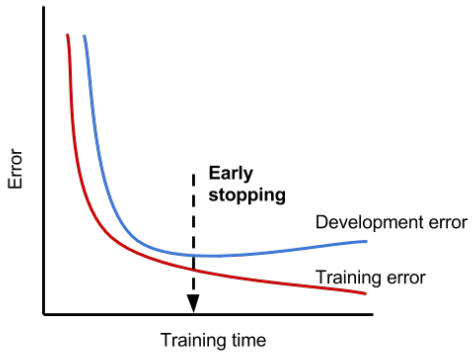
7.1 Sử dụng tập kiểm tra chưa được nhìn thấy

Để đánh giá độ sai lệch của mạng neuron một cách thực tế, ta không thể dùng giá trị của hàm mất mát trên tập huấn luyện bởi vì giá trị này thường sẽ lạc quan (thấp) hơn độ sai lệch thực tế. Muốn khách quan, ta thường sử dụng một *tập kiểm tra* (test set), độc lập với tập huấn luyện, để tính độ sai lệch trên đó. Tập kiểm tra này không được mạng neuron biết đến trong suốt quá trình huấn luyện, và chỉ được sử dụng một khi tham số của mạng neuron đã cố định.

7.2 Dừng huấn luyện sớm (early stopping)

Ta đã biết rằng việc cực tiểu hóa hàm mất mát trên tập huấn luyện có thể gây ra overfitting nếu tập huấn luyện quá nhỏ. Một cách đơn giản để tránh điều này đó là dừng huấn luyện trước khi giá trị hàm mất mát trên tập huấn luyện đạt đến một điểm cực tiểu cục bộ. Tối ưu nhất là nên dừng khi độ sai lệch trên tập kiểm tra bắt đầu tăng lên, cho dù độ sai lệch ở tập huấn luyện vẫn đang giảm (hình 4). Đây chính là lúc overfitting bắt đầu xuất hiện.

Tuy nhiên, như đã nói đến ở phần trước, về nguyên tắc, tập kiểm tra không được can thiệp vào quá trình huấn luyện mạng neuron. Vì thế, ta cần một tập kiểm tra khác để thực hiện dừng sớm, gọi là *tập phát triển* (development set). Khi đó quá trình huấn luyện của ta sẽ diễn ra như sau:



Hình 4: Dừng sớm ngay khi việc huấn luyện làm giảm độ chính xác của mạng neuron trên tập phát triển.

1. Theo dõi độ sai lệch trên tập huấn luyện và tập phát triển.
2. Khi giá trị của độ sai lệch trên tập phát triển bắt đầu tăng, dừng huấn luyện.
3. Cố định mạng neuron với tập tham số mà cho độ sai lệch trên tập phát triển tốt nhất cho đến trước thời điểm dừng. Dùng tập kiểm tra để đánh giá mạng neuron này một lần cuối và thông báo kết quả.

7.3 Bình thường hóa tham số (regularization)

Một cách khác để ngăn ngừa overfitting đó là giới hạn không gian hàm số mà mạng neuron có thể biểu diễn. Điều này được tiến hành bằng việc thu hẹp lại biên độ giá trị của tập tham số. Cụ thể hơn, thay vì huấn luyện mạng chỉ bằng hàm mất mát, ta thêm vào một đại lượng thể hiện "độ lớn" của tập tham số:

$$Loss(w) + \lambda \|w\| \tag{12}$$

với λ là một hằng số chọn trước, $\|w\|$ được gọi là *norm* của vector w.

Khi ta cực tiểu hóa hàm số 12, cả $Loss(w)$ và $\|w\|$ đều sẽ được cực tiểu hóa. Kết quả là, mạng neuron nhận được sẽ có các tham số có giá trị nhỏ hơn bình thường, ngăn ngừa nó trở nên quá linh động và phức tạp.

Trong thực hành, ta thường sử dụng bình phương norm-2, tức là bình phương độ dài của vector [9]:

$$\|w\|_2^2 = \sum_i w_i^2 \tag{13}$$

Bên cạnh đó, huấn luyện với norm-1 lại cho ta tập tham số thưa (có nhiều phần tử là số 0) giúp tăng tốc độ tính toán và giảm bộ nhớ, nhưng norm-1 lại không có đạo hàm liên tục nên việc tối ưu hóa sẽ khó khăn hơn:

$$\|w\|_1 = \sum_i |w_i| \tag{14}$$

Đối với các ma trận của mạng neuron, ta có thể chuyển chúng thành các vector và áp dụng các công thức norm nói trên như bình thường (vì thứ tự các phần tử không quan trọng).

[9] Sử dụng bình phương thay vì lấy căn bậc hai để dễ tính đạo hàm.

8 Tiếp theo là gì?

Trong bài viết này, ta chỉ mới vừa đi qua những vấn đề cơ bản nhất của mạng neuron. Để giúp bạn đọc áp dụng ngay các kiến thức vừa học, tác giả cung cấp hai chương trình huấn luyện mạng neuron để mô phỏng hàm XOR với hai kiến trúc khác nhau để các bạn tham khảo (`https://gitlab.com/khanhptnk/neuralnet_tutorial/tree/master`. Lưu ý là các chương trình được viết bằng ngôn ngữ Python đơn giản và chỉ có tính chất minh họa. Muốn lập trình ra các mạng neuron để giải các bài toán lớn, tác giả gợi ý các bạn nên tìm hiểu về Theano, Pylearn2, Caffe, Torch, hay TensorFlow. Ngoài ra, vì tài liệu về mạng neuron bằng tiếng Việt còn khá khan hiếm, tác giả cũng hy vọng rằng các bạn sẽ đào sâu tìm hiểu thêm các tài liệu bằng tiếng nước ngoài. Cuốn sách Deep Learning được viết bởi các nhà nghiên cứu đầu ngành là một lựa chọn tuyệt vời.

Xin cảm ơn các bạn đã đọc tài liệu này!

Tài liệu

[1] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[2] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[3] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

[4] A. Y. Hannun, C. Case, J. Casper, B. C. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014.

[5] S. Hochreiter and J. Schmidhuber. Long short-term memory. 1997.

[6] K. Hornik, M. B. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[7] A. Karpathy, A. Joulin, and F. F. F. Li. Deep fragment embeddings for bidirectional image sentence mapping. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 1889–1897. Curran Associates, Inc., 2014.

[8] Q. V. Le, R. Monga, M. Devin, K. Chen, G. S. Corrado, J. Dean, and A. Y. Ng. Building high-level features using large scale unsupervised learning. In *In International Conference on Machine Learning, 2012. 103*.

[9] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[10] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, page 1, 2013.

[11] I. S. Rafal Jozefowicz, Wojciech Zaremba. An empirical exploration of recurrent network architectures. In *International Conference in Machine Learning*. ICML, 2015.

[12] R. Socher, D. Chen, C. D. Manning, and A. Ng. Reasoning with neural tensor networks for knowledge base completion. In C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 926–934. Curran Associates, Inc., 2013.

[13] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the*

2013 Conference on Empirical Methods in Natural Language Processing, pages 1631–1642, Stroudsburg, PA, October 2013. Association for Computational Linguistics.

[14] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR 2015*.