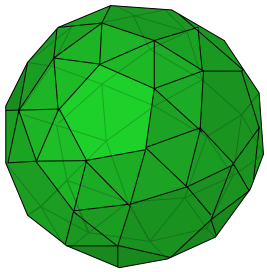
Projects in Mathematics and Applications

NEURAL NETWORK

Ngày 25 tháng 8 năm 2018

Lại Ngọc Tân* †Hà Phương Uyên
Phạm Nguyễn Hoàng Duy‡ §Nguyễn Trung Kiên



*Trường THPT Chuyên Trần Phú - Hải Phòng
†Trường THPT Chuyên Phan Ngọc Hiển - Cà Mau
‡Trường THPT Chuyên Nguyễn Thị Minh Khai - Sóc Trăng
§Trường Phổ thông Năng khiếu, ĐHQG Tp.Hồ Chí Minh

Lời cảm ơn

Lời đầu tiên, xin kính gửi đến Ban tổ chức trại hè PiMA, cũng như những sáng lập viên của trại hè - anh Lê Việt Hải, anh Trần Hoàng Bảo Linh và anh Cấn Trần Thành Trung - lời cảm ơn chân thành nhất vì đã tạo điều kiện hết sức thuận lợi để tất cả các trại sinh đã có được ở PiMA một trải nghiệm vô cùng quý báu. Nó không chỉ dừng lại ở những kiến thức, bài giảng hay, đặc sắc, mà ở đó còn cô đọng những trải nghiệm vô giá, đến từ việc giao lưu, học hỏi giữa trại sinh với nhau từ mọi miền đất nước, đến từ việc sát cánh cùng nhau hoàn thành thật tốt dự án của mình, cũng như sự gắn kết bền chặt giữa những con người cùng chung đam mê, sở thích.

Chúng tôi cũng xin được gửi lời cảm ơn trân trọng đến các anh chị quản lý, người hướng dẫn và các thầy, các cô đến từ trường Đại học Khoa Học Tự Nhiên đã giúp nhóm hoàn thành thật tốt dự án. Đặc biệt, chân thành cảm ơn các anh Nguyễn Trung Nghĩa, anh Phạm Hoàng Nhật, chị Bùi Trần Thục Như, anh Vũ Lê Thế Anh và anh Trần Hoàng Bảo Linh vì đã luôn theo sát nhóm từ những ngày đầu, hướng dẫn tận tâm và miệt mài để nhóm có thể gặt hái được những thành quả tốt nhất.

Hoàn thành năm thứ 3 của trại hè, với những ”cải tiến” về nội dung giảng dạy và số lượng trại sinh, PiMA thật sự đã làm rất tốt, và luôn luôn được chúng tôi tin tưởng sẽ mãi là cầu nối giữa những giấc mơ và kẻ theo đuổi, những con người chung lý tưởng và đam mê, hoặc đơn giản hơn, những người tìm kiếm cho mình một tầm nhìn vào tương lai mới, v.v.

Trại hè khép lại, tương lai và tầm mắt được mở ra. Chúng tôi, những trại sinh PiMA khóa 2018, với thứ hành trang vững chắc đã được trang bị, sẵn sàng đón nhận những thách thức mới trong tương lai.

Tóm tắt nội dung

Một cái nhìn vừa vặn, khá đầy đủ về bản chất, ý nghĩa, cách vận hành của một trong những bước tiến lớn của nhân loại về lĩnh vực toán ứng dụng và khoa học máy tính - Neural Network và thuật toán Gradient Descent. Cùng với đó là hướng phát triển, cải tiến, một số ứng dụng thực tiễn tiêu biểu và cách triển khai, coding rõ ràng cho mọi người cùng kham khảo và tự thực hiện được. Mong rằng bài viết sẽ là lời mời chào nhẹ nhàng, thân mật đến với tương lai mới v.v.

Mục lục

1 Giới thiệu

Artificial Neural Network (ANN) – Mạng neural nhân tạo là gì?

Qua tên gọi, ta có thể biết được rằng cấu trúc của mạng neuron nhân tạo tương tự như một mạng neuron thần kinh của não bộ, gồm nhiều nhóm các **neuron** (các **layer**) được liên kết với nhau và xử lý thông tin bằng cách nhận thông tin đầu vào (**input**) từ các neuron trước và truyền thông tin đã được xử lý (**output**) đến các neuron sau. Mạng neuron hoạt động bằng cách xử lý những dữ liệu đã được cho sẵn gọi là các dữ liệu huấn luyện (**training data**), sau đó tự điều chỉnh lại hoạt động của mình để có thể đưa ra các dự đoán chính xác hơn khi được ứng dụng vào thực tế.

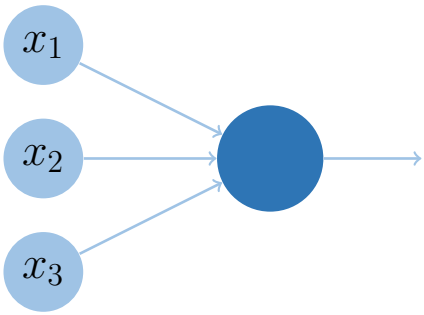
Mạng neuron nhân tạo là một trong những mô hình phổ biến nhất của trí tuệ nhân tạo và máy học và có rất nhiều ứng dụng thực tế, trong đó phổ biến là nhận diện khuôn mặt, giọng nói, chữ số, hay xử lý ngôn ngữ tự nhiên. Công việc nghiên cứu mạng neuron nhân tạo chủ yếu gồm 2 hướng tiếp cận chính, một là nghiên cứu cách hoạt động của mạng neuron sinh học, còn lại là nghiên cứu ứng dụng của ANN vào trí tuệ nhân tạo.

Hiện nay, có rất nhiều loại neural network với cấu trúc, cách hoạt động và chức năng khác nhau. Nhưng trong bản báo cáo này, ta sẽ tìm hiểu một trong những thuật toán đơn giản nhất, và là tiền thân của các neural network hiện đại là mạng perceptron có nhiều lớp - **Multiplayer Perceptron** (MLP).

2 Cấu trúc

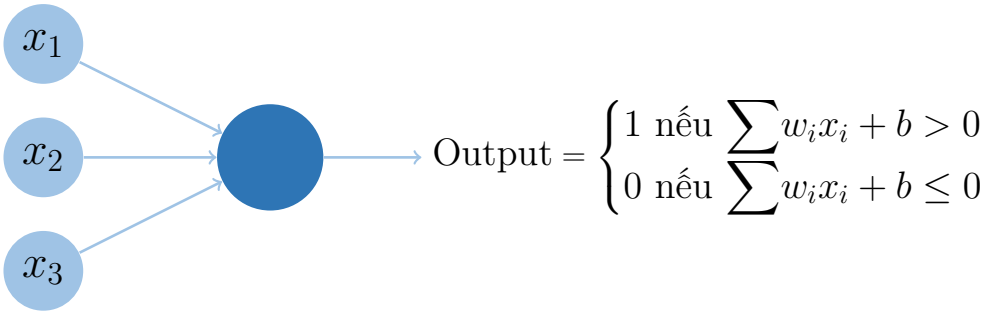
2.1 Perceptron

Perceptron là mô hình neural network đơn giản nhất chỉ gồm một layer. Một perceptron nhận các input nhị phân x_i, thực hiện xử lý tín hiệu với các trọng số w_i (**weight**) và đưa ra một output nhị phân duy nhất dựa trên một ngưỡng (**threshold**) nào đó.



$$\text{Output} = \begin{cases} 1 \text{ nếu } \sum w_i x_i > \text{threshold} \\ 0 \text{ nếu } \sum w_i x_i \leq \text{threshold} \end{cases}$$

Nếu đặt $b = -\text{threshold}$, gọi b là **bias**, ta có sơ đồ đơn giản hơn. Các bias này được xem là các 'ngưỡng' kích hoạt của một perceptron, nếu bias nhỏ thì perceptron đó dễ kích hoat và ngược lại. Chỉ khi một neuron được kích hoạt, output từ neuron đó mới tiếp tục được xử lý và lan truyền trong mạng lưới, nếu không thì neuron đó sẽ không hoạt động. Ta có thể hình dung về perceptron như một thuật toán để đưa ra quyết định bằng cách so sánh (weighing) các input với nhau. Có thể thấy cơ chế này của perceptron khá tương tự với hoạt động của một neuron thần kinh, với các weight thể hiện mức độ liên kết của hai perceptron với nhau.



2.2 Sigmoid neuron

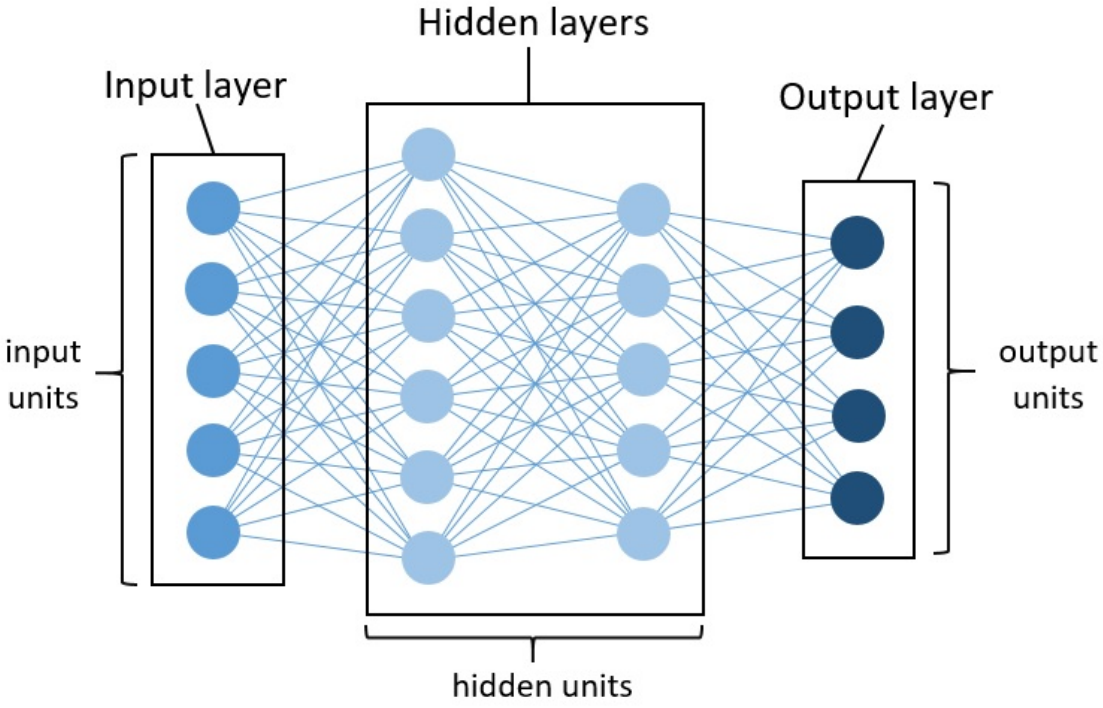
Khi xây dựng một mạng neuron gồm nhiều layer các perceptron, trong quá trình huấn luyện hệ thống bằng cách điều chỉnh các weight và bias cho phù hợp, vì output của perceptron chỉ có thể là 1 hoặc 0, nên một sự thay đổi nhỏ cũng có thể làm thay đổi hoàn toàn output của một perceptron từ 0 thành 1. Như vậy sẽ làm ảnh hưởng khá nhiều đến hoạt động của toàn bộ network, nên ta cần một mô hình khác sao cho khi thực hiện một số điều chỉnh nhỏ, cũng chỉ sẽ thay đổi nhẹ các output của nerwork, như vậy ta sẽ dễ dàng kiểm soát hoạt động của mạng neuron hơn.

Mô hình của một **sigmoid neuron** tương tự như một perceptron, nhưng thay vì chỉ đưa ra output là 0 hoặc 1, output của các sigmoid neuron có thể là bất kỳ số thực nào nằm trong khoảng $(0, 1)$ bằng cách sử dụng hàm activation phi tuyến tính là hàm sigmoid, nên các neuron trong trường hợp này mới có tên gọi là sigmoid neuron.

2.3 Layer

Một mạng lưới neuron nhân tạo bao gồm nhiều lớp các đơn vị neuron. Trong đó, dữ liệu vào được nhập vào layer đầu tiên (input layer), được xử lý qua các layer ở giữa (hidden layer), sau đó hệ thống sẽ xuất kết quả ở layer cuối cùng (output layer). Theo quy ước, input layer được xem là layer 0 và ta bắt đầu đếm từ hidden layer đầu tiên. Trong hình minh họa dưới đây, với 1 input layer, 2 hidden layers và 1 output layer thì network này có tổng cộng 3 layers.

Đơn vị (unit) của mỗi layer là các neuron. Các neuron ở các input layer, hidden layer, output layer được gọi lần lượt là input unit, hidden unit và output unit.



Hình 2.3. Minh họa layer.

2.4 Weight và bias

Trọng số thể hiện sự liên kết giữa 2 neuron trong 2 layer liên tiếp nhau, nếu trọng số nhỏ thì liên kết giữa 2 neuron yếu, ngược lại khi trọng số lớn càng lớn thì liên kết giữa 2 neuron càng mạnh. Gọi $w_{ik}^{(l)}$ là trọng số giữa neuron $n_k^{(l-1)}$ với neuron $n_i^{(l)}$. Vector trọng số $\boldsymbol{w}_i^{(l)}$ là một vector cột biểu hiện các trọng số giữa mỗi neuron trong layer $l-1$ với neuron thứ i trong layer l. Nếu layer $l-1$ có m node, layer l có n node thì:

$$\boldsymbol{w}_i^{(l)} = \begin{bmatrix} w_{1i}^{(l)} & w_{2i}^{(l)} & \dots & w_{mi}^{(l)} \end{bmatrix}^T \in \mathbb{R}^{m\times 1}.$$

Ma trận trọng số $\boldsymbol{w}^{(l)}$ thể hiện các liên kết giữa layer $l-1$ và layer l. Trong một network có L layer thì ta sẽ có L ma trận trọng số như vậy.

$$\boldsymbol{w}^{(l)} = \begin{bmatrix} \boldsymbol{w}_1^{(l)} & \boldsymbol{w}_2^{(l)} & \dots & \boldsymbol{w}_n^{(l)} \end{bmatrix} \in \mathbb{R}^{m\times n}.$$

Như đã nói ở trên, bias của một neuron chính là "ngưỡng" để kích hoạt neuron đó. Bias càng thấp thì neuron đó càng dễ kích hoạt và ngược lại. Tương tự, ta có L ma trận các vector bias $\boldsymbol{b}$, trong đó $\boldsymbol{b}^{(l)}$ là vector cột các bias của các neuron trong layer l.

$$\boldsymbol{b}^{(l)} = \begin{bmatrix} b_1^{(l)} & b_2^{(l)} & \dots & b_n^{(l)} \end{bmatrix}^T \in \mathbb{R}^{n\times 1}.$$

3 Huấn luyện mạng neuron

3.1 Kí hiệu

Ta quy ước các kí hiệu như sau:

- $n_i^{(l)}$: neuron thứ i ở layer thứ l
- $z_i^{(l)}$: input của neuron $n_i^{(l)}$
- $a_i^{(l)}$: output của neuron $n_i^{(l)}$
- $w_{ki}^{(l)}$: trọng số giữa neuron $n_k^{(l-1)}$ và neuron $n_i^{(l)}$
- $b_i^{(l)}$: bias ứng với neuron $n_i^{(l)}$
- $\boldsymbol{z}^{(l)}$: vector input của layer thứ l
- $\boldsymbol{a}^{(l)}$: vector output của layer thứ l
- $\boldsymbol{w}_i^{(l)}$: vector trọng số của neuron $n_i^{(l)}$
- $\boldsymbol{w}^{(l)}$: ma trận trọng số của layer thứ l
- $\boldsymbol{b}^{(l)}$: vector các bias của layer thứ l

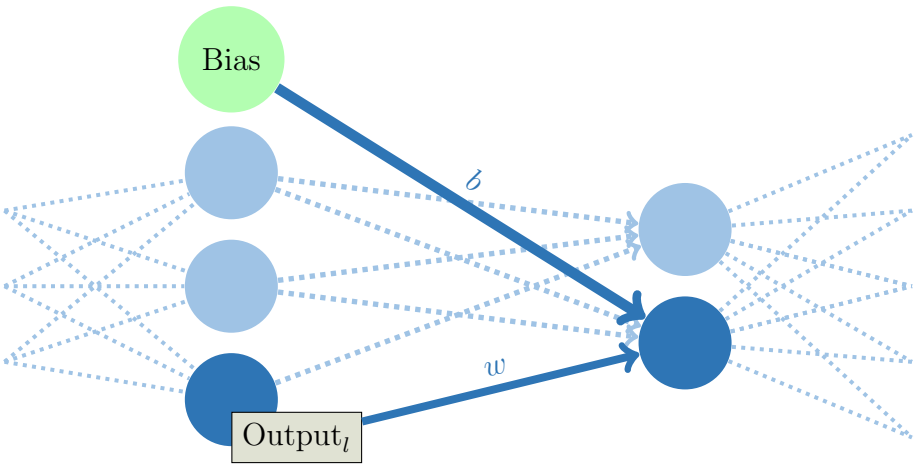
3.2 Feedforward

Neural network bắt đầu hoạt động khi input layer nhận dữ liệu ở các input unit, rồi từ đó thông tin liên tục được xử lý và truyền qua các hidden layer rồi đến output layer, nơi mà network sẽ đưa ra kết luận. Mạng neuron sử dụng cách xử lý này được gọi là feedforward neural network, trong đó feedforward là khái niệm chỉ việc output của layer trước sẽ là input của layer sau qua một hàm activation f nào đó. Input và output (mức activation) của mỗi neuron trong layer l được tính toán dựa trên các output của layer trước như sau:

$$a_i^{(l)} = f(z_i^{(l)}) \text{ với input } z_i^{(l)} = (w_i^{(l)})^T.\boldsymbol{a}^{(l-1)} + b_i^{(l)}.$$

Tổng quát hơn, viết lại dưới dạng vector, ta có:

$$\boldsymbol{a}^{(l)} = f(\boldsymbol{z}^{(l)}) \text{ với } \boldsymbol{z}^{(l)} = (\boldsymbol{w}^{(l)})^T.\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}.$$

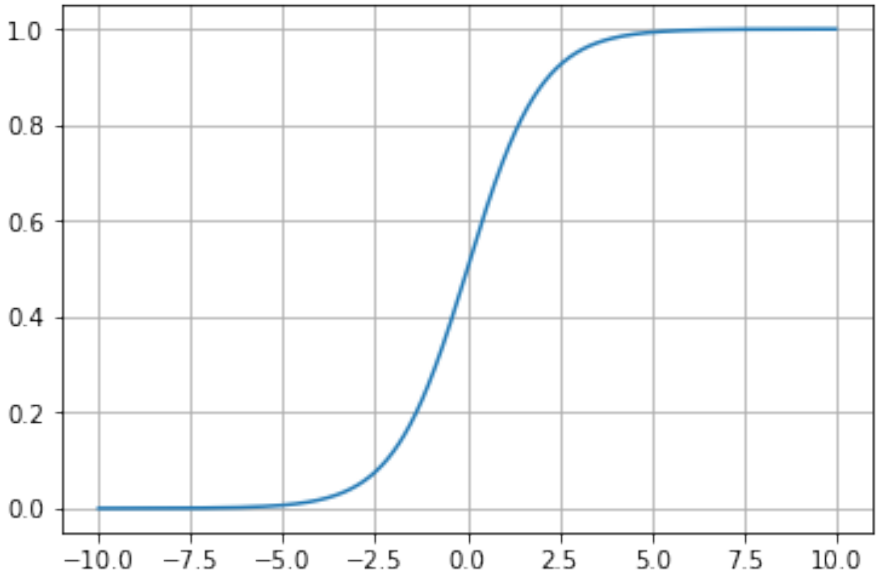


Hình 3.1. Minh họa quá trình feedforward.

3.3 Hàm sigmoid

Hàm sigmoid là một trong những hàm activation thường dùng nhất:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$



Hình 3.2. Dồ thị của hàm Sigmoid.

Hàm sigmoid được sử dụng khá phổ biến nhờ vào đặc điểm đồ thị của nó. Giá trị của hàm biến thiên từ 0 đến 1 khi x biến thiên từ $(-\infty, +\infty)$, cụ thể, khi input càng âm thì output của hàm rất gần với 0, input càng dương thì output của hàm rất gần với 1. Điều này khá tương tự với cơ chế của một perceptron nhưng do có tính liên tục nên nó thể hiện mức kích hoạt của một neuron linh hoạt hơn nhiều so với một perceptron.

Một lý do khác hàm sigmoid dễ lấy đạo hàm nên rất thuận lợi trong quá trình tính toán:

$$\sigma'(x) = \sigma(x).(1 - \sigma(x)).$$

Ngoài ra còn nhiều loại hàm activation khác như tanh, ReLu, v.v.
Trên thực tế, ta còn có thể sử dụng các hàm activation phi tuyến tính khác thay cho hàm sigmoid, ví dụ như hàm tanh.

3.4 Hàm mất mát

Khi huấn luyện, chúng ta cần có một công cụ để đánh giá mức độ sai lệch của kết quả do network đưa ra so với kết quả thực bằng cách sử dụng một hàm mất mát (cost function), ký hiệu $\mathcal{C}$. Có nhiều hàm mất mát được sử dụng như hàm **mean squared error** (MSE) - trung bình bình phương lỗi, cross entropy, v.v. Trong đó hàm MSE được dùng phổ biến nhất vì tính đơn giản của nó.

Đầu tiên ta xét các điểm dữ liệu huấn luyện (x, y). Với mỗi điểm trong tổng số n điểm dữ liệu, lấy bình phương độ sai lệch giữa output $a(x)$ của mỗi neuron trong ouput layer với training ouput y, sau đó cộng tất cả kết quả lại và lấy trung bình của tổng này, ta được giá trị của cost function sau mỗi lần training:

$$\mathcal{C}(w,b) = \frac{1}{2n}\sum_x (y(x) - a(x))^2.$$

Mục tiêu của chúng ta khi xây dựng một mạng neuron là tăng độ chính xác dự đoán của network, tức là giảm độ sai lệch giữa output của network với training output càng nhiều càng tốt. Có thể thấy rằng sự thay đổi của các bộ weight và bias gây ảnh hưởng trực tiếp đến output của network và thay đổi giá trị của hàm mất mát $\mathcal{C}$. Để tìm ra được những bộ weight và bias phù hợp để giảm giá trị của $\mathcal{C}$, ta cần sử dụng thuật toán gradient descent.

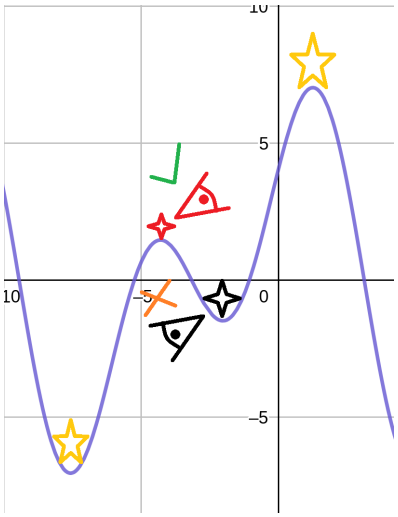
4 Gradient Descent

4.1 Giới thiệu

Như đã nói, huấn luyện một bộ máy, việc chắc chắn xảy ra là nó sẽ sai và ta cần hướng dẫn nó sửa. Dần dần, qua từng "bản cập nhật", sai lầm sẽ bị giảm thiểu và ngày càng đạt đến ngưỡng hoàn hảo. Chính vì lý do đó, việc giảm thiểu độ lớn của hàm mất mát là việc cấp thiết nhất phải làm, kèm với chúng là rất nhiều phương pháp tối ưu, trong số đó, nổi trội và dễ tiếp cận nhất, Gradient Descent ra đời.

4.2 Ý tưởng và cách thực hiện

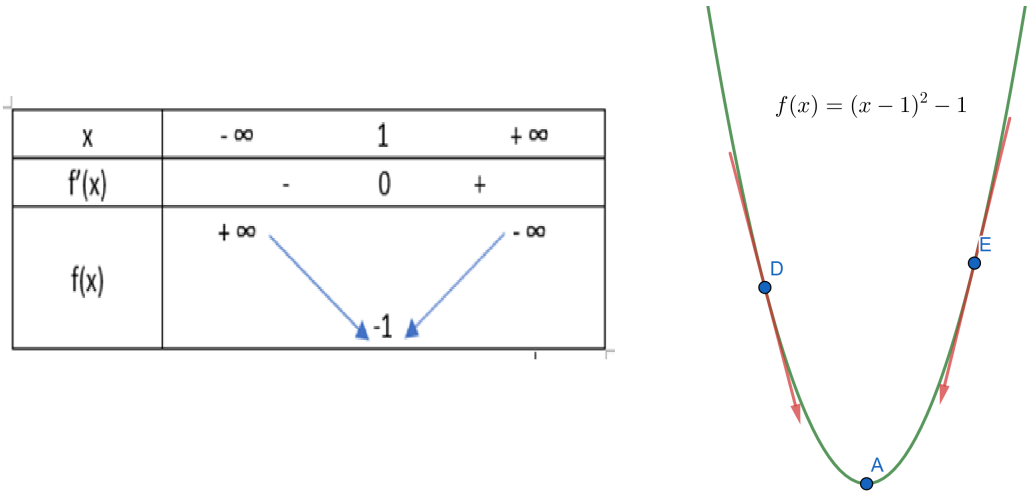
Có bao giờ bạn thử nghĩ, tại sao các thiền sư ở đỉnh núi cao, luôn tìm thấy chính quả và tiêu diệt được mọi thể loại bọn yêu quái? Đó có phải do lũ yêu quái đó quá sức yếu kém nơi bản chất, và cho dù có luyện được thành tinh mạnh đến thế nào cũng chẳng thể đánh bại được công lý? Xin thưa, hoàn toàn không phải thế, mà là do chúng nó đã hết sức bất lợi hơn so với các thiền sư. Hãy ngẫm xem, nếu leo lên một đỉnh núi, theo câu thành ngữ của cha ông có thể dễ dàng "Đứng núi này trông núi nọ" và rút ra kết luận "Núi này cao còn có núi khác cao hơn", đặt trường hợp bạn là một yêu quái, đứng ở local minimum bất kì, bằng cách "tự nhiên" nào bạn có thể biết được liệu đó có phải là global minimum để luyện thành tinh hay không? Và đó là thứ đã ngăn cản việc bọn yêu quái tìm đến được sức mạnh tối thượng của mình và có một cuộc chiến công bằng với liên minh công lý.



Hình 4.1.

Đã có những con yêu tinh "sáng dạ" đã đưa ra những cao kế, đặc sắc nhất trong số chúng chỉ là tìm mọi local minimum nhiều nhất có thể, luyện thành tinh trong hoàn cảnh đó, sau đó so tài (thế vào hàm số) để tìm coi đâu là điểm sâu nhất để luyện được đội quân yêu tinh tối thượng. Thế nhưng, việc này rất mất thời gian, cũng như việc luyện thành công (đạo hàm thành công) cũng không phải dễ. Khó khăn như vậy, Thượng Đế đã ra tay giúp sức, để có được một Infinity War đúng nghĩa, trao cho cái ác thuật toán Gradient Descent để có thể tìm ra Đáy Thế Giới để luyện thành tinh.

Hướng giải quyết: Bắt đầu từ một điểm mà ta coi là "khá" gần với nghiệm của bài toán, sau đó dùng một phép lặp để từ từ tiến dần đến điểm cần tìm, đạo hàm của phương trình ứng với điểm đó bằng 0. Gradient Descent, đúng như tên gọi của nó: "Gradient" (độ dốc) và "Descent" (sự xuống dốc), sẽ thực hiện một cách hiệu quả ý tưởng trên.



Hình 4.2.

Nhìn vào hình vẽ, điều ta cần tìm là tìm một thuật toán để đưa x_k tiến càng gần về x^* càng tốt. Kèm với đó là hai nhận xét quan trọng:

1. Dựa theo bảng biến thiên, nếu đạo hàm của hàm số tại x_k: $f'(x_k) > 0$ thì x_k nằm về bên phải so với x^*. Kết luận này cũng đúng đối với trường hợp ngược lại. Từ quan sát thú vị này, để điểm tiếp theo x_{k+1} gần với x^* hơn, ta cần di chuyển ngược dấu với đạo hàm.

2. Thêm vào đó, x_k càng xa x^* về phía bên phải thì $f'(x_k)$ càng lớn hơn 0 (và ngược lại). Vậy, với việc di chuyển một lượng δ, một cách trực quan, là tỉ lệ thuận với $-f'(x_k)$.

Và thế là, công thức "tối thượng" ra đời, cũng là biểu diễn của áp dụng Gradient Descent cho hàm 1 biến:

$$x_{k+1} = x_k - \eta f'(x_k).$$

Trong đó:

- η, thường là một số dương bé hơn 1, được gọi là learning rate (tốc độ học).
- Dấu trừ thể hiện sự di chuyển ngược dấu với đạo hàm.

4.3 Ứng dụng vào hàm nhiều biến

Điều này cũng được áp dụng tương tự đối với hàm nhiều biến, giả sử ta cần tìm global minimum cho hàm $f(\theta)$ trong đó θ là một vector, dùng để biểu hiện các tham số trong một mô hình cần tối ưu hoá (trong Neural Network thì các tham số chính là hệ số w (weight), b (bias) ta cũng có thuật toán chung:

$$\theta_{k+1} = \theta_t - \eta \nabla_\theta f(\theta_t).$$

Trong đó:

- $\nabla_\theta f(\theta_t)$ (vector gradient) chính là đạo hàm của hàm đó tại một điểm θ bất kì
- Tương tự hàm 1 biến, thuật toán Gradient Descent cho hàm nhiều biến cũng bắt đầu bằng việc khởi tạo giá trị ngẫu nhiên cho biến θ và cập nhật dần theo quy tắc trên.

Áp dụng vào neural network
Ta có thể điều chỉnh và tìm được w_k (weights) và b_k (biases) lý tưởng để tối thiểu hoá hàm mất mát $\mathcal{C}$. Thay thế lần lượt θ bằng các tham số cần tính toán w_m và b_n, cùng với đó, vector gradient $\nabla_\theta f(\theta_t)$ trở thành $\dfrac{\partial \mathcal{C}}{\partial w_m}$ và $\dfrac{\partial \mathcal{C}}{\partial b_n}$:

$$\begin{cases} w_m^{(l+1)} := w_m^{(l)} - \eta \dfrac{\partial \mathcal{C}}{\partial w_m}, \\ b_n^{(l+1)} := b_n^{(l)} - \eta \dfrac{\partial \mathcal{C}}{\partial b_n}. \end{cases}$$

Và cứ lặp lại như vậy, ta sẽ ngày càng tiến tới trị số w_m, b_n mong muốn.

4.4 Các bước cải tiến thuật toán

4.4.1 Mini-batch và Stochastic Gradient Descent

- **Vấn đề:**

  Thuật toán Gradient Descent mà chúng ta sử dụng từ đầu đến giờ đều sử dụng toàn bộ dataset D, tức khi cập nhật $\theta = w$, chúng ta sử dụng tất cả điểm dữ liệu x_i. Trong trường hợp dataset quá lớn, việc này sẽ dẫn đến sự cồng kềnh và phức tạp trong tính toán, cũng như ảnh hưởng của những "điểm nhiễu" (noise) là khá đáng kể

- **Hướng khắc phục:**

  Ta có: $\nabla_\theta f(\theta) = \mathrm{E}_{(x,y)\in D} \nabla_\theta \mathcal{C}(f_\theta(x), y) = \dfrac{1}{|D|} \sum\limits_{(x,y)\in D} \nabla_\theta \mathcal{C}(f_\theta(x), y).$

  Từ đó dẫn đến nhận xét rằng nếu ta lấy một batch D_i từ D rồi tính giá trị trung bình của hàm mất mát trên D, ta sẽ được một ước lượng khác tốt của $\nabla_\theta f(\theta_t)$, nếu các batch D_i có kích cỡ gần bằng nhau. Như vậy, ta sẽ chia toàn bộ dữ liệu thành các mini-batch với độ lớn dữ liệu bằng nhau (thông thường từ 50 đến 100), rồi lần lượt cập nhật cho mỗi batch.

  Khi độ lớn của mỗi batch bằng 1, thuật toán được gọi là Stochastic Gradient Descent.

- **Giải thích:**

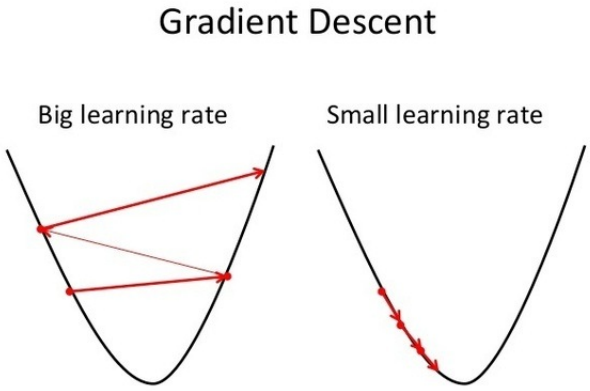
  Nhìn vào một mặt, việc cập nhật từng mini-batch như thế có thể làm giảm tốc độ thực hiện 1 epoch (thường thì mỗi epoch ứng với một lần cập nhật θ). Mặt khác, Mini-batch và Stochastic GD chỉ yêu cầu một lượng epoch nhỏ, sau đó dữ liệu mới thì chỉ cần chạy dưới một epoch là đã có nghiệm tốt, dẫn đến việc giảm thiểu gánh nặng về bộ nhớ.

  Ngoài ra, có thể dễ dàng nhận thấy, batch size càng nhỏ, khả năng tổng quát hoá của mô hình càng tốt, đặc biệt đối với các dataset dồi dào dựa vào khả năng "thích ứng" cũng như kháng noise ưu việt hơn.

4.4.2 Điều chỉnh tốc độ học (Learning Rate Schedule)

- **Vấn đề:**

  Tốc độ học η, một yếu tố hết sức quan trọng góp phần làm nên thành công của thuật toán Gradient Descent nói chung. Dù giá trị chỉ nhỏ thôi (thường chỉ từ 0 đến 1), nhưng do độ "nhạy cảm" cao của thuật toán đối với η là không thể bỏ qua được. Nếu η quá lớn, thuật toán dù tiến rất nhanh tới gần đích, nhưng sẽ không hội tụ được vì bước nhảy quá lớn. Nếu η quá nhỏ, tốc độ hội tụ rất chậm, và khó rời khỏi local minimum.



Hình 4.3.

- **Hướng khắc phục:**

  Tạo ra một hàm $\eta(k)$ để điều chỉnh độ lớn của tốc độ học theo số lần cập nhật đã thực hiện theo tính chất sau: Hàm $\eta(k)$ cần có giá trị lớn ban đầu để thuật toán nhanh chóng tìm ra các khu vực có cực tiểu, và giảm dần để đảm bảo thuật toán hội tụ thành công đến cực tiểu.

- **Một số ví dụ:**

  Hàm tốc độ học mũ (exponential learning rate): $\eta(k) = \eta_0.10^{-\frac{t}{r}}$ với η_0 là tốc độ học ban đầu (khá lớn), và r là hệ số giảm.

  Hàm tốc độ học luỹ thừa (power learning rate): $\eta(k) = \eta_0.\left(1 + \frac{t}{r}\right)^{-c}$.

4.4.3 Điều kiện dừng (Stopping Criteria) và Dừng sớm (Early Stopping)

- **Vấn đề:**

  Đôi khi chưa hẳn là để thuật toán chạy tự do đến cuối cùng đã là tốt nhất. Các vấn đề phát sinh từ noise, tốc độ học chưa tốt, local minimum khó chịu làm hao tổn nhiều resource và thời gian.

- **Hướng giải quyết:**

  Tạo ra các điều kiện dừng để hàm có thể giảm tải những cập nhật thừa thải mà vẫn bảo toàn được kết quả chất lượng. Chú ý rằng việc chọn ra những điều kiện dừng nào thường được rút ra từ tập huấn luyện, do đó dataset sẽ phải bị dàn trải để huấn luyện hai lần: Lần đầu để khảo sát và xác định điều kiện dừng chính xác, lần hai cập nhật bình thường với sự góp mặt của điều kiện dừng.

- **Một số ví dụ:**

  So sánh gradient của nghiệm tại hai lần cập nhật liên tiếp, khi nào giá trị này đủ nhỏ thì dừng lại.

  So sánh giá trị hàm $\mathcal{C}$ qua p lần thử liên tiếp so với gái trị ở các lần thử trước, khi nào giá trị hàm $\mathcal{C}$ không tăng đáng kể thì dừng lại.

  So sánh nghiệm sau một vài lần cập nhật (thường dùng trong Mini-batch và Stochastic Gradient Descent).

5 Backpropagation

Thuật toán gradient descent giúp tối thiểu hóa hàm mất mát $\mathcal{C}$ của network. Để thực hiện được thuật toán gradient descent, trước hết ta cần tính được cụ thể các đạo hàm riêng của hàm mất mát $\mathcal{C}$ là $\dfrac{\partial \mathcal{C}}{\partial w}$ và $\dfrac{\partial \mathcal{C}}{\partial b}$, vì chúng cho ta biết chiều thay đổi của hàm $\mathcal{C}$ khi thay đổi một weight hay bias bất kỳ nào trong hệ thống. Ta sẽ bắt đầu với các weight và bias ở layer output vì đó là layer gần nhất.

Ở layer cuối cùng, ta thử tính $\dfrac{\partial \mathcal{C}}{\partial w_{kj}^{(L)}}$ với $w_{kj}^{(L)}$ là một weight cụ thể nào đó. Áp dụng chain rule, ta có:

$$\frac{\partial \mathcal{C}}{\partial w_{kj}^{(L)}} = \frac{\partial \mathcal{C}}{\partial a_j^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial z_j^{(L)}}{\partial w_{kj}^{(L)}} \tag{1}$$

Có nhiều cách chọn hàm mất mát $\mathcal{C}$ khác nhau (hàm MSE, cross entropy, v.v) nên đạo hàm của $\mathcal{C}$ theo $a_i^{(L)}$ sẽ thay đổi tùy thuộc vào cách ta chọn $\mathcal{C}$; do đó ta sẽ cố định $\mathcal{C}$. Theo cách xây dựng mô hình Neural Network, ta có:

$$\begin{cases} a_j^{(L)} = \sigma\left(z_j^{(L)}\right), \\ z_j^{(L)} = \left(w_j^{(L)}\right)^T \boldsymbol{a}^{(L-1)} + b_j^{(L)}. \end{cases}$$

nên suy ra đơn giản rằng:

$$\begin{cases} \dfrac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \sigma'\left(z_j^{(L)}\right), \\ \dfrac{\partial z_j^{(L)}}{\partial w_{kj}^{(L)}} = a_j^{(L-1)}. \end{cases}$$

Thế vào phương trình (1) ta có:

$$\frac{\partial \mathcal{C}}{\partial w_{kj}^{(L)}} = \frac{\partial \mathcal{C}}{\partial a_j^{(L)}} \cdot \sigma'\left(z_j^{(L)}\right) \cdot a_k^{(L-1)}.$$

Khi đã biết cách tính gradient của $\mathcal{C}$ theo một weight bất kỳ, ta có thể tính được gradient của $\mathcal{C}$ theo các weight trong vector $\boldsymbol{w}_j^{(L)}$:

$$\frac{\partial \mathcal{C}}{\partial \boldsymbol{w}_j^{(L)}} = \frac{\partial \mathcal{C}}{\partial a_j^{(L)}} \cdot \sigma'\left(z_j^{(L)}\right) \cdot \boldsymbol{a}^{(L-1)}.$$

Vậy gradient của $\mathcal{C}$ theo ma trận trọng số ở layer cuối cùng là:

$$\frac{\partial \mathcal{C}}{\partial \boldsymbol{w}^{(L)}} = \frac{\partial \mathcal{C}}{\partial a_j^{(L)}} \cdot \sigma'\left(\boldsymbol{z}^{(L)}\right) \boldsymbol{a}^{(L-1)}.$$

Với ý tưởng tương tự, ta dễ dàng tính được:

$$\frac{\partial \mathcal{C}}{\partial \boldsymbol{b}^{(L)}} = \frac{\partial \mathcal{C}}{\partial a_i^{(L)}} \cdot \sigma'\left(\boldsymbol{z}^{(L)}\right).$$

Để thuận tiện cho việc viết công thức, ta định nghĩa sai số (error) ở neuron $n_i^{(l)}$ là $e_i^{(l)}$ với $e_i^{(l)} = \dfrac{\partial \mathcal{C}}{\partial z_i^{(l)}} = \dfrac{\partial \mathcal{C}}{\partial a_i^{(l)}} \cdot \sigma'(z_i^{(l)})$, trong đó $\boldsymbol{e}^{(l)}$ là vector biểu thị các sai số của các neuron thuộc layer thứ l. Khi đã có một hàm mất mát $\mathcal{C}$ cụ thể, ta dễ dàng tính được $\dfrac{\partial \mathcal{C}}{\partial a_j^{(L)}}$. Viết lại công thức trên dưới dạng ma trận, ta được công thức đầy đủ:

$$\boldsymbol{e}^{(L)} = \nabla_a C \odot \sigma'(\boldsymbol{z}^{(L)}).$$

Viết lại 2 công thức ở trên ta có:

$$\begin{aligned} \frac{\partial \mathcal{C}}{\partial \boldsymbol{w}^{(L)}} &= \boldsymbol{e}^{(L)} \boldsymbol{a}^{(L-1)}, \\ \frac{\partial \mathcal{C}}{\partial \boldsymbol{b}^{(L)}} &= \boldsymbol{e}^{(L)}. \end{aligned}$$

Ta đã có công thức tính gradient của $\mathcal{C}$ theo các weight và bias ở layer cuối cùng. Bây giờ để tính gradient của $\mathcal{C}$ theo bất kỳ weight và bias nào trong network, ta sẽ cần phải tính ngược từ layer cuối cùng, đó cũng là ý tưởng chính của thuật toán backpropagation.

Xét layer $l+1$ có h node. Để tính ngược $e_i^{(l)}$ từ các layer phía sau, ta có:

$$e_i^{(l)} = \frac{\partial \mathcal{C}}{\partial a_i^{(l)}} \cdot \sigma'(z_i^{(l)}) = \left(\sum_{k=1}^{h} \frac{\partial \mathcal{C}}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial a_i^{(l)}} \right) \cdot \sigma'(z_i^{(l)})$$
$$= \left(\sum_{k=1}^{h} e_k^{(l+1)} \cdot w_{ik}^{(l)}) \right) \cdot \sigma'(z_i^{(l)}) = (\boldsymbol{w}_i^{(l+1)T} \boldsymbol{e}^{(l+1)}) \cdot \sigma'(z_i^{(l)}).$$

Vậy để tính được vector $\boldsymbol{e}^{(l)}$ ta có công thức:

$$\boldsymbol{e}^{(l)} = ((\boldsymbol{w}^{(l+1)})^T \boldsymbol{e}^{(l+1)}) \odot \sigma'(\boldsymbol{z}^{(l)}).$$

Có thể tóm tắt thuật toán backpropagation theo các bước cơ bản như sau:

- **Bước 1**

  Tính vector sai số ở layer cuối cùng với một hàm mất mát $\mathcal{C}$ cụ thể:

$$\boldsymbol{e}^{(L)} = \nabla_a C \odot \sigma'(\boldsymbol{z}^{(L)}) \tag{2}$$

- **Bước 2**

  Tìm vector sai số ở các layer còn lại bằng cách tính ngược từ layer cuối cùng:

$$\boldsymbol{e}^{(l)} = ((\boldsymbol{w}^{(l+1)})^T \boldsymbol{e}^{(l+1)}) \odot \sigma'(\boldsymbol{z}^{(l)}) \tag{3}$$

- **Bước 3**

  Tìm đạo hàm của hàm mất mát $\mathcal{C}$ theo từng bias trong network:

$$\frac{\partial \mathcal{C}}{\partial b_i^{(l)}} = e_i^{(l)} \tag{4}$$

- **Bước 4**

  Tính đạo hàm của $\mathcal{C}$ theo từng weight trong network:

$$\frac{\partial \mathcal{C}}{\partial w_{ki}^{(l)}} = a_k^{(l-1)} e_i^{(l)} \tag{5}$$

6 Các vấn đề thường gặp khi huấn luyện mạng neuron

6.1 Overfitting

6.1.1 Overfitting

Tương tự như train error, ta định nghĩa test error là độ lệch giữa kết quả thực tế và kết quả dự đoán khi xử lý một dữ liệu mới, có công thức tính như sau (với y là kết quả thực tế và x là kết quả được dự đoán bởi mô hình) :

$$\mathcal{C}(\boldsymbol{w}, \boldsymbol{b}, \boldsymbol{y}) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - x_i(w, b))^2.$$

Ta coi một hệ thống hoạt động tốt khi giá trị của cả 2 hàm trên đều nhỏ. Nếu train error thấp nhưng test error cao, ta nói mô hình bị overfitting. Nếu train error cao và test error cao, ta nói mô hình bị underfitting. Nếu train error cao nhưng test error thấp có thể do may mắn hoặc tập dữ liệu test quá nhỏ. Tuy nhiên, ở đây ta chỉ đề cập đến một lỗi vô cùng quan trọng là overfitting.

Về cơ bản, "overfitting" là mô hình tạo ra phức tạp quá mức khi tập trung xử lý chính xác những dữ liệu dùng để "training". Điều này dẫn đến mô hình tạo ra chỉ áp dụng được với những dữ liệu đã đưa vào mà chưa tổng quát được những dữ liệu khác. Có thể dẫn đến sai sót lớn khi ta sử dụng hệ thống xử lý các dữ liệu mới. Điều này tương tự như việc học sinh A ôn tập để kiểm tra với 10 bài toán và lời giải. A đã học "vẹt" 10 bài toán nên hoàn thành rất tốt khi giáo viên yêu cầu giải 1 trong 10 bài toán đó, nhưng không làm được với bài toán chưa biết trước.

6.1.2 Validation

Để phát hiện overfitting, ta cần kiểm tra bằng các dữ liệu mới rồi quan sát biểu đồ của tập huấn luyện và tập kiểm tra. Ta sẽ lấy một phần dữ liệu từ tập huấn luyện để thử độ chính xác của hệ thống (gọi là validation data). Bằng validation data, ta có thể dự đoán để điều chỉnh tham số với mong muốn kết quả khi xử lý test data tương tự với validation data. Với ví dụ trên, thay vì để học sinh A học 10 bài, ta để học sinh A học 8 bài, còn 2 bài học sinh A làm dưới sự giám sát xem có hiểu bài hay cần giảng thêm không, cuối cùng mới thực hiện bài kiểm tra.

Cách làm này không chỉ giúp tận dụng dữ liệu mà còn gợi ý một cách giảm overfitting nhanh chóng: dừng huấn luyện ngay tại thời điểm phát hiện overfitting dựa vào quan sát kết quả của tập huấn luyện và tập validation (early stopping).

Tuy nhiên, tập validation khiến quá trình huấn luyện phải sử dụng ít dữ liệu hơn. Do đó, thông thường sẽ huấn luyện 2 lần:

- Lần 1: trước khi bị overfitting, xác định giá trị hàm mất mát đạt được hoặc số lần cập nhật tốt nhất có thể.
- Lần 2: dùng cả tập validation tức dùng toàn bộ dữ liệu huấn luyện đến khi đến khi số lần cập nhật hoặc hàm mất mát có giá trị ở bước trên.

Ngoài ra, qua mỗi lần thử, ta có thể chuyển một số dữ liệu từ tập huấn luyện sang tập validation và ngược lại (cross-validation). Điều đó giúp hệ thống được đánh giá trên nhiều tập validation và tập huấn luyện khác nhau. Độ tốt của mô hình được tính bằng trung bình cộng các lần thử.

6.1.3 L2 regularization

Ta đã biết overfitting xảy ra khi mô hình tạo ra quá phức tạp, không áp dụng được trên dữ liệu mới. Ta có thể điều chỉnh để tạo ra mô hình đơn giản hơn nhưng vẫn xử lý tương đối chính xác dữ liệu đã có. Điều đó có thể khiến giá trị hàm mất mát tăng lên một chút nhưng giúp mô hình tăng tính khái quát với nhiều dữ liệu. Kĩ thuật này gọi chung là regularization.

Ở đây chúng ta đề cập đến một trong số những kĩ thuật được sử dụng thường xuyên là L2 regularization. Nếu $\boldsymbol{w}$ là một vector có n thành phần hay không gian tham số của $\boldsymbol{w}$ là $\mathbb{R}^n$, mỗi thành phần của $\boldsymbol{w}$ có thể tự do thay đổi trong không gian này. Vì vậy, ta cần giới hạn nó để tránh các thành phần của $\boldsymbol{w}$ trở nên quá lớn, điều này cũng đồng thời giúp hạn chế độ phức tạp của mô hình. Nhưng giới hạn $\boldsymbol{w}$ có thể khiến hàm mất mát không đạt được giá trị tốt nhất. Từ đó ta sẽ cân bằng hai yếu tố trên bằng một hàm số sau:

$$\mathcal{C}(\boldsymbol{w}, \boldsymbol{b}) = \mathcal{C}_{\mathcal{D}_{\text{train}}}(\boldsymbol{w}, \boldsymbol{b}) + \lambda \mathcal{R}(\boldsymbol{w}).$$

Trong đó $\mathcal{C}_{\mathcal{D}_{\text{train}}}(\boldsymbol{w}, \boldsymbol{b})$ là hàm mất mát, $\mathcal{R}(\boldsymbol{w})$ là một hàm số phụ thuộc vào $\boldsymbol{w}$ sao cho việc giảm thiểu hàm này tương đương với giới hạn không gian tham số của $\boldsymbol{w}$. Ngoài ra, sự xuất hiện của λ khiến cho vai trò của $\mathcal{C}_{\mathcal{D}_{\text{train}}}(\boldsymbol{w}, \boldsymbol{b})$ và $\mathcal{R}(\boldsymbol{w})$ trở nên bất đối xứng. λ thường là một số dương rất nhỏ để thể hiện việc tối thiểu hàm mất mát được ưu tiên hơn $\mathcal{R}(\boldsymbol{w})$. Ở kĩ thuật L2 regularization, ta sử dụng $\mathcal{R}(\boldsymbol{w}) = \|\boldsymbol{w}\|_2^2$. Hàm số này có vài điểm đặc biệt như sau:

- $\|\boldsymbol{w}\|_2^2$ có đạo hàm rất đơn giản là $2\boldsymbol{w}$. Do đó có thể dễ dàng dùng phương pháp theo gradient descent để cập nhật nghiệm:

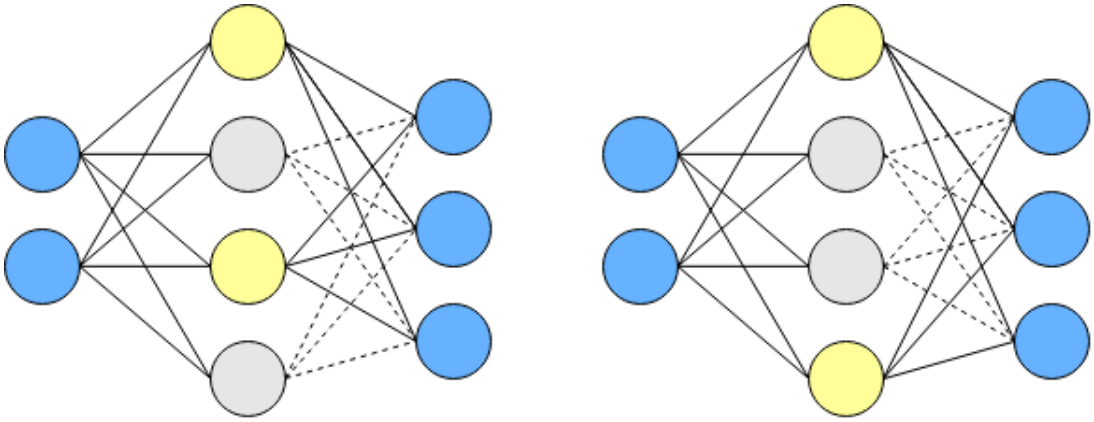
$$\frac{\partial \mathcal{C}}{\partial \boldsymbol{w}} = \frac{\partial \mathcal{C}_{\mathcal{D}_{\text{train}}}}{\partial \boldsymbol{w}} + \lambda \boldsymbol{w}.$$

- Việc tối thiểu $\|\boldsymbol{w}\|_2^2$ giúp đưa $\boldsymbol{w}$ tiến gần đến 0. Điều này có thể đưa hệ số trong ma trận trọng số của mô hình nhỏ hơn, dẫn đến giảm giá trị các hidden units , giúp giảm overfitting.

Trong quá trình huấn luyện, ta vẫn chú ý quan sát kết quả để sử dụng early stopping và thay đổi λ cho phù hợp. Ngoài L2 regulaiation, còn có kĩ thuật L1 regulaiation là việc ta cho $\mathcal{R}(\boldsymbol{w}) = \|\boldsymbol{w}\|_1 = \sum_{i=0}^{n} |\boldsymbol{w_i}|$, khi đó ta tối thiểu $\boldsymbol{w}$ có thể cho ra nhiều thành phần bằng 0, thuận lợi trong tính toán và lưu trữ.

6.1.4 Dropout

Cũng như L1, L2 regularization, dropout cũng là một kĩ thuật của regularization. Tuy nhiên dropout không sửa đổi hàm mất mát, thay vào đó, nó thay đổi chính cấu trúc mạng nơron. Về cơ bản, dropout là việc một số units bất kì theo một xác suất p cho trước bị ẩn đi trong mỗi bước huấn luyện. Điều này giúp mỗi unit tăng khả năng tự điều chỉnh mà không phụ thuộc vào các unit khác, từ đó để giảm sự phức tạp trong mối liên hệ của các layer.



Hình 6.1. Mô hình Dropout ($p = 0.5$)

Dropout hoạt động như sau:

- Dropout được áp dụng cho mỗi layer với xác suất p cho trước. Mỗi layer khác nhau có thể áp dụng xác suất khác nhau nhưng trên một layer chỉ có một dropout.
- Ở mỗi bước trong quá trình huấn luyện, mỗi layer chọn "ngẫu nhiên" theo xác suất p các units bị ẩn, các units này coi như giá trị là 0.
- Trong quá trình kiểm tra, những units có trọng số nối đến nó đã được áp dụng dropout thì sẽ thay đổi giá trị w của trọng số đó: $w := w.p$. Thực ra đây là phép tính giá trị trung bình của trọng số trong tất cả các trường hợp đã dùng dropout huấn luyện.

Cách làm trên có thể coi là ta sẽ tìm những mô hình có tính tổng quát nhất bằng cách thực hiện với nhiều mạng nơron khác nhau. Đôi khi, kết quả mỗi mô hình có thể sai khác nhưng ta sẽ lấy giá trị trung bình để giảm thiểu điều này.

6.2 Vanishing gradient

Ta sử dụng mạng neuron nhiều lớp để tăng độ chính xác của kết quả dự đoán. Tuy nhiên, khi mạng có nhiều lớp hơn cần thiết có thể gây ra khó khăn khi huấn luyện, đặc biệt là hiện tượng vanishing gradient.

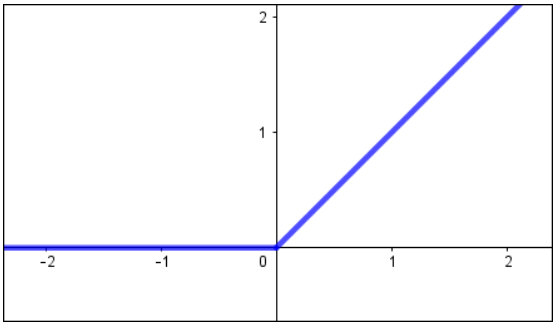
Để hiểu được bài toán vanishing gradient, ta sẽ ví dụ bằng một hàm activation phổ biến là sigmoid function $= f(x) = \dfrac{1}{1+e^{-x}}$. Ta thấy khi các giá trị inputs quá lớn (âm hoặc dương) thì giá trị gần như không đổi, tức đạo hàm xấp xỉ 0. Do đó ở mỗi lần cập nhật trong quá trình backpropagation, weights chỉ thay đổi không đáng kể, điều này khiến hệ thống phải cập nhật rất nhiều lần. Mặt khác, khi sử dụng chain rule để tính gradient của một layer dựa vào gradients của các layer trước đó, bước tính toán càng về layer cuối thì gradient sẽ càng giảm nhanh (do bằng tích của các giá trị nhỏ hơn 1).

Do đó ta tìm cách thay đổi hàm activation, ở đây thay bằng hàm ReLU:

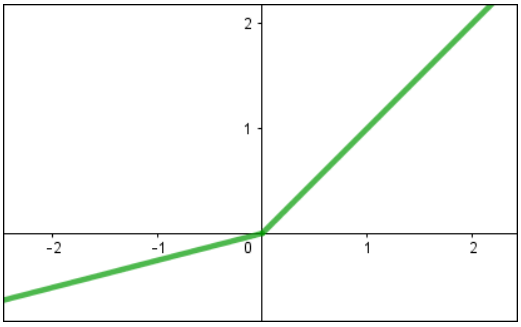
$$\text{ReLU}(z) = f(z) = \max(0, z).$$

Đối với các giá trị $z > 0$, hàm này giúp thực hiện tính toán dễ dàng. Tuy nhiên, khi $z < 0$ ta có $f(z) = 0$, nghĩa là output $= 0$ và gradient của f bằng 0. Do đó, nếu learning rate lớn sẽ có nhiều neural cho ra giá trị là 0 khi training, chúng trở nên mất tác dụng.

Để khắc phục nhược điểm này, ta có thể sử dụng các biến thể của ReLU như Leaky ReLU, RReLU, PReLU,v.v.. Ví dụ: Leaky ReLU$(z) = f(z) = \max(\alpha z, z)$, khi đó α giúp điều chỉnh độ dốc của hàm khi $z < 0$ thay vì bằng 0 như sử dụng hàm ReLU.



Hình 6.2. Hàm ReLU



Hình 6.3. Hàm Leaky ReLU ($\alpha = 0.3$)

Nhiều trường hợp khác, khi gradient trở nên lớn hơn trong quá trình Backpropagation, giá trị cập nhật cho weights trở nên quá lớn khiến chúng khó hội tụ, và điều đó làm hệ thống không đạt được kết quả mong muốn. Hiện tượng này gọi là exploding gradient. Tuy nhiên, điều này rất dễ phát hiện và chỉnh sửa khi ta quan sát các kết quả dự đoán.

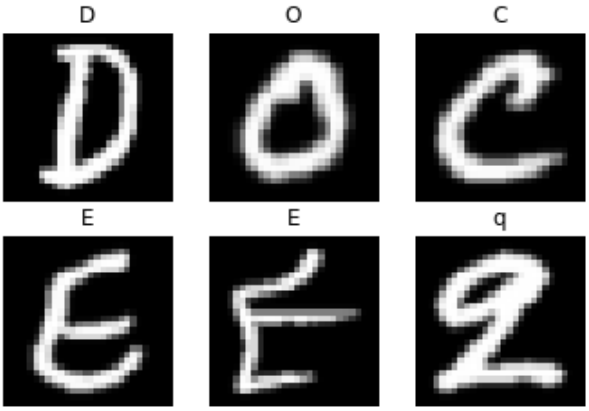
7 Áp dụng của mô hình

7.1 Thực trạng

- Khó khăn trong việc chấm điểm bài thi tự luận của nhiều giáo viên khi chữ viết thí sinh quá xấu (Ví dụ là công tác dịch thuật của các thầy khi giải thích bài thi IMO ở mỗi quốc gia cho hội đồng chấm điểm, khó khăn trong việc chấm điểm môn thi văn của học sinh).
- Tư liệu dạng bản cứng chưa được tận dụng triệt để (tốn khoảng trống đựng tài liệu, muốn tìm kiếm tư liệu dạng cứng rất khó khăn).
- Một vài tư liệu chữ viết rất cần thiết trong việc điều tra, dù đã cắt ra được từ tư liệu dạng ghi hình nhưng vẫn rất khó khai thác triệt để.
- Khó khăn trong việc tra cứu nhanh một cụm từ bất ngờ gặp phải trong ngày.

7.2 Hướng giải quyết - Character recognition:

Với tài liệu dạng hình ảnh, bản scan, giấy viết, hay những thông tin chứa văn bản khác, việc sử dụng machine learning để "học" cách nhận diện chữ viết sẽ dễ dàng hơn viết một giải thuật dùng các quy luật để đánh giá. Sau đây là chương trình của nhóm dùng để nhận diện chữ viết tay, sử dụng mô hình neural network với bộ dữ liệu Extended-MNIST.



Hình 7.1. Dữ liệu mẫu của EMNIST.

EMNIST bao gồm 131800 bức ảnh của 47 loại chữ cái và số. Mỗi bức ảnh có kích thước 28×28. Chúng ta sẽ dùng xây dựng một neural network đơn giản gồm 4 layers dựa trên các thư viện machine learning:

- 1 input layer gồm 784 neuron tương ứng với 28×28 pixel.
- 2 hidden layer, gồm 512 và 256 neuron để tổng quát hoá dữ liệu.
- 1 output layer, gồm 47 neuron ứng với 47 loại chữ cái và số cần phân loại.

```
model = Sequential()

model.add(Dense(512, input_dim = 784, activation = 'sigmoid'))
model.add(Dense(256, activation = 'sigmoid'))
model.add(Dense(47, activation = "softmax"))
```

Hình 7.2. Cấu trúc mạng neuron.

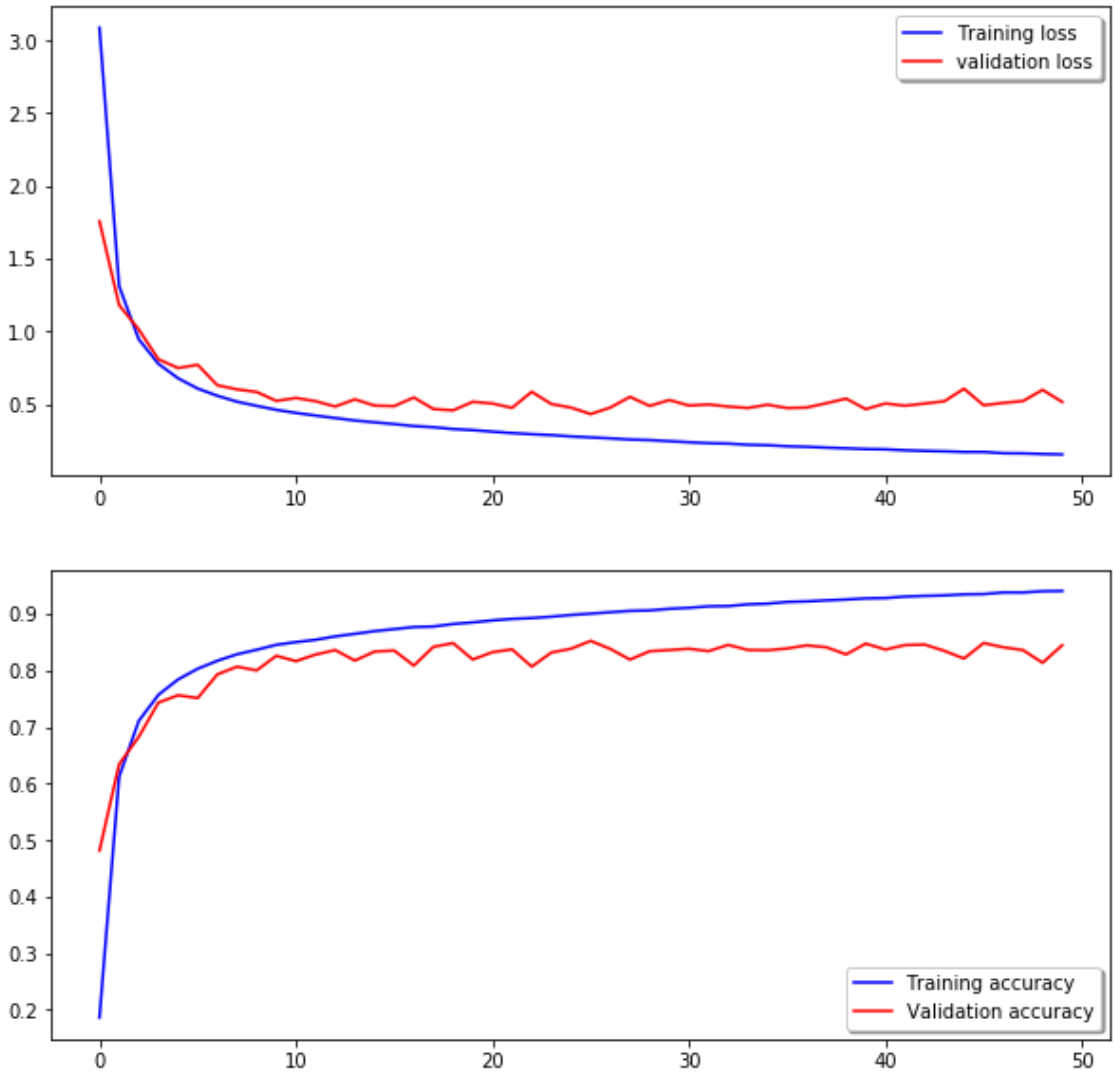
Mô hình sẽ sử dụng hàm mất mát mean squared error (MSE) và tối ưu hoá nó bằng thuật toán stochastic gradient descent (SGD) như đã miêu tả trong bài viết. Trong trường hợp này, ta dùng learning rate = 3.

```
optimizer = SGD(lr = 3.0)

# Compile model
model.compile(optimizer = optimizer , loss = "mean_squared_error", metrics=["accuracy"])
```

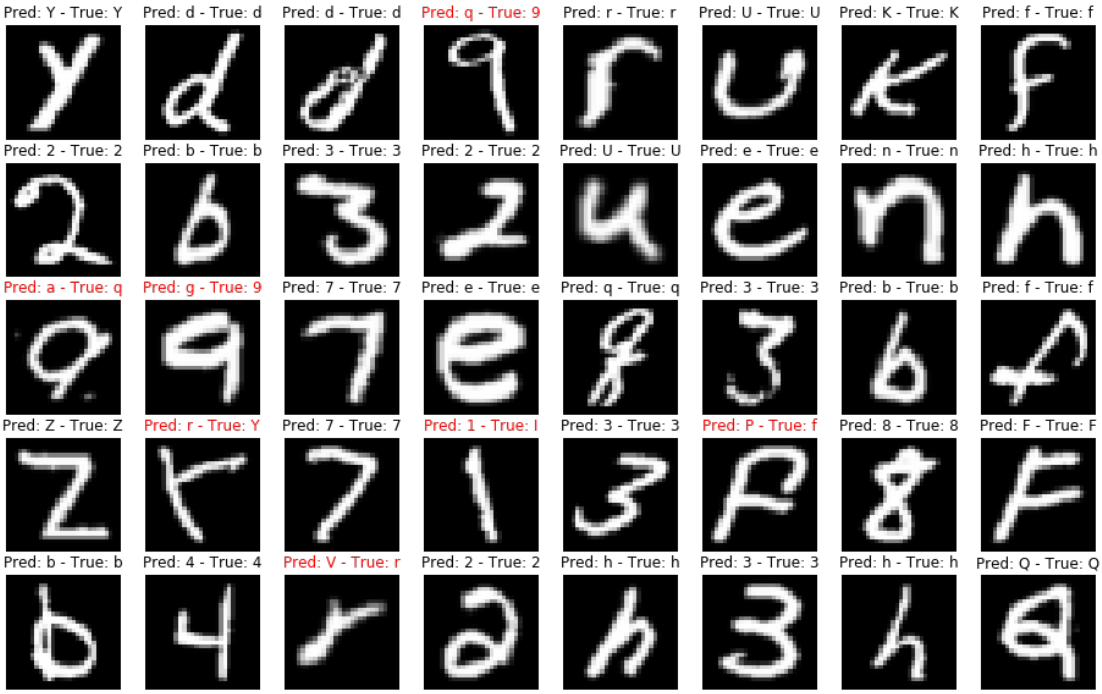
Hình 7.3. Nền tảng toán học đằng sau mô hình.

Chúng ta sẽ train mô hình với bộ dữ liệu EMNIST 50 lần (epoch), với 512 example mỗi minibatch. Ta thu được kết quả như sau:



Hình 7.4. Đồ thị của Cost function và độ chính xác của mô hình qua 50 epoch.

Thống kê độ chính xác, mô hình này đạt $\approx 0.7880851063829787$. Sau đây là dữ liệu mẫu, chữ màu đỏ thể hiện mô hình đã nhận diện sai:



Hình 7.5. Ví dụ các lần nhận diện của mô hình.

Tài liệu

[1] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. EMNIST: an extension of MNIST to handwritten letters. *CoRR*, abs/1702.05373, 2017.

[2] Vinh Nguyen. Vanishing & exploding gradients problems in deep neural networks (part 1).

[3] Vinh Nguyen. Vanishing & exploding gradients problems in deep neural networks (part 2).

[4] Michael Nielsen. Neural networks and deep learning.

[5] Tiep Vu. Machine learning cơ bản.