

**ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN**

----



HUS
VNU UNIVERSITY OF SCIENCE

TIÊU LUẬN CUỐI KỲ

Môn học: Tính toán song song

Giảng viên: TS. Nguyễn Hải Vinh

**Chủ đề: Fashion Items Image Classification With One Hidden
Layer Using Parallel API**

NHÓM 5

Họ và tên: Nguyễn Chí Dũng – MSV: 20002040

Họ và tên: Nguyễn Hoàng Giang – MSV: 20002048

Họ và tên: Đinh Tiến Dũng – MSV: 20002039

Họ và tên: Đinh Thành Công – MSV: 20002033

Lớp: K65A5 Khoa học dữ liệu – Khoa: Toán – Cơ – Tin học

LỜI MỞ ĐẦU

1. Tổng quan về đề tài

Tiểu luận này sẽ đưa chúng ta đến hành trình xây dựng mạng lưới thần kinh nhận biết được những hình ảnh thời trang. Chúng ta bắt đầu với những đơn vị dự đoán cẩn bản và từ từ cải thiện khi nó chạm đến giới hạn của nó. Dọc theo hành trình này chúng ta cũng sẽ học về một số ngữ cảnh toán học cần thiết để hiểu cách mạng lưới thần kinh học và dự đoán các giải pháp cho các vấn đề. Cụ thể, chúng ta cùng tìm hiểu về cấu trúc của mạng lưới thần kinh truyền thẳng, một số đơn vị cấu thành mạng lưới thần kinh như perceptron, tuyến tính, sigmoid, đưa ra những ý tưởng toán học, thuật toán cốt lõi như gradient descent, incremental gradient descent, backpropagation. Từ đó, chúng ta áp dụng xây dựng chương trình về một mạng lưới thần kinh truyền thẳng với một lớp ẩn sử dụng thuật toán backpropagation đồng thời áp dụng tính toán song song để chương trình thực thi có tính đồng bộ hóa nhằm tối ưu về thời gian phân loại các hình ảnh thời trang từ bộ dữ liệu Fashion-MNIST.

2. Luận chứng lựa chọn đề tài

Sự lạc quan và tham vọng về trí tuệ nhân tạo đã lên cao khi chủ đề này được chính thức hóa vào những năm 1950. Những thành công ban đầu khi máy tính chơi các trò chơi đơn giản và chứng minh các định lý. Một số người tin rằng những cỗ máy có trí thông minh ở cấp độ con người sẽ xuất hiện trong vòng một thập kỷ hoặc lâu hơn.

Nhưng trí tuệ nhân tạo tỏ ra khó khăn và tiến độ bị đình trệ. Những năm 1970 chứng kiến một thách thức học thuật tàn khốc đối với tham vọng về trí tuệ nhân tạo, sau đó là cắt giảm tài trợ và mất hứng thú.

Có vẻ như những cỗ máy logic khô cứng, với 1 và 0, sẽ không bao giờ có thể đạt được các sắc thái, đôi khi mờ nhạt, quá trình suy nghĩ của bộ não sinh học.

Sau một thời gian không có nhiều tiến triển, một ý tưởng vô cùng quyền năng đã xuất hiện để đưa việc kiểm tra thông minh của cỗ máy ra khỏi lối mòn của nó. Tại sao không thử xây dựng bộ não nhân tạo bằng cách sao chép cách bộ não sinh học thực sự hoạt động? Bộ não thực sự với các tế bào thần kinh thay vì cỗng logic, thay vì các thuật toán truyền thống khô khan.

Các nhà khoa học đã lấy cảm hứng từ sự đơn giản, rõ ràng của bộ não của một con ong hoặc chim bồ câu so với những nhiệm vụ phức tạp mà chúng có thể thực hiện. Bộ não có kích thước chỉ bằng một phần của một gam đường như có thể làm những việc như điều khiển hướng bay và thích ứng với gió, xác định thức ăn và kẻ săn mồi, đồng thời nhanh chóng quyết định nên chiến đấu hay chạy trốn. Chắc chắn máy tính, giờ đây với nguồn tài nguyên khổng lồ, có thể bắt chước và cải thiện những bộ não này? Một con ong có khoảng 950.000

tế bào thần kinh, liệu máy tính ngày nay với hàng gigabyte và terabyte tài nguyên có thể vượt trội hơn loài ong?

Nhưng với các phương pháp truyền thống để giải quyết vấn đề, những máy tính này với bộ lưu trữ lớn và bộ xử lý cực nhanh không thể đạt được điều mà bộ não tương đối nhỏ của loài chim và loài ong có thể làm được.

Mạng lưới thần kinh xuất hiện từ sinh học truyền cảm hứng cho tính toán thông minh và tiếp tục trở thành một trong những phương pháp mạnh mẽ và hữu ích nhất trong lĩnh vực trí tuệ nhân tạo. Ngày nay, Deepmind của Google, đã đạt được những điều tuyệt vời như tự học cách chơi trò chơi điện tử và đánh bại một bậc thầy thế giới trong trò chơi cờ vây, có mạng lưới thần kinh làm nền tảng. Mạng lưới thần kinh đã là trung tâm của công nghệ hàng ngày như nhận dạng biển số ô tô tự động hay xe tự động lái,..

Chúng ta sử dụng bộ dữ liệu Fashion-MNIST thay cho những bộ dữ liệu ảnh khác ví dụ như bộ dữ liệu MNIST về chữ viết tay truyền thống. Bởi bộ dữ liệu Fashion-MNIST là một bộ dữ liệu phức tạp chứa đựng những bức ảnh thời trang hiện đại một cách chi tiết.



Ta có thể thấy được độ chi tiết, sắc nét của những hình ảnh thời trang về hình dáng các mẫu mã như váy, áo phông, túi,... cũng như chi tiết các họa tiết trên từng hình ảnh của các mẫu thời trang. Do đó, đây là một bộ dữ liệu đầy thách thức đòi hỏi phải triển khai mạng lưới thần kinh một cách hiệu quả để cho ra có thể phân loại một cách chính xác các mẫu thời trang.

MỤC LỤC

Chương 1. Cơ sở lý thuyết.....	4
1.1 Tổng quan về mạng lưới thần kinh truyền thẳng	4
1.1.1 Khái niệm mạng lưới thần kinh truyền thẳng	4
1.1.2 Neuron là gì?.....	4
1.1.3 Tại sao gọi mạng lưới thần kinh là hộp đen?	5
1.1.4 Perceptron	5
1.1.4.1 Perceptron là gì?.....	7
1.1.4.2 Quyền năng của perceptron.....	8
1.1.5 Mạng lưới thần kinh nhiều lớp	9
1.2 Thuật toán Backpropagation	10
1.2.1 Quy tắc huấn luyện cho perceptron	10
1.2.2 Gradient Descent và quy tắc Delta.....	11
1.2.2.1 Trực quan hóa không gian giả thuyết.....	12
1.2.2.2 Đạo hàm của quy tắc Gradient Descent.....	15
1.2.2.3 Incremental Gradient Descent.....	17
1.2.3 Thuật toán Backpropagation cho mạng lưới thần kinh nhiều lớp	18
1.2.3.1 Đơn vị ngưỡng chứa hàm khả vi	
1.2.3.2 Thuật toán Backpropagation áp dụng cho mạng lưới thần kinh truyền thẳng	
1.3 Chuẩn hóa dữ liệu	24
1.3.1 Chuẩn hóa dữ liệu đầu vào.....	24
1.3.2 Chọn đầu ra mục tiêu	26
1.3.3 Khởi tạo ngẫu nhiên trọng số	27
Chương 2. Áp dụng tính toán song song.....	28
2.1 Tổng quan về OpenMP.....	28
 2.2 Các thành phần được sử dụng của OpenMP	
2.2.1 Mệnh đề: for num_threads().....	29
2.2.2 Vòng lặp song song	29
Chương 3. Trực quan hóa dữ liệu.....	31
3.1 Bộ dữ liệu Fashion MNIST.....	31
3.1.1 Sơ lược về bộ dữ liệu Fashion MNIST	31
3.1.2 Nội dung của bộ dữ liệu Fashion MNIST	32
3.2 Trực quan hóa bộ dữ liệu thời trang Fashion MNIST	33
Chương 4. Xây dựng, thực thi chương trình	36
4.1 Tổng quan chương trình	36
4.2 Quy trình thực hiện	36
4.3 Thực thi chương trình	43
TÀI LIỆU THAM KHẢO.....	48

NỘI DUNG

Chương 1. Cơ sở lý thuyết

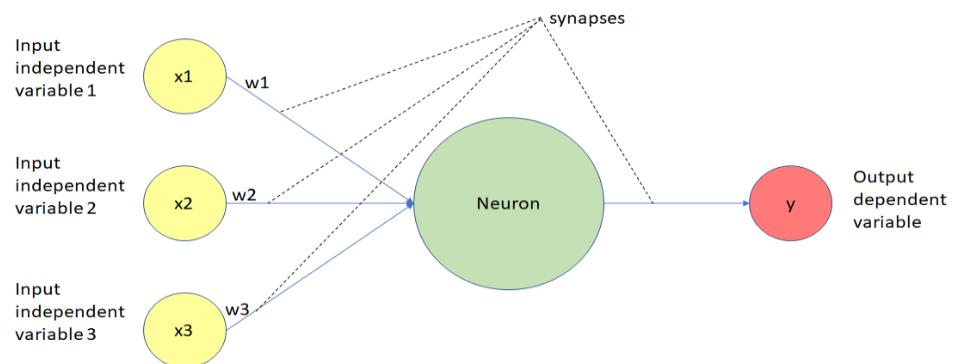
1.1 Tổng quan về mạng lưới thần kinh truyền thẳng

1.1.1 Khái niệm mạng lưới thần kinh truyền thẳng

Mạng lưới thần kinh truyền thẳng là mạng lưới thần kinh nhân tạo nơi mà kết nối giữa các neuron không tạo thành một chu trình. Trong mạng lưới này, thông tin chỉ được truyền theo một chiều – truyền thẳng từ các neuron đầu vào, đến các neuron ở lớp ẩn (nếu có) và đến các neuron đầu ra. Chú ý rằng không có chu trình hay vòng lặp nào trong mạng lưới này.

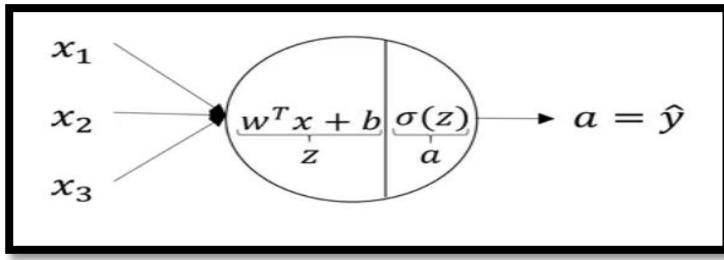
1.1.2 Neuron là gì?

Khối xây dựng cơ bản của mạng lưới thần kinh chính là neuron. Một neuron cơ bản bao gồm hộp đen (Nơi chứa các thông tin quan trọng mà không làm lộ bất kỳ thông tin nào về quá trình hoạt động bên trong) với các trọng số đầu vào (Weighted inputs) và đầu ra (output).



Ta có thể thấy trong hình minh họa, neuron có các khớp thần kinh (synapses) kết nối từ lớp (layer) trước. Mỗi khớp thần kinh có liên kết với các trọng số w .

Trong khu vực hộp đen của neuron bao gồm các hàm kích hoạt (activation functions)



Đầu tiên là Neuron pre-activation function (Hàm tiền kích hoạt):

$$z(x) = b + \sum_i w_i x_i = b + \mathbf{w}^\top \mathbf{x}$$

Sau đó là Neuron activation function (Hàm kích hoạt):

$$a(z) = g(z(x)) = g(b + \sum_i w_i x_i)$$

Trong đó:

- w : trọng số
- b : neuron bias
- $g()$: Hàm kích hoạt

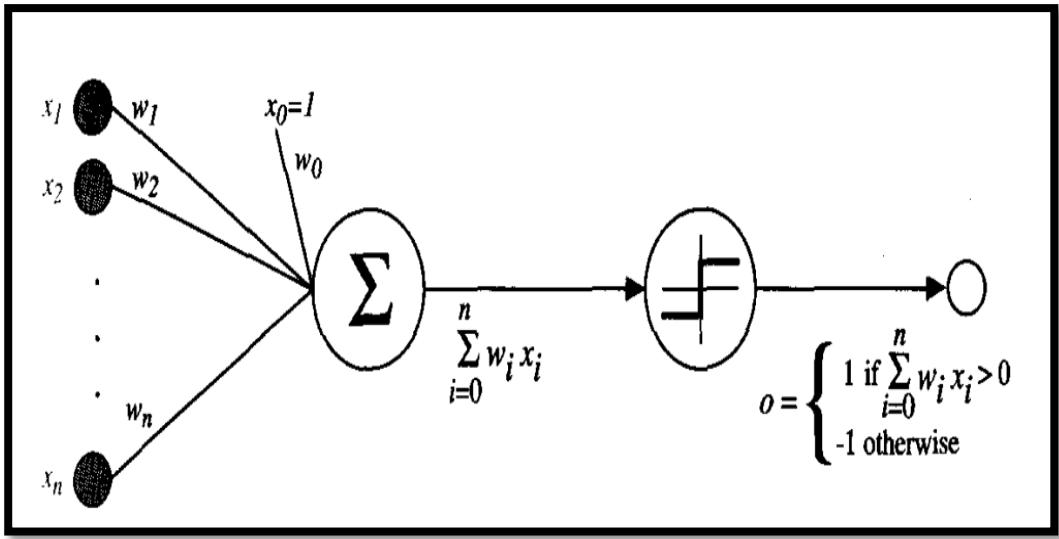
1.1.3 Tại sao gọi mạng lưới thần kinh là hộp đen?

Các trọng số mà mạng lưới thần kinh học được thường rất khó để ta có thể diễn giải. Ví dụ khi ta có một mạng lưới thần kinh và khi ta chọn một neuron cụ thể để phân tích cách thức nó hoạt động trong quá trình đưa ra kết quả cuối cùng của mạng lưới thần kinh, nó thật sự phức tạp do quá trình cập nhật trọng số cũng như toán học phi tuyến tính nhiều chiều. Do vậy chúng ta không biết được các neuron riêng lẻ hoạt động với nhau như thế nào để đưa ra kết quả đầu ra cuối cùng và nhiều khi không biết một neuron bất kỳ đang tự làm gì.

1.1.4 Perceptron

1.1.4.1 Perceptron là gì?

Perceptron là một neuron nhân tạo. Nó là mạng lưới thần kinh đơn giản nhất.



Một perceptron lấy một vectơ đầu vào có giá trị thực, tính toán tổ hợp tuyến tính của các đầu vào này, sau đó xuất ra 1 nếu kết quả lớn hơn một ngưỡng nhất định và -1 nếu ngược lại. Chính xác hơn, các đầu vào đã cho từ x_1 đến x_n đầu ra $o(x_1, \dots, x_n)$ được tính bởi perceptron là:

$$o(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

Trong đó mỗi w_i là một hằng số có giá trị thực hoặc trọng số, xác định sự đóng góp của đầu vào x_i vào đầu ra perceptron. Lưu ý rằng $(-w_0)$ là một ngưỡng mà sự kết hợp có trọng số của các đầu vào $w_1x_1 + \dots + w_nx_n$ phải vượt qua để perceptron xuất ra 1.

Để đơn giản hóa ký hiệu, chúng ta thêm một đầu vào hằng số bổ sung $x_0 = 1$, cho phép chúng ta viết bất đẳng thức trên dưới dạng:

$$\sum_{i=0}^n w_i x_i > 0,$$

Để ngắn gọn, đôi khi chúng ta viết hàm perceptron dưới dạng:

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

Với:

$$sgn(y) = \begin{cases} 1 & \text{if } y > 0 \\ -1 & \text{otherwise} \end{cases}$$

Học một perceptron liên quan đến việc chọn giá trị cho các trọng số w_0, \dots, w_n . Do đó, không gian H của các ứng cử viên giả thuyết được xem xét trong việc học perceptron là tập hợp tất cả các vecto trọng số có giá trị thực.

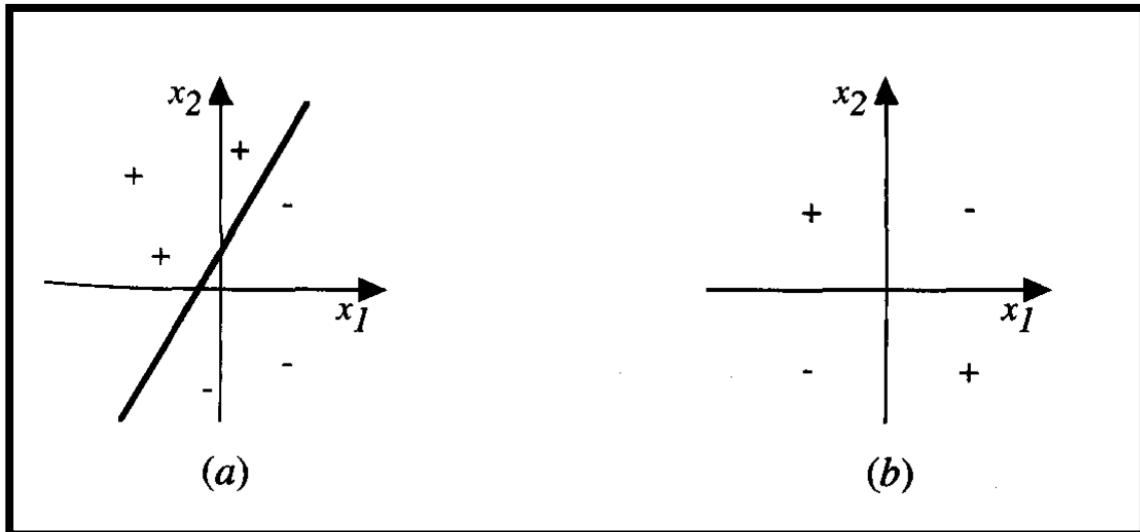
$$H = \{\vec{w} \mid \vec{w} \in \Re^{(n+1)}\}$$

1.1.4.2 Quyền năng của perceptron

Chúng ta có thể xem perceptron như đại diện cho một bờ mặt quyết định siêu phẳng trong không gian n chiều của các điểm. Perceptron xuất ra giá trị 1 đối với các trường hợp nằm ở một bên của siêu phẳng và xuất ra -1 đối với các trường hợp nằm ở phía bên kia. Phương trình cho quyết định này là $\vec{w} \cdot \vec{x} = 0$. Tất nhiên, một số tập ví dụ dương và âm không thể được phân tách bằng bất kỳ siêu phẳng nào. Những cái có thể tách được gọi là các tập ví dụ có thể tách tuyến tính.

Một perceptron có thể được sử dụng để biểu diễn nhiều hàm boolean. Ví dụ: nếu ta giả sử các giá trị boolean là 1 (true) và -1 (false), thì một cách để sử dụng perceptron hai đầu vào để triển khai hàm AND là đặt trọng số $w_0 = -0.8$ và $w_1 = w_2 = 0.5$. Perceptron này có thể được tạo để biểu diễn hàm OR thay vì hàm AND bằng cách thay đổi ngưỡng thành $w_0 = -0.3$. Trên thực tế, AND và OR có thể được xem là các trường hợp đặc biệt của các hàm m-của-n: Nghĩa là các hàm mà ít nhất m của n đầu vào của perceptron phải đúng. Hàm OR tương ứng với $m = 1$ và hàm AND tương ứng với $m = n$. Bất kỳ hàm m của n nào cũng được biểu diễn dễ dàng bằng perceptron bằng cách đặt tất cả các trọng số đầu vào thành cùng một giá trị (ví dụ: 0,5) và sau đó đặt ngưỡng w_0 tương ứng.

Perceptron có thể đại diện cho tất cả các hàm boolean nguyên thủy AND, OR, NAND và NOR. Tuy nhiên, thật không may, một số hàm boolean không thể được biểu diễn bằng một perceptron duy nhất, chẳng hạn như hàm XOR có giá trị là 1 khi và chỉ khi x_1 khác x_2 . Lưu ý tập hợp các ví dụ huấn luyện không thể tách tuyến tính được hiển thị trong hình 1.1(b) tương ứng với hàm XOR này.

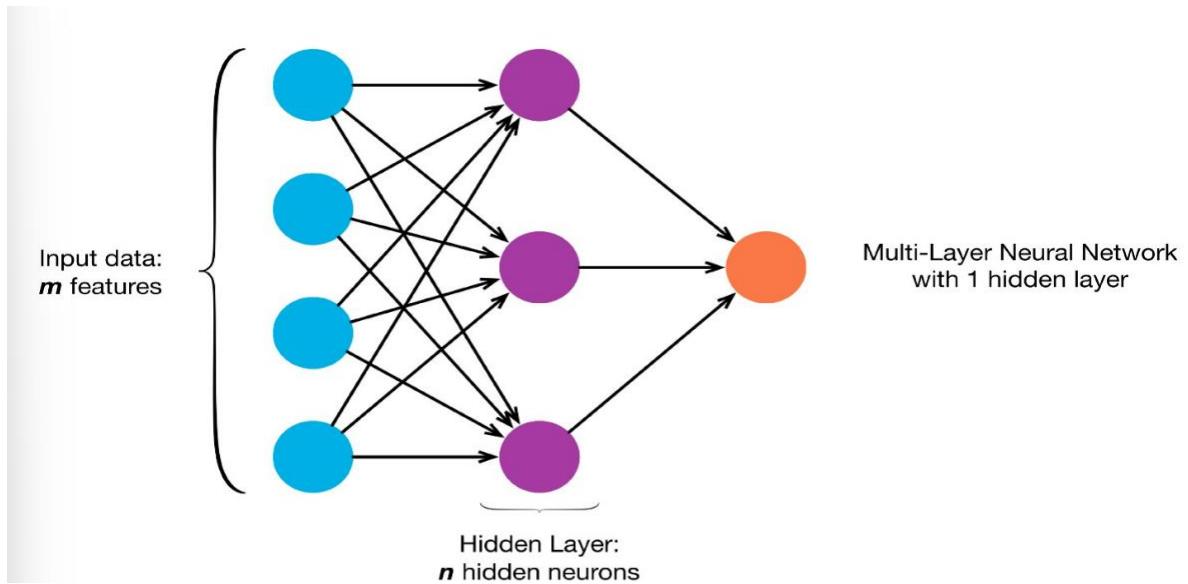


Hình minh họa 1.1: Bè mặt quyết định được đại diện bởi một perceptron với hai đầu vào (a). Một tập hợp các ví dụ huấn luyện và bè mặt quyết định của một perceptron phân loại chúng một cách chính xác (b). Một tập hợp các ví dụ huấn luyện không thể phân tách tuyến tính (nghĩa là không thể phân loại chính xác theo bất kỳ đường thẳng nào). x_1 và x_2 là đầu vào perceptron. Các ví dụ dương được biểu thị bằng "+", âm bằng "-".

Khả năng của các perceptron biểu diễn AND, OR, NAND và NOR là quan trọng vì mọi hàm boolean có thể được biểu diễn bằng một số mạng lưới của các đơn vị được kết nối với nhau dựa trên các hàm nguyên thủy này. Trên thực tế, mọi hàm boolean có thể được biểu diễn bằng một số mạng của perceptron với độ sâu là 2, trong đó các đầu vào được truyền tới cho nhiều đơn vị và đầu ra của các đơn vị này sau đó được đưa vào giai đoạn thứ hai, cuối cùng.

Bởi lẽ, các mạng lưới của các đơn vị ngưỡng có thể biểu diễn nhiều chức năng phong phú mà các đơn vị đơn riêng lẻ không thể biểu diễn, nên chúng ta sẽ quan tâm đến việc học các mạng lưới nhiều lớp của các đơn vị ngưỡng.

1.1.5 Mạng lưới thần kinh nhiều lớp



Một mạng lưới thần kinh nhiều lớp gồm có ít nhất 3 lớp: một lớp đầu vào, một lớp ẩn và một lớp đầu ra. Lớp ẩn của mạng lưới thần kinh đơn giản chỉ là thêm những neuron vào giữa lớp đầu vào và lớp đầu ra.

Trước tiên ta có m dữ liệu đầu vào (x_1, x_2, \dots, x_m), ta gọi là m đặc trưng. Đặc trưng đơn giản là một biến ta nhận định có ảnh hưởng tới một đầu ra nhất định.

$$W \cdot X = w_1x_1 + w_2x_2 + \dots + w_mx_m = \sum_{i=1}^m w_i x_i$$

Definition of a Dot Product

Những điều ta cần làm là:

1. Với m đặc trưng của đầu vào X , ta cần m trọng số để biểu diễn tích vô hướng
2. Với n neuron ẩn trong lớp ẩn, ta cần n bộ trọng số (W_1, W_2, \dots, W_n) để biểu diễn tích vô hướng
3. Với một lớp ẩn, ta biểu diễn n tích vô hướng để có đầu ra lớp ẩn h : (h_1, h_2, \dots, h_n)
4. Và giờ giống như một lớp perceptron hay neuron, ta chỉ cần dùng đầu ra của lớp ẩn h : (h_1, h_2, \dots, h_n) như là dữ liệu đầu ra với n đặc trưng, biểu diễn tích vô hướng với một bộ trọng số n (w_1, w_2, \dots, w_n) để có kết quả đầu ra cuối cùng.

1.2 Thuật toán Backpropagation

1.2.1 Quy tắc huấn luyện cho perceptron

Mặc dù chúng ta quan tâm đến việc học mạng lưới có nhiều đơn vị được kết nối với nhau, nhưng chúng ta hãy bắt đầu bằng cách tìm hiểu cách học các trọng số cho một perceptron đơn lẻ. Ở đây, vấn đề học chính xác là xác định một vectơ trọng số làm cho perceptron tạo ra đầu ra ± 1 chính xác cho mỗi ví dụ huấn luyện đã cho

Một số thuật toán được biết để giải quyết vấn đề học này. Ở đây chúng ta xem xét hai: quy tắc perceptron và quy tắc delta. Hai thuật toán này được đảm bảo hội tụ đến các giả thuyết có thể chấp nhận được hơi khác nhau, trong các điều kiện hơi khác nhau. Chúng rất quan trọng đối với mạng lưới thần kinh vì chúng cung cấp cơ sở cho việc học của các mạng lưới với nhiều đơn vị.

Một cách để học một vectơ trọng số có thể chấp nhận được là bắt đầu với các trọng số ngẫu nhiên, sau đó lặp đi lặp lại áp dụng perceptron cho từng ví dụ huấn luyện, sửa đổi các trọng số perceptron bắt cứ khi nào nó phân loại sai một ví dụ. Quá trình này được lặp đi lặp lại qua các ví dụ huấn luyện nhiều lần nếu cần cho đến khi perceptron phân loại chính xác tất cả các ví dụ huấn luyện. Các trọng số được sửa đổi ở mỗi bước theo quy tắc huấn luyện perceptron, quy tắc này sửa đổi trọng số w_i được liên kết với đầu vào xi theo quy tắc.

$$w_i \leftarrow w_i + \Delta w_i$$

Với:

$$\Delta w_i = \eta(t - o)x_i$$

Ở đây t là đầu ra mục tiêu cho ví dụ huấn luyện hiện tại, o là đầu ra được tạo bởi perceptron và η là một hằng số dương được gọi là tốc độ học. Vai trò của tốc độ học là điều chỉnh mức độ thay đổi trọng số ở mỗi bước. Nó thường được đặt với giá trị nhỏ (Ví dụ: 0,1) và đôi khi bị giảm dần khi số lần lặp lại điều chỉnh trọng số tăng lên.

Tại sao quy tắc cập nhật này có thể hội tụ về giá trị trọng số thành công? Để có được cảm nhận trực quan, hãy xem xét một số trường hợp cụ thể. Giả sử ví dụ huấn luyện đã được phân loại chính xác bởi perceptron. Trong trường hợp này, $(t - o)$ bằng 0, làm cho Δw_i bằng 0, do đó không có trọng số nào

được cập nhật. Giả sử đầu ra perceptron là -1, khi đầu ra mục tiêu là +1. Để làm cho đầu ra perceptron là +1 thay vì -1 trong trường hợp này, trọng số phải được thay đổi để tăng giá trị của $\vec{w} \cdot \vec{x}$. Ví dụ: nếu $x_i > 0$, thì việc tăng w_i sẽ đưa perceptron tiến gần hơn đến việc phân loại chính xác ví dụ này. Lưu ý rằng quy tắc huấn luyện sẽ tăng w_i , trong trường hợp này, bởi vì $(t - o)$, η và x_i đều dương. Ví dụ: nếu $x_i = 0.8$, $\eta = 0.1$, $t = 1$ và $o = -1$ thì cập nhật trọng số sẽ là $\Delta w_i = \eta(t - o)x_i = 0.1(1 - (-1)) 0.8 = 0.16$. Mặt khác, nếu $t = -1$ và $o = 1$, thì trọng số tương ứng với x_i dương sẽ giảm thay vì tăng.

Trên thực tế, quy trình học ở trên có thể được chứng minh là hội tụ trong một số lượng hữu hạn các ứng dụng của quy tắc huấn luyện perceptron đối với một vectơ trọng số phân loại chính xác tất cả các ví dụ huyền luận, miễn là các ví dụ huấn luyện có thể phân tách tuyến tính và sử dụng η đủ nhỏ (xem trong cuốn Perceptrons | Marvin Minsky, Seymour Papert 1969) . Nếu dữ liệu không thể phân tách tuyến tính, sự hội tụ không được đảm bảo.

1.2.2 Gradient Descent và quy tắc Delta

Mặc dù quy tắc perceptron tìm thấy một vectơ trọng số thành công khi các ví dụ huấn luyện có thể phân tách tuyến tính, nhưng nó có thể không hội tụ nếu các ví dụ không thể phân tách tuyến tính. Quy tắc huấn luyện thứ hai, được gọi là quy tắc delta, được thiết kế để vượt qua khó khăn này. Nếu các ví dụ huấn luyện không thể phân tách tuyến tính, quy tắc delta sẽ hội tụ hướng tới một xấp xỉ phù hợp nhất với ngữ cảnh mục tiêu.

Ý tưởng chính đằng sau quy tắc delta là sử dụng gradient descent để tìm kiếm không gian giả thuyết của các vectơ trọng số khả dĩ nhằm tìm ra các trọng số phù hợp nhất với các ví dụ huấn luyện. Quy tắc này rất quan trọng vì gradient descent cung cấp cơ sở cho thuật toán backpropagation, thuật toán này có thể học các mạng lưới thần kinh có nhiều đơn vị được kết nối với nhau. Nó cũng quan trọng vì gradient descent có thể đóng vai trò là cơ sở cho các thuật toán học phải tìm kiếm trong các không gian giả thuyết chứa nhiều loại tham số hóa liên tục được giả thuyết khác nhau.

Quy tắc huấn luyện delta được hiểu rõ nhất bằng cách xem xét nhiệm vụ huấn luyện một perceptron không có ngưỡng; nghĩa là, một đơn vị tuyến tính mà đầu ra o được cho bởi:

$$o(\vec{x}) = \vec{w} \cdot \vec{x} \quad (1.1)$$

Như vậy, một đơn vị tuyến tính tương ứng với giai đoạn đầu tiên của perceptron không có ngưỡng.

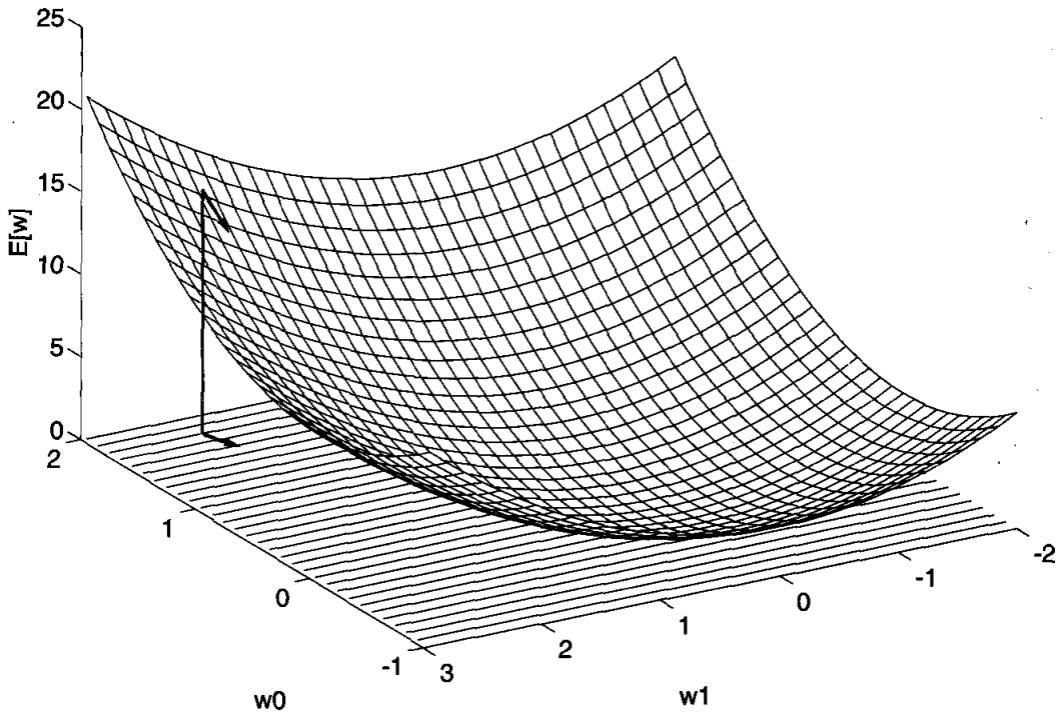
Để rút ra quy tắc học trọng số cho các đơn vị tuyến tính, chúng ta hãy bắt đầu bằng cách chỉ định thước đo cho sai số huấn luyện của giả thuyết (vectơ trọng số), liên quan đến các ví dụ huấn luyện. Mặc dù có nhiều cách để xác định sai số này, một biện pháp phổ biến sẽ trở nên đặc biệt thuận tiện là:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \quad (1.2)$$

trong đó D là tập hợp các ví dụ huấn luyện, t_d là đầu ra mục tiêu cho ví dụ huấn luyện d và o_d là đầu ra của đơn vị tuyến tính cho ví dụ huấn luyện d. Theo định nghĩa này, $E(\vec{w})$ đơn giản là một nửa bình phương chênh lệch giữa đầu ra mục tiêu t_d và đầu ra đơn vị tuyến tính o_d , tổng của tất cả các ví dụ huấn luyện. Ở đây chúng ta mô tả E là một hàm của \vec{w} , bởi vì đầu ra đơn vị tuyến tính o phụ thuộc vào vectơ trọng số này. Tất nhiên E cũng phụ thuộc vào tập ví dụ huấn luyện cụ thể, nhưng ta giả định chúng được cố định trong quá trình huấn luyện.

1.2.2.1 Trực quan hóa không gian giả thuyết

Để hiểu thuật toán gradient descent, sẽ rất hữu ích nếu trực quan hóa toàn bộ không gian giả thuyết của các vectơ trọng số khả dĩ và các giá trị E tương ứng của chúng, như được minh họa trong hình (1.2). Ở đây, các trục w_0 và w_1 biểu thị các giá trị có thể có cho hai trọng số của một đơn vị tuyến tính đơn giản. Do đó, **Mặt phẳng** w_0, w_1 biểu diễn toàn bộ không gian giả thuyết. Trục tung biểu thị sai số E liên quan đến một số ví dụ huấn luyện cố định. Do đó, **Bề mặt** sai số được hiển thị trong hình cô đọng mong muốn của mọi vectơ trọng số trong không gian giả thuyết (chúng ta mong muốn một giả thuyết có sai số tối thiểu). Dựa vào cách mà ta đã chọn để xác định E, đối với các đơn vị tuyến tính, bề mặt sai số này phải luôn là hình parabol với một cực tiểu toàn cục duy nhất. Tất nhiên, parabol cụ thể sẽ phụ thuộc vào tập ví dụ huấn luyện cụ thể.



Hình minh họa 1.2 Sai số của các giả thuyết khác nhau. Đối với một đơn vị tuyến tính có hai trọng số, không gian giả thuyết H là mặt phẳng w_0, w_1 . Trục tung biểu thị sai số của không gian giả thuyết vecto trọng số tương ứng, liên quan đến một tập các ví dụ huấn luyện cố định. Mũi tên hiển thị gradient tại một điểm cụ thể, biểu thị hướng đi trong mặt phẳng w_0, w_1 tạo ra độ dốc lớn nhất dọc theo bề mặt sai số.

Gradient descent tìm, xác định một vectơ trọng số làm giảm thiểu E bằng cách bắt đầu với một vectơ trọng số ban đầu tùy ý, sau đó liên tục sửa đổi nó theo các bước nhỏ. Tại mỗi bước, vectơ trọng số được thay đổi theo hướng tạo ra độ dốc lớn nhất dọc theo bề mặt sai số. Quá trình này tiếp tục cho đến khi đạt được sai số tối thiểu toàn cục.

1.2.2.2 Đạo hàm của quy tắc Gradient Descent

Làm thế nào chúng ta có thể tính toán hướng dốc nhất dọc theo bề mặt sai số? Hướng này có thể tìm được bằng cách tính đạo hàm của E đối với từng thành phần của vectơ \vec{w} . Đạo hàm vectơ này được gọi là gradient của E đối với \vec{w} , được viết là $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right] \quad (1.3)$$

Lưu ý $\nabla E(\vec{w})$ tự nó là một vectơ, có các thành phần là đạo hàm riêng của E đối với từng w_i . Khi được hiểu là một vectơ trong không gian trọng số,

gradient xác định hướng tạo ra mức tăng dốc nhất trong E. Do đó, giá trị âm của vectơ này cho hướng giảm dốc nhất. Ví dụ, mũi tên trong hình (1.2) hiển thị gradient đối nghịch $-\nabla E(\vec{w})$ cho một điểm cố định trong mặt w_0, w_1 .

Do độ dốc xác định hướng tăng dốc nhất của E, nên quy tắc huấn luyện gradient là:

$$\vec{w} \leftarrow \vec{w} + \Delta \vec{w}$$

Với:

$$\Delta \vec{w} = -\eta \nabla E(\vec{w}) \quad (1.4)$$

Ở đây η là hằng số dương được gọi là tốc độ học, xác định kích thước bước trong tìm kiếm gradient descent. Dấu âm xuất hiện vì chúng ta muốn di chuyển vectơ trọng số theo hướng giảm E. Quy tắc huấn luyện này cũng có thể được viết ở dạng thành phần của nó:

$$w_i \leftarrow w_i + \Delta w_i$$

Với:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \quad (1.5)$$

làm rõ độ dốc lớn nhất đạt được bằng cách thay đổi từng thành phần w_i của \vec{w} theo tỷ lệ với $\frac{\partial E}{\partial w_i}$.

Để xây dựng một thuật toán thực tế để cập nhật lặp lại các trọng số theo phương trình (1.5), chúng ta cần một cách hiệu quả để tính toán gradient ở mỗi bước. May mắn thay, điều này không khó. Vectơ của các đạo hàm $\frac{\partial E}{\partial w_i}$ thứ hình thành nên gradient có thể thu được bằng cách vi phân E từ phương trình (1.2).

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d) \\
\frac{\partial E}{\partial w_i} &= \sum_{d \in D} (t_d - o_d) (-x_{id})
\end{aligned}$$

(1.6)

Trong đó x_{id} biểu thị thành phần đầu vào x_i cho ví dụ huấn luyện d . Bây giờ chúng ta có một phương trình đưa ra $\frac{\partial E}{\partial w_i}$ dưới dạng đầu vào đơn vị tuyến tính x_{id} , đầu ra o_d và giá trị mục tiêu t_d được liên kết với các ví dụ huấn luyện. Thế phương trình (1.6) vào phương trình (1.5) có được quy tắc cập nhật trọng số cho gradient descent:

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id} \quad (1.7)$$

1.2.2.3 Incremental Gradient Descent

Gradient descent là một phương pháp quan trọng trong việc học trọng số. Đó là một chiến lược để tìm kiếm thông qua một không gian giả thuyết lớn hoặc vô hạn có thể được áp dụng bất cứ khi nào (1) không gian giả thuyết chứa các tham số hóa được giả thuyết liên tục (ví dụ: các trọng số trong một đơn vị tuyến tính) và (2) hàm sai số có thể vi phân với các tham số giả thuyết này. Những khó khăn thực tế chính trong việc áp dụng gradient descent là (1) hội tụ đến cực tiểu cục bộ đôi khi có thể khá chậm (nghĩa là có thể yêu cầu hàng nghìn bước gradient descent) và (2) nếu có nhiều cực tiểu cục bộ trong bề mặt sai số, thì không có gì đảm bảo rằng sẽ tìm được giá trị cực tiểu toàn cục.

Một biến thể phổ biến của gradient descent nhằm giảm bớt những khó khăn này được gọi là incremental gradient descent. Trong khi quy tắc huấn luyện gradient descent được trình bày trong phương trình (1.7) tính toán cập nhật trọng số sau khi tính tổng tất các ví dụ huấn luyện trong D, ý tưởng đằng sau incremental gradient descent là ước tính gradient descent tìm kiếm bằng cách cập nhật các trọng số tăng dần, theo phép tính sai số cho từng ví dụ riêng lẻ. Quy tắc huấn luyện sửa đổi giống với quy tắc huấn luyện được đưa ra bởi phương trình (1.7) ngoại trừ khi chúng ta lặp qua từng ví dụ huấn luyện, chúng ta cập nhật trọng số theo công thức:

$$\Delta w_i = \eta(t - o) x_i \quad (1.8)$$

trong đó t , o và x_i là giá trị mục tiêu, đầu ra đơn vị và đầu vào thứ i cho ví dụ huấn luyện được đề cập. Để sửa đổi thuật toán gradient descent trong bảng (1.1) cho việc thực thi incremental gradient descent, phương trình (T4.2) được xóa, và phương trình (T4.1) thay bằng phương trình (1.9)

$$w_i \leftarrow w_i + \eta(t - o) x_i \quad (1.9)$$

Một cách để xem incremental gradient descent này là xem xét một hàm sai số riêng biệt $E_d(\vec{w})$ xác định cho từng ví dụ huấn luyện riêng lẻ d như sau:

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2 \quad (1.10)$$

trong đó t_d và o_d là giá trị mục tiêu và giá trị đầu ra đơn vị cho ví dụ huấn luyện d. Incremental gradient descent lặp lại các ví dụ huấn luyện d trong D, tại mỗi lần lặp thay đổi trọng số theo gradient đối với $E_d(\vec{w})$. Quá trình cập nhật các trọng số này, khi được lặp lại trên tất cả các ví dụ huấn luyện, cung cấp một xấp xỉ hợp lý để giảm gradient đối với hàm sai số ban đầu của $E(\vec{w})$. Bằng cách làm cho giá trị của η (kích thước bước gradient descent) đủ nhỏ, incremental gradient descent có thể thực hiện việc xấp xỉ gradient descent tùy ý gần đúng. Sự khác biệt chính giữa gradient descent tiêu chuẩn và incremental gradient descent là:

- Trong gradient descent tiêu chuẩn, sai số được tính tổng trên tất cả các ví dụ trước khi cập nhật trọng số, trong khi ở incremental gradient descent, các trọng số được cập nhật khi kiểm tra từng ví dụ huấn luyện.

- Tính tổng trên nhiều ví dụ theo gradient descent tiêu chuẩn yêu cầu tính toán nhiều hơn cho mỗi bước cập nhật trọng số. Mặt khác, vì nó sử dụng độ dốc thực, gradient descent tiêu chuẩn thường được sử dụng với kích thước bước lớn hơn trên mỗi lần cập nhật trọng số so với incremental gradient descent.
- Trong trường hợp có nhiều cực tiểu cục bộ đối với $E(\vec{w})$, incremental gradient descent đôi khi có thể tránh rơi vào các cực tiểu cục bộ này vì nó sử dụng đa dạng $\nabla E_d(\vec{w})$ khác nhau thay vì $\nabla E(\vec{w})$ để hướng dẫn việc tìm kiếm của nó.

Tóm gọn, thuật toán gradient descent để huấn luyện các đơn vị tuyến tính như sau: Khởi tạo một vectơ trọng số ngẫu nhiên. Áp dụng đơn vị tuyến tính cho tất cả các ví dụ huấn luyện, sau đó tính toán Δw_i cho mỗi trọng số theo phương trình (1.7). Cập nhật từng trọng số w_i bằng cách thêm Δw_i , sau đó lặp lại quy trình này. Thuật toán này được đưa ra trong bảng (1.1). Bởi vì bề mặt sai số chỉ chứa một cực tiểu toàn cục, thuật toán này sẽ hội tụ thành một vectơ trọng số với sai số tối thiểu, bất kể các ví dụ huấn luyện có thể phân tách tuyến tính hay không, với tốc độ học η đủ nhỏ được sử dụng. Nếu η quá lớn, tìm kiếm gradient descent có nguy cơ vượt quá mức tối thiểu trong bề mặt sai số thay vì dừng tại nó. Vì lý do này, một sửa đổi phổ biến đối với thuật toán là giảm dần giá trị của η cũng như số bước gradient descent tăng lên nhưng nếu quá nhỏ sẽ rất tốn thời gian để thuật toán được thực thi xong.

GRADIENT-DESCENT(*training_examples*, η)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, where \vec{x} is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
 - Initialize each Δw_i to zero.
 - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
 - Input the instance \vec{x} to the unit and compute the output o
 - For each linear unit weight w_i , Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \quad (\text{T4.1})$$

- For each linear unit weight w_i , Do

$$w_i \leftarrow w_i + \Delta w_i \quad (\text{T4.2})$$

Bảng 1.1: Thuật toán gradient descent huấn luyện cho đơn vị tuyến tính. Để thực thi incremental gradient descent, phương trình (T4.2) được xóa, và phương trình (T4.1) thay bằng phương trình (1.9).

1.2.3 Thuật toán Backpropagation cho mạng lưới thần kinh nhiều lớp

1.2.3.1 Đơn vị ngưỡng chứa hàm khả vi

Loại đơn vị nào chúng ta nên sử dụng làm cơ sở để xây dựng mạng lưới thần kinh nhiều lớp? Ban đầu, chúng ta có thể muốn chọn các đơn vị tuyến tính được thảo luận trong phần trước, mà chúng ta đã đưa ra một quy tắc học gradient descent. Tuy nhiên, nhiều lớp đơn vị tuyến tính vẫn chỉ tạo ra các hàm tuyến tính và chúng ta muốn mạng lưới thần kinh có khả năng biểu diễn các hàm có độ phi tuyến tính cao. Đơn vị perceptron là một lựa chọn khả thi khác, nhưng ngưỡng không liên tục của nó làm cho nó không thể vi phân được và do đó không phù hợp cho gradient descent. Những gì chúng ta cần là một đơn vị có đầu ra là một hàm phi tuyến tính đầu vào của nó, đồng thời đầu ra của nó phải là một hàm khả vi đầu vào của nó. Một giải pháp là đơn vị sigmoid - một đơn vị rất giống perceptron, nhưng dựa trên hàm ngưỡng khả vi.

Đơn vị sigmoid được minh họa trong Hình 1.3. Giống như perceptron, đơn vị sigmoid trước tiên tính toán tổ hợp tuyến tính các đầu vào của nó, sau đó áp dụng một ngưỡng cho kết quả. Tuy nhiên, trong trường hợp của đơn vị sigmoid, đầu ra ngưỡng là một hàm liên tục đầu vào của nó. Chính xác hơn, đơn vị sigmoid tính toán đầu ra o của nó như sau:

$$o = \sigma(\vec{w} \cdot \vec{x})$$

Với:

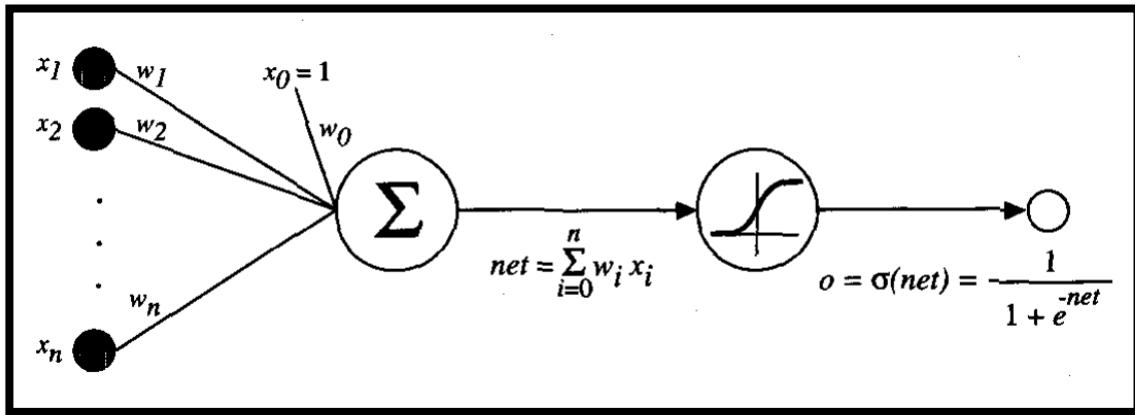
$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

σ thường được gọi là hàm sigmoid hay còn được gọi là hàm logistic. Lưu ý đầu ra của nó nằm trong khoảng $(0,1)$, tăng đơn điệu với đầu vào của nó. Bởi vì nó ánh xạ một miền đầu vào rất lớn tới một phạm vi nhỏ ở các đầu ra, nên nó thường được gọi là hàm nén của đơn vị. Hàm sigmoid có thuộc tính hữu ích là đạo hàm của nó dễ dàng được biểu thị theo đầu ra của nó.

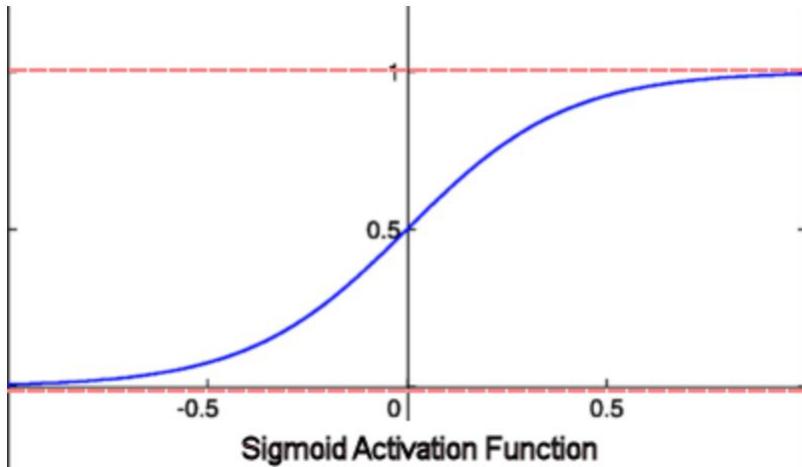
Cụ thể:

$$\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$$

Như ta thấy ở phần trên, quy tắc học gradient descent sử dụng đạo hàm này.



Hình 1.3: Đơn vị ngưỡng Sigmoid



Hình 1.4: Đồ thị hàm Sigmoid

1.2.3.2 Thuật toán Backpropagation áp dụng cho mạng lưới thần kinh truyền thẳng

Thuật toán backpropagation học các trọng số cho mạng lưới thần kinh có một tập hợp các đơn vị và kết nối cố định. Nó áp dụng gradient descent để cố gắng giảm thiểu sai số bình phương giữa các giá trị đầu ra của mạng và các giá trị mục tiêu cho các đầu ra này.

Bởi vì chúng ta đang xem xét mạng lưới thần kinh có nhiều đơn vị đầu ra thay vì các đơn vị đơn lẻ như trước đây, nên chúng ta bắt đầu bằng cách định nghĩa lại E để tính tổng các sai số trên tất cả các đơn vị đầu ra của mạng lưới:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

trong đó outputs là tập hợp các đơn vị đầu ra trong mạng lưới và t_{kd} và o_{kd} là các giá trị mục tiêu và các giá trị đầu ra được liên kết với đơn vị đầu ra thứ k và ví dụ huấn luyện d.

Vấn đề học mà backpropagation gặp phải là tìm kiếm một không gian giả thuyết lớn được xác định bởi tất cả các giá trị trọng số của tất cả các đơn vị trong mạng lưới thần kinh. Tình huống này có thể được hình dung dưới dạng bề mặt sai số tương tự như bề mặt được hiển thị cho các đơn vị tuyến tính trong Hình 1.2. Sai số trong sơ đồ đó được thay thế bằng định nghĩa mới của chúng ta về E và các chiều khác của không gian giờ đây tương ứng với tất cả các trọng số được liên kết với tất cả các đơn vị trong mạng lưới thần kinh. Như trong trường hợp huấn luyện một đơn vị duy nhất, gradient descent có thể được sử dụng để cố gắng tìm ra giả thuyết nhằm giảm thiểu E.

Để xây dựng thuật toán Backpropagation dựa trên incremental gradient descent trước tiên chúng ta đi tìm đạo hàm của quy tắc Backpropagation.

Nhắc lại từ phương trình (1.10) rằng incremental gradient descent liên quan đến việc lặp lại lần lượt qua các ví dụ huấn luyện cùng một lúc, đối với mỗi ví dụ huấn luyện d giảm gradient của sai số E_d . Nói cách khác, đối với mỗi ví dụ huấn luyện d, tất cả trọng số w_i được cập nhật bằng cách thêm nó với Δw_{ij}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \quad (1.11)$$

trong đó E_d là sai số trong ví dụ huấn luyện d, được tính tổng trên tất cả các đơn vị đầu ra trong mạng lưới thần kinh

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

Ở đây, outputs là tập hợp các đơn vị đầu ra trong mạng lưới thần kinh, t_k là giá trị mục tiêu của đơn vị k cho ví dụ huấn luyện d và o_k là đầu ra của đơn vị k cho bởi ví dụ huấn luyện d.

Đạo hàm của nguyên tắc incremental gradient descent về mặt khái niệm là đơn giản, nhưng yêu cầu theo dõi một số các chỉ số và biến. Chúng ta sẽ làm theo ký hiệu trong Hình 1.3, thêm chỉ số j để biểu thị cho đơn vị thứ j của mạng lưới như sau:

- x_{ij} : đầu vào thứ i của đơn vị j
- w_{ij} : trọng số tương ứng với đầu vào thứ i của đơn vị j
- net_j : $\sum_i w_{ji}x_{ji}$ tổng trọng số tất cả các đầu vào của đơn vị j
- o_j : đầu ra được tính bởi đơn vị j
- t_j : đầu ra mục tiêu của đơn vị j
- σ : hàm sigmoid
- outputs: tập các đơn vị trong lớp đầu ra cuối cùng của mạng lưới thần kinh
- Downstream(j): tập hợp các đơn vị có đầu vào lập tức bao gồm đầu ra của đơn vị j

Bây giờ chúng ta đưa ra một biểu thức cho $\frac{\partial E_d}{\partial w_{ij}}$ để thực thi quy tắc incremental gradient descent được thấy trong phương trình (1.11). Để bắt đầu, hãy lưu ý rằng trọng số w_{ji} có thể ảnh hưởng đến phần còn lại của mạng chỉ thông qua net_j .

Do vậy, ta có dùng quy tắc dây chuyền (chain rule) để viết:

$$\begin{aligned}\frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ji}} \\ &= \frac{\partial E_d}{\partial \text{net}_j} x_{ji}\end{aligned}\quad (1.12)$$

Với phương trình (1.12) nhiệm vụ còn lại của chúng ta là rút ra một biểu thức thuận tiện cho $\frac{\partial E_d}{\partial \text{net}_j}$. Chúng ta lần lượt xem xét hai trường hợp: Trường hợp đơn vị j là đơn vị đầu ra của mạng lưới và trường hợp j là đơn vị bên trong.

Trường hợp 1: Quy tắc huấn luyện cho trọng số của đơn vị đầu ra. Giống như w_{ij} chỉ có thể ảnh hưởng đến phần còn lại của mạng lưới thông qua net_j , net_j chỉ có thể ảnh hưởng đến mạng thông qua o_j . Do đó, chúng ta có thể gọi lại quy tắc dây chuyền để viết

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial net_j} \quad (1.13)$$

Trước tiên ta sẽ tính toán thành phần đầu tiên của phương trình (1.13):

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j} \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

Đạo hàm $\frac{\partial}{\partial o_j} (t_k - o_k)^2$ sẽ bằng 0 với tất cả các đơn vị đầu ra k trừ khi k = j. Do đó, ta bỏ việc tính tổng trên các đơn vị đầu ra và chỉ cần đặt k = j.

$$\begin{aligned} \frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j} \frac{1}{2} (t_j - o_j)^2 \\ &= \frac{1}{2} 2(t_j - o_j) \frac{\partial(t_j - o_j)}{\partial o_j} \\ &= -(t_j - o_j) \end{aligned} \quad (1.14)$$

Tiếp theo ta sẽ tính toán thành phần thứ hai của phương trình (1.13). Khi $o_j = \sigma(net_j)$, đạo hàm $\frac{\partial o_j}{\partial net_j}$ chỉ là đạo hàm của hàm sigmoid chính bằng:

$$\sigma(net_j)(1 - \sigma(net_j)).$$

Do đó:

$$\begin{aligned} \frac{\partial o_j}{\partial net_j} &= \frac{\partial \sigma(net_j)}{\partial net_j} \\ &= o_j(1 - o_j) \end{aligned} \quad (1.15)$$

Thay biểu thức (1.15) và (1.14) vào (1.13) ta thu được:

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j) o_j(1 - o_j)$$

Kết hợp với phương trình (1.11) và (1.12) ta có quy tắc incremental gradient descent cho các đơn vị đầu ra:

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta (t_j - o_j) o_j (1 - o_j) x_{ji}$$

Trường hợp 2: Quy tắc huấn luyện cho các trọng số của đơn vị ẩn. Trong trường hợp j là một đơn vị nội bộ hay ẩn trong mạng, việc tạo ra quy tắc huấn luyện cho w_{ji} phải tính đến các cách gián tiếp mà w_{ji} có thể ảnh hưởng đến đầu ra của mạng lưới và E_d . Vì lý do này, ta sẽ thấy hữu ích khi đề cập đến tập hợp tất cả các đơn vị ngay lập tức truyền xuống đơn vị j trong mạng lưới (nghĩa là tất cả các đơn vị có đầu vào trực tiếp bao gồm đầu ra của đơn vị j). Ta biểu thị tập hợp các đơn vị này bằng $Downstream(j)$. Lưu ý rằng net_j có thể ảnh hưởng đến đầu ra mạng lưới (và do đó là E_d) chỉ thông qua các đơn vị trong $Downstream(j)$. Vì vậy, ta có thể viết:

$$\begin{aligned}\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k w_{kj} \frac{\partial o_j}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k w_{kj} o_j (1 - o_j)\end{aligned}$$

Biểu diễn lại các thành phần với δ_j dùng để chỉ $-\frac{\partial E_d}{\partial net_j}$ ta có:

$$\delta_j = o_j (1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj}$$

Và:

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

Trong trường hợp chỉ có một lớp ẩn và một lớp đầu ra $Downstream(j) = outputs$.

BACKPROPAGATION(*training_examples*, η , n_{in} , n_{out} , n_{hidden})

Each training example is a pair of the form (\vec{x}, \vec{t}) , where \vec{x} is the vector of network input values, and \vec{t} is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

The input from unit i into unit j is denoted x_{ji} , and the weight from unit i to unit j is denoted w_{ji} .

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.

- Initialize all network weights to small random numbers (e.g., between -.05 and .05).

- Until the termination condition is met, Do

 - For each (\vec{x}, \vec{t}) in *training_examples*, Do

Propagate the input forward through the network:

1. Input the instance \vec{x} to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit k , calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \quad (\text{T4.3})$$

3. For each hidden unit h , calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{kh} \delta_k \quad (\text{T4.4})$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \delta_j x_{ji} \quad (\text{T4.5})$$

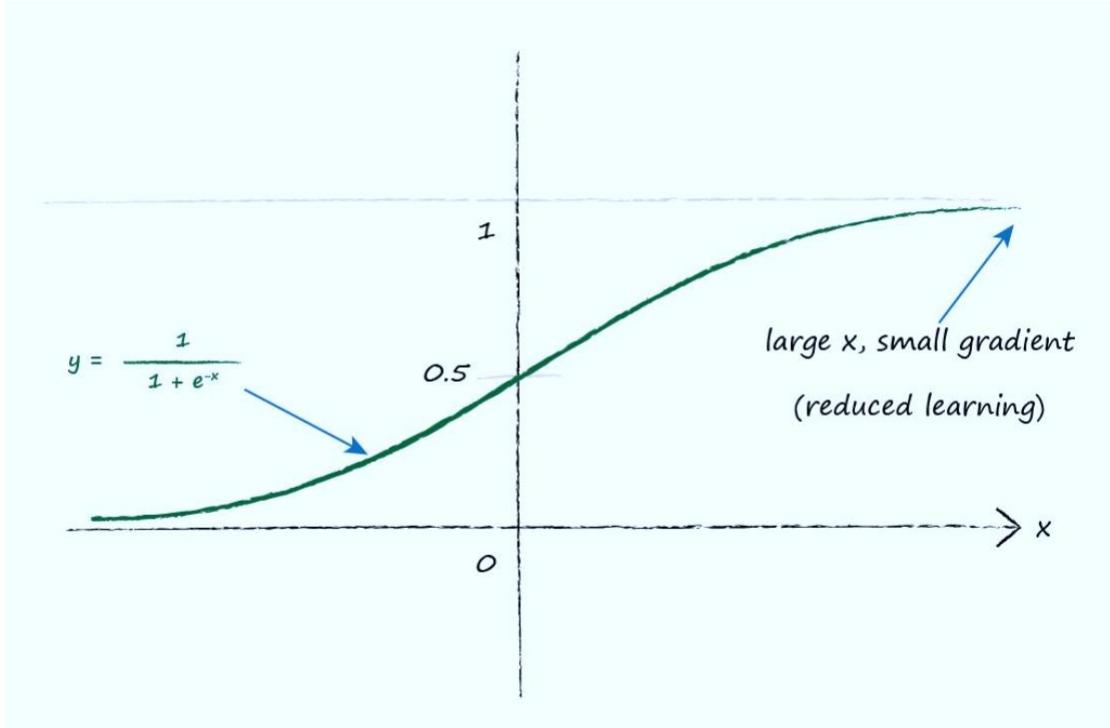
Bảng 4.2: Phiên bản incremental gradient descent với thuật toán Backpropagation cho mạng lưới thần kinh truyền thẳng với 2 lớp gồm lớp hidden và lớp output sử dụng đơn vị sigmoid.

1.3 Chuẩn hóa dữ liệu

1.3.1 Chuẩn hóa dữ liệu đầu vào

Trong thực tế, ta thường phải đối mặt với nhiều kiểu biến trong cùng một tập dữ liệu và một vấn đề xảy ra là phạm vi của các biến có thể khác nhau rất nhiều cũng như khoảng giá trị có phạm vi lớn. Việc sử dụng thang đo gốc có thể ảnh hưởng đến kết quả dự đoán của mạng lưới thần kinh.

Ví dụ trong đồ thị ở hình minh họa dưới đây về hàm kích hoạt Sigmoid. Ta thấy rằng nếu đầu vào mang giá trị lớn, hàm kích hoạt trở nên rất phẳng:



Một hàm kích hoạt rất phẳng có vấn đề vì chúng ta dùng phương pháp gradient descent để học các trọng số mới. Nhìn lại biểu thức về cập nhật trọng số, nó phụ thuộc vào gradient của hàm chúc năng. Một gradient siêu bé có nghĩa là chúng ta đã hạn chế khả năng học. Điều này được gọi là bão hòa mạng lưới thần kinh. Do vậy chúng ta nên cố gắng giữ cho giá trị đầu vào nhỏ.

Có khá nhiều cách được khuyến nghị để chuẩn hóa giá dữ liệu đầu vào, một trong số đó là chuẩn hóa **z-score**.

Chuẩn hóa z-score là quá trình chuẩn hóa tất cả giá trị trong tập dữ liệu sao cho trung bình của tất cả giá trị là 0 và độ lệch chuẩn là 1.

Công thức để biểu diễn chuẩn hóa z-score cho tất cả giá trị trong tập dữ liệu:

$$\text{Giá trị mới} = \frac{(x - \mu)}{\sigma}$$

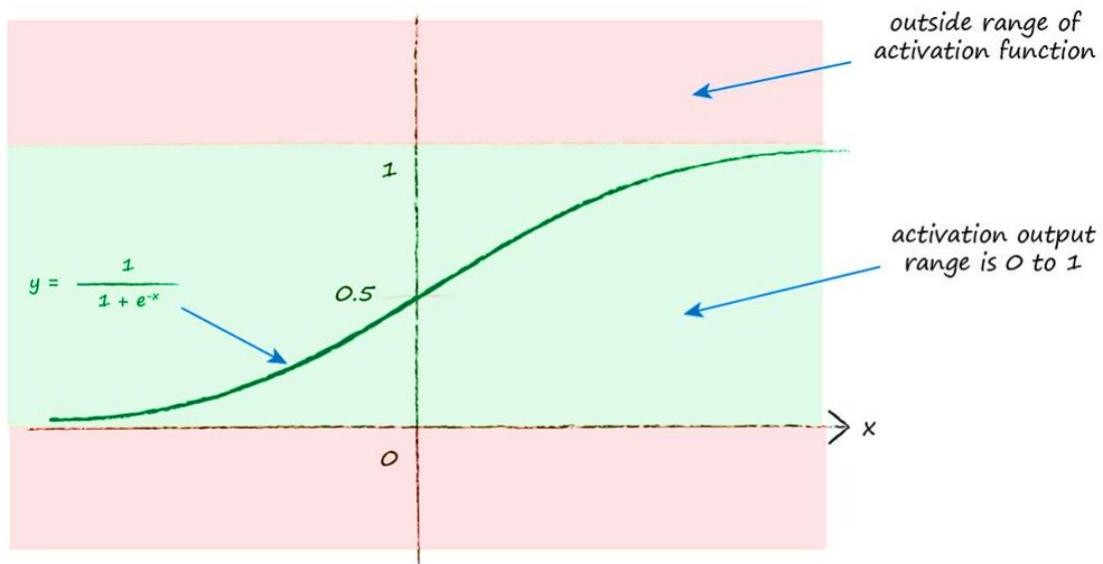
Trong đó:

- x : giá trị ban đầu
- μ : trung bình của dữ liệu
- σ : độ lệch chuẩn của dữ liệu

1.3.2 Chọn đầu ra mục tiêu

Đầu ra của mạng lưới thần kinh là giá trị lấy từ hàm kích hoạt của lớp cuối. Nếu chúng ta đang sử dụng một hàm kích hoạt sigmoid thì không thể tạo ra giá trị trên 1 hoặc dưới 0 thì sẽ thật ngớ ngẩn nếu cố gắng đặt các giá trị lớn hơn 1 hay nhỏ hơn 0 làm mục tiêu huấn luyện. Lưu ý rằng hàm sigmoid thậm chí không đạt đến 1 hay 0, nó chỉ tiệm cận đến các giá trị đó.

Hình minh họa sau đây làm rõ rằng các giá trị đầu ra lớn hơn 1 và nhỏ hơn 0 là không khả thi từ hàm kích hoạt sigmoid.



Nếu chúng ta đặt các giá trị mục tiêu trong các phạm vi nằm ngoài ví dụ như khoảng được biểu diễn bởi mũi tên trên cùng trên hình vẽ, thì quá trình huấn luyện mạng lưới sẽ đưa ra các trọng số lớn nhằm cố gắng tạo ra các đầu ra lớn hơn và lớn hơn mà hàm kích hoạt không bao giờ thực sự có thể tạo ra được.

Vì vậy, chúng ta nên thay đổi khoảng các giá trị mục tiêu của mình để phù hợp với các đầu ra có thể có từ hàm kích hoạt. Trong thực tế, chúng ta thường đặt giá trị mục tiêu trong khoảng từ 0.1 đến 0.9.

1.3.3 Khởi tạo ngẫu nhiên trọng số

Chúng ta nên tránh khởi tạo các trọng số ban đầu lớn vì chúng dẫn đến tình trạng bão hòa mà chúng ta nói đến ở trên và giảm khả năng học các trọng số tốt hơn.

Dù làm gì, chúng ta đừng đặt các trọng số ban đầu có cùng giá trị không đổi, đặc biệt là bằng 0. Điều đó rất tệ! Vì mỗi đơn vị trong mạng lưới sẽ nhận được cùng một giá trị đơn vị đầu vào và đầu ra của mỗi đơn vị đầu ra giống nhau. Nếu sau đó chúng ta tiến hành cập nhật các trọng số trong mạng lưới bằng phương pháp backpropagation, sai số được chia đều. Nhớ rằng sai số được chia theo tỷ lệ với trọng số. Điều đó sẽ dẫn đến cập nhật các trọng số như nhau lại dẫn đến một tập hợp các trọng số có giá trị như nhau khác. Sự đối xứng này là xấu bởi mạng lưới được huấn luyện đúng cách sẽ có trọng số không bằng nhau (gần như hầu hết trong mọi vấn đề) và bạn sẽ không bao giờ đạt được điều đó.

Trọng số bằng 0 thậm chí còn tệ hơn vì chúng làm xóa bỏ đầu vào của các đơn vị. Hàm cập nhật trọng số, phụ thuộc vào các đầu vào đến, bằng không. Điều đó giết chết hoàn toàn khả năng cập nhật trọng số.

Ở phần trên chuẩn hóa dữ liệu đầu vào chúng ta đã đưa dữ liệu về dạng phân bố chuẩn $N(0,1)$ sử dụng phương pháp Marsaglia polar (Phương pháp lấy mẫu số giả ngẫu nhiên để tạo ra cặp biến ngẫu nhiên độc lập).

Phương pháp Marsaglia hoạt động bằng cách chọn ngẫu nhiên các điểm (x,y) với $-1 < x < 1$ và $-1 < y < 1$ cho đến khi $0 < s = x^2 + y^2 < 1$ và trả về cặp biến ngẫu nhiên chuẩn theo yêu cầu là:

$$x \sqrt{\frac{-2 \ln(s)}{s}}, \quad y \sqrt{\frac{-2 \ln(s)}{s}}$$

Chương 2. Áp dụng tính toán song song

Với đề tài này, ta sẽ sử dụng OpenMP (Open Multi Processing) - một môi trường lập trình song song phù hợp để viết các chương trình song song chạy đa luồng trên hệ thống bộ nhớ dùng chung.

2.1 Tổng quan về OpenMP

OpenMP ra đời với mục tiêu cung cấp một chuẩn chung cho rất nhiều kiến trúc và nền tảng phần cứng. Nó là thư viện mã nguồn mở cung cấp rất nhiều các hàm, các chỉ thị giúp cho người lập trình linh động và dễ dàng phát triển ứng dụng song song của mình.

OpenMP không phải là một ngôn ngữ lập trình mới, nó là phần mở rộng cho các ngôn ngữ lập trình khác thường là Fortran hoặc C/C++. Trong đề tài này, ta sẽ sử dụng ngôn ngữ C.

Giao diện lập trình ứng dụng của OpenMP là tập hợp của:

- **Chỉ thị biên dịch (Compiler directive):** Chỉ thị biên dịch là bắt buộc có đối với mỗi chương trình ứng dụng song song. Chỉ thị biên dịch sẽ báo cho trình biên dịch biết sự bắt đầu của khối mã thực hiện song song.

```
#pragma omp parallel [clause [, ] clause] ...
structured-block
```

Khi chạy đến chỉ thị **pragma omp parallel**, một nhóm luồng sẽ được hình thành và khối cấu trúc được thực thi bởi tất cả các luồng trong nhóm.

- **Các hàm phương thức hỗ trợ (Supporting functions):**

omp_get_num_threads(): Trả về giá trị là tổng số luồng được thực thi trong vùng song song.

omp_get_thread_num(): Trả về chỉ số của luồng hiện tại đang thực thi đoạn mã trong vùng song song.

- **Các biến môi trường (Shell variables):**

omp_num_threads(): Đặt số lượng luồng để sử dụng trong vùng song song.

omp_thread_limit(): Giới hạn số lượng luồng một chương trình sử dụng.

2.2 Các thành phần được sử dụng của OpenMP

2.2.1 Mệnh đề: for num_threads()

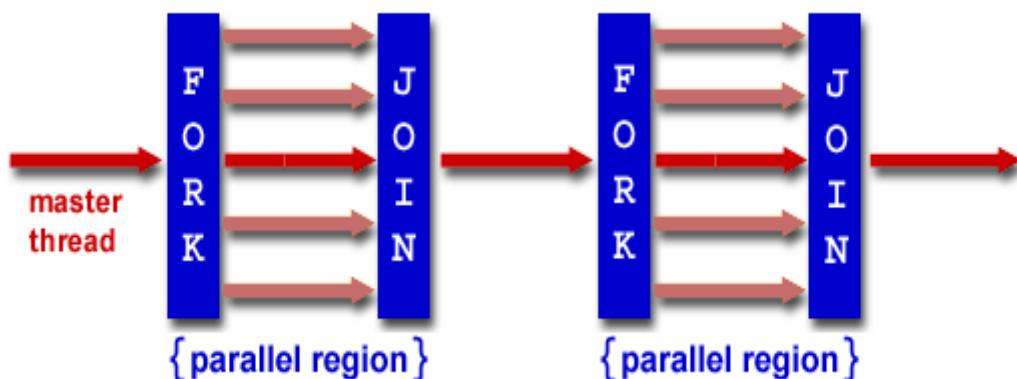
```
#pragma omp parallel for num_threads(NUM_THREADS)
```

Khi chạy đến chỉ thị biên dịch trên, một nhóm luồng sẽ được hình thành với số luồng là giá trị của biến NUM_THREADS.

2.2.2 Vòng lặp song song

```
#pragma omp for [clause [[,] clause] ...]  
for-loops
```

Khi chạy đến chỉ thị trên, vòng lặp for sẽ được xử lý bởi một nhóm luồng trong vùng song song, mỗi phần của vòng lặp sẽ được chia cho các luồng khác nhau để xử lý. Khi các thành phần được thực thi xong, các luồng trong nhóm sẽ được đồng bộ hóa ở rào cản ngầm (implicit barrier) ở cuối vòng lặp. Nếu muốn loại bỏ rào cản ngầm ta sử dụng mệnh đề nowait.



Mô hình FORK – JOIN

Trong mô hình này, tất cả các chương trình khi bắt đầu chạy sẽ được xử lý tuần tự bởi luồng chủ (Master Thread) cho đến khi bắt gặp vùng song song.

FORK: Luồng chủ sẽ tạo ra các luồng thực hiện song song. Các đoạn mã song song trong chương trình sẽ được các luồng này thực thi một cách đồng thời.

JOIN: Khi các luồng thực thi các đoạn mã trong vùng song song kết thúc, chúng sẽ được đồng bộ sau đó công việc lại được thực thi bởi luồng chủ.

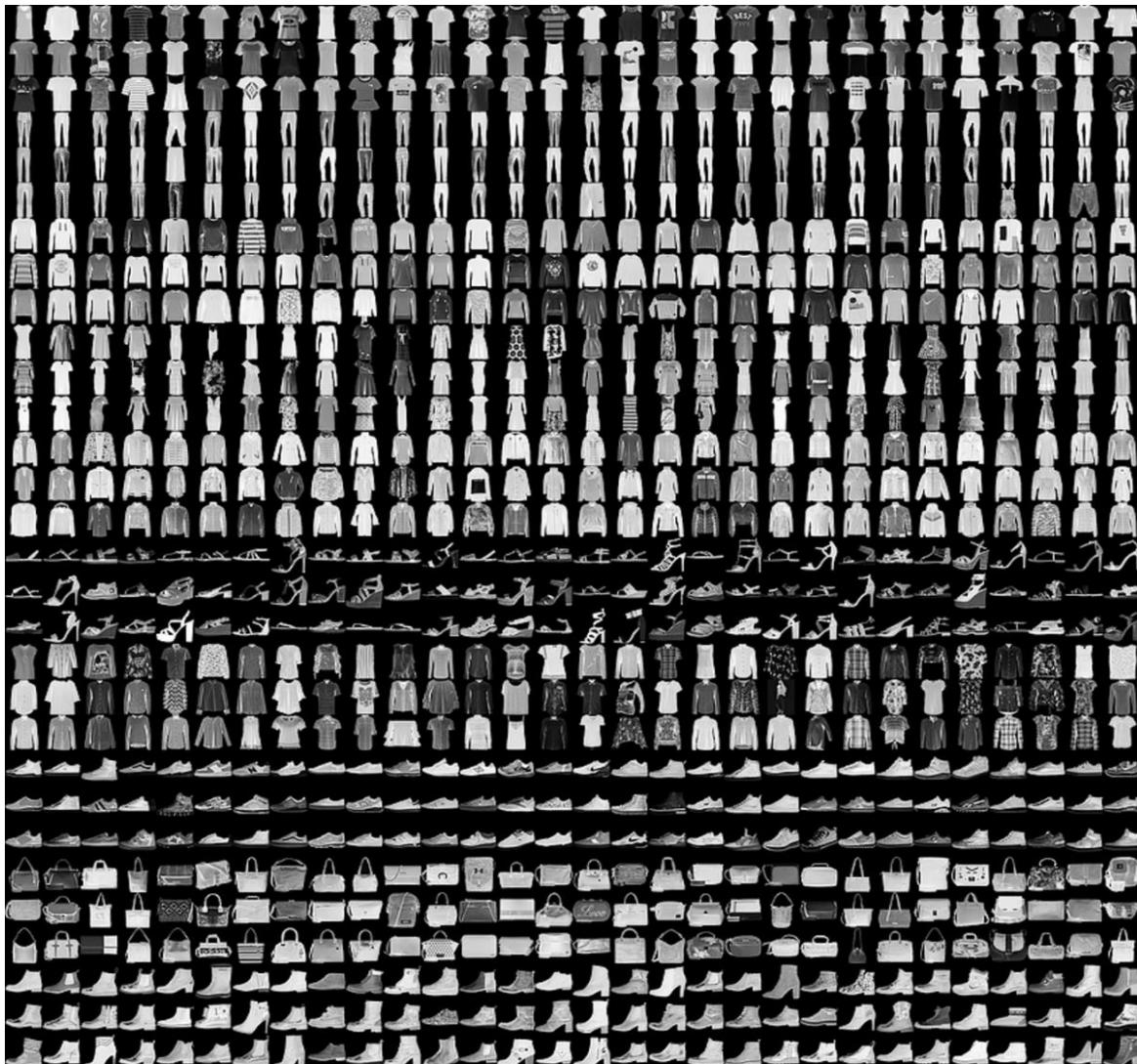
Mục đích của việc sử dụng nhiều luồng cùng một lúc là để có thể thực hiện xử lý nhiều phép tính hơn trong cùng một khoảng thời gian dẫn đến tối ưu thời gian chạy chương trình. Tuy nhiên, sẽ có lúc chạy nhiều luồng không được tối ưu hoặc chậm hơn do công việc trên các luồng được phân chia chưa đồng đều. Ngoài ra, nếu kết quả của luồng này ảnh hưởng đến luồng kia thì ta cần cẩn trọng trong việc phân tích và xử lý sắp xếp sao cho có được kết quả đầu ra mong muốn.

Ta sử dụng các vòng lặp song song hỗ trợ việc tính toán vecto cho từng lớp trong mạng lưới thần kinh do kết quả mỗi vecto trong từng lớp của một phần của vòng lặp là độc lập với kết quả của phần khác nên ta có thể thực hiện tính toán chúng song song mà không sợ ảnh hưởng đến kết quả đầu ra.

Chương 3. Trực quan hóa dữ liệu

3.1 Bộ dữ liệu Fashion MNIST

3.1.1 Sơ lược về bộ dữ liệu Fashion MNIST



Fashion MNIST là bộ dữ liệu gồm các hình ảnh thời trang thuộc về công ty thương mại điện tử Zalando - bao gồm tập huấn luyện với 60.000 ví dụ và tập kiểm tra với 10.000 ví dụ. Mỗi ví dụ là một hình ảnh thang độ xám 28x28, được liên kết với nhãn từ 10 lớp. Zalando tạo ra bộ dữ liệu Fashion MNIST nhằm đóng vai trò thay thế trực tiếp cho bộ dữ liệu MNIST ban đầu về chữ viết tay để đo điểm chuẩn cho các thuật toán máy học. Nó chia sẻ cùng kích thước hình ảnh và cấu trúc phân tách huấn luyện và kiểm tra.

Bộ dữ liệu MNIST (Modified National Institute of Standards and Technology) về chữ viết tay có thể nói rằng là bộ dữ liệu triển khai “Hello World” cho học máy và bộ dữ liệu này được sử dụng làm tiêu chuẩn học máy trên toàn thế giới. Đây là một bộ dữ liệu cực kỳ tốt cho những người muốn thử các kỹ thuật học máy và phương pháp nhận dạng trên dữ liệu trong thế

giới thực trong khi dành tối thiểu thời gian và công sức cho việc định dạng và tiền xử lý dữ liệu. Tính đơn giản và dễ sử dụng của nó là những gì làm cho bộ dữ liệu này được sử dụng rộng rãi và hiệu sâu sắc.

Lý do ta dùng bộ dữ liệu Fashion-MNIST thay cho MNIST như ở lời mở đầu là do ta muốn nhận biết những hình ảnh phức tạp hơn để đánh giá chương trình của ta khắt khe hơn.

3.1.2 Nội dung của bộ dữ liệu Fashion MNIST

Như đã nói ở trên, tập dữ liệu Fashion MNIST được chia thành hai tập dữ liệu là huấn luyện(training) với 60000 hình ảnh và tập dữ liệu kiểm tra(testing) với 10000 hình ảnh. Vì hai tập dữ liệu này đều có cùng cấu trúc và chỉ khác nhau về số lượng dữ liệu nên chúng ta sẽ sử dụng một trong hai tập dữ liệu để giới thiệu. Ở đây chúng ta sẽ sử dụng tập dữ liệu huấn luyện.

Mỗi hình ảnh có chiều cao 28 pixels và chiều rộng 28 pixels, tổng cộng là 784 pixels. Mỗi pixel có một giá trị pixel duy nhất được liên kết với nó, cho biết độ sáng hoặc độ tối của pixel đó, với các số cao hơn có nghĩa là tối hơn. Giá trị pixel này là một số nguyên từ 0 đến 255.

Tập dữ liệu này bao gồm 785 cột trong đó:

- Cột đầu tiên: Ứng với các nhãn (label) có giá trị từ 0-9.
- 784 cột còn lại: Chứa các giá trị pixel các hình ảnh được liên kết.

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel1775	pixel1776	pixel1777	pixel1778	pixel1779	pixel1780	pixel1781	pixel1782	pixel1783	pixel1784
0	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	6	0	0	0	0	0	0	0	5	0	...	0	0	0	30	43	0	0	0	0	0
3	0	0	0	0	1	2	0	0	0	0	...	3	0	0	0	0	1	0	0	0	0
4	3	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

Hình ảnh tập dữ liệu Fashion-MNIST training

Các giá trị cột nhãn (label) tương ứng với các mặt hàng thời trang sau:

- 0: T-shirt/top (Áo thun/áo ba lỗ)
- 1: Trouser (Quần dài)
- 2: Pullover (Áo dài tay không có mũ trùm đầu)
- 3: Dress (Váy)
- 4: Coat (Áo choàng ngoài)

- 5: Sandal (Dép)
- 6: Shirt (Áo sơ mi)
- 7: Sneaker (Giày thể thao)
- 8: Bag (Túi)
- 9: Ankle boot (Giày cao cổ)

Ví dụ:

- Ở hàng số 0 (hàng đầu tiên) giá trị cột label bằng 2 đại diện cho Pullover
- Ở hàng số 2 giá trị cột label là 9 đại diện cho Ankle boot.

Chúng ta cũng có thể biết được số lượng hình ảnh của mỗi nhãn thời trang trong tập dữ liệu:

```
[ ] data['label'].value_counts()
2    6000
9    6000
6    6000
0    6000
3    6000
4    6000
5    6000
8    6000
7    6000
1    6000
Name: label, dtype: int64
```

Số lượng dữ liệu (hình ảnh) tương ứng với mỗi nhãn thời trang

Như chúng ta cũng thấy ở hình trên thì mỗi giá trị nhãn (label) sẽ có 6000 hình ảnh, tổng sẽ là $6000 * 10 = 60000$ hình ảnh đúng bằng số hình ảnh trong tập dữ liệu. Điều này cho chúng ta thấy được đây là một tập dữ liệu có sự cân bằng về mặt phân phối lớp dữ liệu.

3.2 Trực quan hóa bộ dữ liệu thời trang Fashion MNIST

Trước khi trực quan hóa tập dữ liệu thì chúng ta phải chuẩn hóa tập dữ liệu đó. Các giá trị cột 784 pixelXXX trong mỗi hàng chứa thông tin độ sáng cho từng pixel trong hình ảnh thang độ xám 28x28. Độ sáng của pixel được lưu dưới dạng số nguyên 8 bit, nằm trong khoảng từ 0 (thường biểu thị màu đen) đến 255 (màu trắng). Các giá trị ở giữa đại diện cho các sắc thái khác nhau của màu xám. Do đó, để chuẩn hóa các giá trị, chúng ta cần chia mỗi ô pixelXXX cho 255.

	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10	...	pixel1775	pixel1776	pixel1777	pixel1778	pixel1779	pixel1780	pixel1781	pixel1782	pixel1783	pixel1784
0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	0.0	...	0.000000	0.0	0.0	0.000000	0.000000	0.000000	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	0.0	0.0	...	0.000000	0.0	0.0	0.000000	0.000000	0.000000	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.019608	0.0	0.0	...	0.000000	0.0	0.0	0.117647	0.168627	0.000000	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.003922	0.007843	0.0	0.0	0.000000	0.0	0.0	...	0.011765	0.0	0.0	0.000000	0.000000	0.003922	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	0.0	0.0	...	0.000000	0.0	0.0	0.000000	0.000000	0.000000	0.0	0.0	0.0	0.0

5 rows x 784 columns

Tập dữ liệu sau khi đã được chuẩn hóa

3.2.1 Trực quan hóa hình ảnh huấn luyện

Từ tập dữ liệu ban đầu chúng ta tách ra thành hai khung dữ liệu:

- Input_data: Chứa các giá trị pixel.

	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10	...	pixel1775	pixel1776	pixel1777	pixel1778	pixel1779	pixel1780	pixel1781	pixel1782	pixel1783	pixel1784
0	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	0.0	0.0	...	0.000000	0.0	0.0	0.000000	0.000000	0.000000	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	0.0	0.0	...	0.000000	0.0	0.0	0.000000	0.000000	0.000000	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.019608	0.0	0.0	...	0.000000	0.0	0.0	0.117647	0.168627	0.000000	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.003922	0.007843	0.0	0.0	0.000000	0.0	0.0	...	0.011765	0.0	0.0	0.000000	0.000000	0.003922	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.000000	0.0	0.0	...	0.000000	0.0	0.0	0.000000	0.000000	0.000000	0.0	0.0	0.0	0.0

5 rows x 784 columns

- Targer: Chứa các giá trị label.

label	
0	2
1	9
2	6
3	0
4	3

Trên thực tế, điều quan trọng là phải kiểm tra tập dữ liệu theo cách thủ công. Và nó là cần thiết để trực quan hóa dữ liệu mẫu. Trong phần này, ta sử dụng thư viện matplotlib để trực quan hóa 30 hình ảnh huấn luyện (training) đầu tiên trong khung dữ liệu input_data với các nhãn (label) tương ứng trong ma trận 6x5.

```

label = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# Set the figure size
plt.figure(figsize=(10,10))

# Show only the first 30 pictures
for i in range(30):
    plt.subplot(6,5, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(np.array(input_data.iloc[i, :]).reshape(28,28), cmap=plt.cm.binary)
    plt.xlabel(label[target.label.iloc[i]])

```



30 hình ảnh đầu tiên của tập dữ liệu huấn luyện (training)

Chương 4. Xây dựng, thực thi chương trình

4.1 Tổng quan chương trình

Xây dựng mạng lưới thần kinh bao gồm 3 lớp: lớp đầu vào, lớp ẩn và lớp đầu ra.

Sử dụng tập dữ liệu: Fashion-MNIST

Sử dụng hàm kích hoạt: Sigmoid

Sử dụng hàm tính sai số: MSE

Tốc độ học: 0.05

Ngôn ngữ lập trình sử dụng: C

4.2 Quy trình thực hiện

Bước 1: Định nghĩa cấu trúc mô hình, số lượng của các đặc trưng đầu vào, thư viện hỗ trợ

```
// DEFINITIONS
#define NUM_THRDS 4
#define NL1 100          // 1st layer size
#define NL2 10          // output layer size
#define NINPUT 784       //input size
#define NTRAIN 60000      //training set size
#define NTEST 10000       //testing set size
#define ITERATIONS 1     //number of epochs
#define ALPHA (double)0.05 //learning rate
// ****
// INCLUDES
#include "extra_functions.c"
#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
// ****
```

Bước 2: Khởi tạo các biến.

```
// GLOBAL VARS
double WL1[NL1][NINPUT + 1];
double WL2[NL2][NL1 + 1];
// layer internal states
double DL1[NL1];
double DL2[NL2];
// layer outputs
double OL1[NL1];
double OL2[NL2];
// layer deltas
double delta2[NL2];
double delta1[NL1];
//data
double data_train[NTRAIN][NINPUT];
double data_test[NTEST][NINPUT];
int class_train[NTRAIN];
int class_test[NTEST];
double input[NINPUT];
// ****
```

Bước 3: Đọc số liệu từ tập dữ liệu.

```
// Used for reading the MNIST fashion dataset.
int readfile(char *filepath, int *Class, double Data[][NINPUT], int numVectors) {
    FILE *fp;
    char B[20001], *p;
    int j;

    fp = fopen(filepath, "r");
    if (fp == NULL)
        return -1;
    if (fgets(B, 2000, fp) != B)
        return -2;
    for (j = 0; j < numVectors; j++) {
        if (fgets(B, 2000, fp) != B)
            return -2;
        p = strtok(B, ",");
        if (p == NULL)
            return -3;
        Class[j] = atoi(p);

        for (int i = 0; i < NINPUT; i++) {
            p = strtok(NULL, ",\n");
            Data[j][i] = atof(p);
        }
    }
    printf("Loaded %d examples\n", j);

    fclose(fp);
    return 0;
}
```

Bước 4: Chuẩn hóa dữ liệu theo phân bố chuẩn tắc.

```
// Normalizes the input data into normal distribution N(0,1)
void normalizeData(double in[][NINPUT], int inSize){
    double average[NINPUT] = {0};
    double var[NINPUT] = {0};
    #pragma omp parallel for num_threads(NUM_THREAD)
    for (int i = 0; i < NINPUT; i++)
    {
        for (int j = 0; j < inSize; j++)//calculate mean
        {
            average[i] += in[j][i];
        }
        average[i] /= inSize;
        double register mean = average[i];
        for (int j = 0; j < inSize; j++)//calculate variance
        {
            var[i] += (in[j][i] - mean)*(in[j][i] - mean);
        }
        var[i] /= inSize-1;
    }
    #pragma omp parallel for num_threads(NUM_THREAD)
    for (int i = 0; i < NINPUT; i++)
    {
        double register mean = average[i];
        double register stddev = sqrt(var[i]);
        for (int j = 0; j < inSize; j++)
        {
            in[j][i] -= mean;
            in[j][i] /= stddev;
        }
    }
}
```

Bước 5: Triển khai hàm lấy mẫu ngẫu nhiên theo phân bố chuẩn tắc $N(0,1)$ sử dụng phương pháp Marsaglia polar.

```
// Samples a standard normal distribution [~N(0,1)]
// using the Marsaglia polar method: https://en.wikipedia.org/wiki/Marsaglia\_polar\_method
double gaussrand() {
    static double V1, V2, S;
    static int phase = 0;
    double X;

    if (phase == 0) {
        do {
            double U1 = (double)rand() / RAND_MAX;
            double U2 = (double)rand() / RAND_MAX;

            V1 = 2 * U1 - 1;
            V2 = 2 * U2 - 1;
            S = V1 * V1 + V2 * V2;
        } while (S >= 1 || S == 0);

        X = V1 * sqrt(-2 * log(S) / S);
    }
    else
        X = V2 * sqrt(-2 * log(S) / S);

    phase = 1 - phase;

    return X;
}
```

Bước 6: Khởi tạo các trọng số khớp thần kinh với giá trị từ phân phối chuẩn.

```
// initializes neuron synapses' weights with values from a standard normal distribution.
void initVecs() {
    double register stddev1 = 1;
    double register stddev2 = 1;
    for (int i = 0; i < NL1; i++) {
        for (int j = 0; j < NINPUT + 1; j++) {
            WL1[i][j] = gaussrand() * stddev1;
        }
    }
    for (int i = 0; i < NL2; i++) {
        for (int j = 0; j < NL1 + 1; j++) {
            WL2[i][j] = gaussrand() * stddev2;
        }
    }
}
```

Bước 7: Triển khai hàm Sigmoid.

```
// Implements the logistic sigmoid function.
double logistic(double x) {
    return 1.0 / (1.0 + exp(-x));
}
```

Bước 8: Triển khai phần truyền của mạng lưới thần kinh với hàm kích hoạt là Sigmoid.

```
// Implements the feedforward part of the Neural Network using the vector "in" as input.
void activateNN(double *in){
    // layer1
    #pragma omp parallel for num_threads(NUM_THREADS)
    for (int i = 0; i < NL1; i++)
    {
        double register sum = 0;
        for (int j = 0; j < NINPUT; j++)
        {
            sum += WL1[i][j] * in[j];
        }
        sum += WL1[i][NINPUT]; //add bias neuron weight
        DL1[i] = sum;
        OL1[i] = logistic(sum);
    }
    // layer2
    #pragma omp parallel for num_threads(NUM_THREADS)
    for (int i = 0; i < NL2; i++)
    {
        double register sum = 0;
        for (int j = 0; j < NL1; j++)
        {
            sum += WL2[i][j] * OL1[j];
        }
        sum += WL2[i][NL1]; //add bias neuron weight
        DL2[i] = sum;
        OL2[i] = logistic(sum);
    }
}
```

Bước 9: Triển khai thuật toán Backpropagation sử dụng Gradient Descent.

```
void trainNN(double *in,double *desired){

    // Calculate Neural Network outputs
    activateNN(in);
    // Output layer deltas
    #pragma omp parallel num_threads(NUM_THREAD)
    {
        #pragma omp for
        for (int i = 0; i < NL2; i++)
        {
            delta2[i] = (OL2[i]-desired[i])*OL2[i]*(1-OL2[i]);
        }
        // Layer 1 Deltas
        #pragma omp for
        for (int i = 0; i < NL1; i++)
        {
            double register sum = 0;
            for (int j = 0; j < NL2; j++)
            {
                sum += WL2[j][i] * delta2[j];
            }
            double register Oi = OL1[i];
            delta1[i] = sum * Oi * (1-Oi);
        }
    }

    #pragma omp parallel num_threads(NUM_THREAD)
    {
        // update weights of layer 2
        #pragma omp for nowait
        for (int i = 0; i < NL2; i++)
        {
            for (int j = 0; j < NL1; j++)
            {
                WL2[i][j] -= ALPHA * OL1[j] * delta2[i];
            }
            WL2[i][NL1] -= ALPHA * delta2[i];//update bias neuron weight
        }
        // update weights of layer 1
        #pragma omp for
        for (int i = 0; i < NL1; i++)
        {
            for (int j = 0; j < NINPUT; j++)
            {
                WL1[i][j] -= ALPHA * in[j] * delta1[i];
            }
            WL1[i][NINPUT] -= ALPHA * delta1[i];//update bias neuron weight
        }
    }
}
```

Bước 10: Triển khai hàm đánh giá mạng lưới dự đoán đầu vào thuộc lớp nào bằng cách tìm argmax của lớp đầu ra.

```
// Evaluates which class the network predicts that the input belongs to  
// by finding the argmax of the output layer.  
// Afterwards it updates the confusion matrix with the prediction.  
void evaluate(int inputClass,double confMatrix[NL2][NL2])  
{  
    int maxIndex = 0;  
    double maxVal = 0;  
    for (int i = 0; i < NL2; i++)  
    {  
        if (maxVal<OL2[i])  
        {  
            maxVal = OL2[i];  
            maxIndex = i;  
        }  
    }  
    //Edge case where both the correct output layer and another one have the same output value, we consider that a correct classification.  
    if (maxVal==OL2[inputClass])  
    {  
        maxIndex = inputClass;  
    }  
    confMatrix[maxIndex][inputClass]++;  
}
```

Bước 11: Triển khai hàm main để đưa ra kết quả.

```
int main() {  
    double desiredOut[NL2]={0};  
    double confusionMatrixTrain[NL2][NL2]= {0};  
    double confusionMatrixTest[NL2][NL2]= {0};  
    double start = omp_get_wtime();  
    readfile("D:/Download/archive/fashion-mnist_train.csv",class_train,data_train,NTRAIN);  
    readfile("D:/Download/archive/fashion-mnist_test.csv",class_test,data_test,NTEST);  
    normalizeData(data_test,NTEST);  
    normalizeData(data_train,NTRAIN);  
    initVecs();//initialise weights  
    for (int i = 0; i < NL2; i++)//initialise desired vector values  
    {  
        desiredOut[i] = 0.1;  
    }  
    int register tmp = 0;  
    for (int i = 0; i < NTRAIN*ITERATIONS; i++)//train the nn  
    {  
        tmp = rand()%NTRAIN;  
        desiredOut[class_train[tmp]] = 0.9;  
        trainNN(data_train[tmp],desiredOut);  
        desiredOut[class_train[tmp]] = 0.1;  
    }  
    printf("TRAINING FINISHED!\n\n");
```

```

        for (int i = 0; i < NTRAIN; i++)//test with training set
        {
            activateNN(data_train[i]);
            evaluate(class_train[i],confusionMatrixTrain);
        }

        for (int i = 0; i < NTEST; i++)//test with testing set
        {
            activateNN(data_test[i]);
            evaluate(class_test[i],confusionMatrixTest);
        }

        double register testCorrect = 0;
        double register trainCorrect = 0;
        for (int i = 0; i < NL2; i++)
        {
            testCorrect += confusionMatrixTest[i][i];
            trainCorrect += confusionMatrixTrain[i][i];
        }

        double end = omp_get_wtime();

        double totalCorrect = testCorrect + trainCorrect;
        testCorrect /= (double)NTEST;
        trainCorrect /= (double)NTRAIN;
        totalCorrect /= ((double)NTEST+(double)NTRAIN);

        printf("TRAINING SAMPLES CONFUSION MATRIX:\n");
        printTable(confusionMatrixTrain);
        printf("TESTING SAMPLES CONFUSION MATRIX:\n");
        printTable(confusionMatrixTest);
        printf("Correct rate in training samples: %0.3f\n",trainCorrect);
        printf("Correct rate in testing samples: %0.3f\n",testCorrect);
        printf("Overall hit rate: %0.3f\n",totalCorrect);
        printf("Learning rate = %0.4f\n",ALPHA);
        printf("EPOCHS = %d\n",(int)ITERATIONS);
        printf("Time: \t %f \n", ((end-start)));
        return 0;
    }
}

```

4.3 Thực thi chương trình

Thực thi trên: CPU core I7 thế hệ 11, hệ điều hành Window.

Chạy chương trình trên: 1, 4, 8 luồng và số lần truyền dữ liệu là 1, 3, 5, 10, 100 để đưa ra thời gian chênh lệch giữa chạy trên đơn luồng và đa luồng và độ chính xác khi tăng số lần truyền dữ liệu.

Với 1 lần truyền dữ liệu (1 epochs)

Ma trận lỗi cho mẫu dữ liệu huấn luyện:

4589.00	42.00	152.00	230.00	70.00	17.00	1237.00	2.00	69.00	8.00
57.00	5514.00	24.00	73.00	56.00	4.00	35.00	0.00	1.00	2.00
152.00	82.00	3733.00	73.00	654.00	26.00	874.00	1.00	67.00	4.00
730.00	282.00	162.00	5287.00	562.00	28.00	599.00	13.00	122.00	13.00
72.00	19.00	1107.00	135.00	3769.00	3.00	692.00	0.00	63.00	2.00
97.00	15.00	145.00	47.00	99.00	4808.00	209.00	417.00	117.00	201.00
132.00	21.00	530.00	97.00	659.00	46.00	2035.00	3.00	59.00	21.00
2.00	6.00	4.00	7.00	2.00	692.00	8.00	5195.00	75.00	412.00
155.00	14.00	134.00	46.00	127.00	133.00	290.00	46.00	5378.00	38.00
14.00	5.00	9.00	5.00	2.00	243.00	21.00	323.00	49.00	5299.00

Ma trận lỗi cho mẫu dữ liệu kiểm tra:

753.00	10.00	36.00	38.00	9.00	6.00	206.00	0.00	11.00	0.00
8.00	918.00	4.00	20.00	11.00	1.00	11.00	0.00	0.00	0.00
28.00	20.00	601.00	18.00	85.00	3.00	136.00	0.00	12.00	2.00
121.00	40.00	22.00	873.00	94.00	4.00	107.00	4.00	23.00	1.00
12.00	5.00	198.00	21.00	650.00	2.00	104.00	0.00	15.00	1.00
17.00	2.00	26.00	10.00	16.00	795.00	45.00	69.00	15.00	46.00
19.00	3.00	88.00	12.00	119.00	5.00	333.00	0.00	17.00	4.00
2.00	1.00	0.00	2.00	0.00	118.00	1.00	858.00	10.00	78.00
37.00	1.00	22.00	5.00	16.00	23.00	49.00	3.00	885.00	7.00
3.00	0.00	3.00	1.00	0.00	43.00	8.00	66.00	12.00	861.00

Độ chính xác trong mẫu dữ liệu huấn luyện (76%): Được tính bằng cách lấy tổng các phần tử trên đường chéo chính của ma trận lỗi chia cho số lượng ví dụ huấn luyện là 60000, lý do để hai chữ số thập phân sau dấu phẩy là để kiểm tra xem kết quả đưa ra về số lượng dự đoán đúng và sai tương ứng với từng loại thời trang có phải số nguyên không nếu không phải, chương trình đang gặp lỗi.

Độ chính xác trong mẫu dữ liệu kiểm tra (75,3%): Được tính bằng cách lấy tổng các phần tử trên đường chéo chính của ma trận lỗi chia cho số lượng ví dụ kiểm tra là 10000.

Độ chính xác chung (75,9%): Được tính bằng cách lấy tổng của tổng các phần tử trên đường chéo chính của ma trận lỗi huấn luyện và tổng các phần tử trên đường chéo chính của ma trận lỗi kiểm tra chia cho tổng số lượng ví dụ của huấn luyện và kiểm tra là 70000.

Thời gian chạy trên

1 luồng: 40.857s

4 luồng: 25.823s

8 luồng: 23.706s

Với 3 lần truyền dữ liệu (3 epochs), thời gian chạy

1 luồng: 91.465s

4 luồng: 56.777s

8 luồng: 55.706s

Ma trận lỗi cho mẫu dữ liệu huấn luyện:

4990.00	48.00	121.00	277.00	48.00	22.00	1104.0	5.00	62.00	7.00
18.00	5665.00	14.00	53.00	25.00	6.00	20.00	0.00	2.00	1.00
116.00	50.00	4435.00	81.00	694.00	16.00	779.00	0.00	50.00	4.00
368.00	160.00	95.00	5208.00	307.00	49.00	317.00	18.00	55.00	7.00
38.00	15.00	710.00	182.00	4173.00	13.00	516.00	1.00	36.00	10.00
9.00	0.00	12.00	6.00	7.00	4923.00	18.00	174.00	45.00	113.00
349.00	50.00	532.00	146.00	682.00	87.00	3063.00	7.00	114.00	17.00
2.00	4.00	0.00	4.00	0.00	550.00	0.00	5462.00	28.00	289.00
102.00	7.00	80.00	38.00	63.00	136.00	178.00	47.00	5598.00	33.00
8.00	1.00	1.00	5.00	1.00	198.00	5.00	286.00	10.00	5519.00

Ma trận lỗi cho mẫu dữ liệu kiểm tra:

808.00	11.00	24.00	44.00	3.00	9.00	209.00	0.00	9.00	4.00
3.00	943.00	2.00	16.00	3.00	2.00	4.00	0.00	1.00	0.00
32.00	11.00	728.00	15.00	103.00	7.00	120.00	0.00	8.00	2.00
70.00	20.00	19.00	863.00	48.00	10.00	56.00	2.00	10.00	1.00
5.00	5.00	122.00	26.00	718.00	3.00	89.00	0.00	9.00	1.00
1.00	1.00	2.00	3.00	0.00	804.00	4.00	34.00	7.00	28.00
54.00	8.00	86.00	24.00	115.00	11.00	487.00	0.00	29.00	5.00
0.00	1.00	0.00	3.00	0.00	97.00	0.00	886.00	6.00	57.00
26.00	0.00	17.00	5.00	9.00	23.00	28.00	5.00	920.00	8.00
1.00	0.00	0.00	1.00	1.00	34.00	3.00	73.00	1.00	894.00

Độ chính xác trong mẫu dữ liệu huấn luyện: 81,7%

Độ chính xác trong mẫu dữ liệu kiểm tra: 80,5%

Độ chính xác chung: 81.6%

Với 5 lần truyền dữ liệu (5 epochs), thời gian chạy

1 luồng: 142.560s

4 luồng: 88.499s

8 luồng: 89.171s

Ma trận lỗi cho mẫu dữ liệu huấn luyện:

4727.00	31.00	59.00	181.00	24.00	15.00	837.00	3.00	19.00	3.00
29.00	5728.00	18.00	57.00	23.00	3.00	26.00	0.00	0.00	1.00
77.00	25.00	4435.00	75.00	531.00	10.00	678.00	0.00	33.00	3.00
382.00	138.00	69.00	5293.00	304.00	15.00	294.00	1.00	39.00	0.00
42.00	16.00	826.00	180.00	4628.00	25.00	695.00	2.00	62.00	4.00
12.00	2.00	18.00	12.00	11.00	5402.00	13.00	248.00	79.00	113.00
599.00	46.00	503.00	146.00	427.00	15.00	3297.00	2.00	105.00	5.00
3.00	1.00	0.00	1.00	0.00	277.00	1.00	5401.00	23.00	220.00
116.00	9.00	72.00	46.00	51.00	48.00	155.00	26.00	5616.00	6.00
13.00	4.00	0.00	9.00	1.00	190.00	4.00	317.00	24.00	5645.00

Ma trận lỗi cho mẫu dữ liệu kiểm tra:

771.00	5.00	7.00	26.00	1.00	7.00	166.00	0.00	3.00	0.00
7.00	950.00	3.00	17.00	3.00	1.00	7.00	0.00	1.00	0.00
16.00	5.00	727.00	18.00	90.00	4.00	102.00	1.00	6.00	0.00
70.00	21.00	14.00	875.00	42.00	1.00	45.00	1.00	8.00	1.00
8.00	4.00	138.00	33.00	776.00	2.00	113.00	0.00	15.00	0.00
4.00	0.00	2.00	6.00	1.00	871.00	3.00	49.00	11.00	34.00
98.00	15.00	91.00	16.00	76.00	4.00	528.00	2.00	24.00	1.00
0.00	0.00	0.00	3.00	1.00	59.00	0.00	866.00	5.00	49.00
23.00	0.00	16.00	4.00	8.00	7.00	32.00	1.00	921.00	1.00
3.00	0.00	2.00	2.00	2.00	44.00	4.00	80.00	6.00	914.00

Độ chính xác trong mẫu dữ liệu huấn luyện: 83,6%

Độ chính xác trong mẫu dữ liệu kiểm tra: 82%

Độ chính xác chung: 83,4%

Với 10 lần truyền dữ liệu (10 epochs), thời gian chạy

1 luồng: 270.105s

4 luồng: 168.012s

8 luồng: 183.347s

Ma trận lỗi cho mẫu dữ liệu huấn luyện:

4917.00	18.00	59.00	185.00	16.00	3.00	863.00	2.00	20.00	1.00
31.00	5761.00	6.00	64.00	21.00	3.00	29.00	0.00	1.00	0.00
104.00	28.00	4603.00	61.00	441.00	14.00	663.00	0.00	46.00	2.00
284.00	127.00	56.00	5210.00	151.00	4.00	193.00	0.00	26.00	1.00
62.00	22.00	817.00	282.00	4913.00	3.00	664.00	0.00	52.00	3.00
15.00	3.00	16.00	9.00	7.00	5626.00	18.00	189.00	55.00	143.00
464.00	28.00	372.00	138.00	402.00	9.00	3394.00	0.00	60.00	2.00
2.00	2.00	0.00	1.00	0.00	194.00	1.00	5579.00	24.00	243.00
115.00	10.00	71.00	46.00	48.00	35.00	173.00	25.00	5704.00	6.00
6.00	1.00	0.00	4.00	1.00	109.00	2.00	205.00	12.00	5599.00

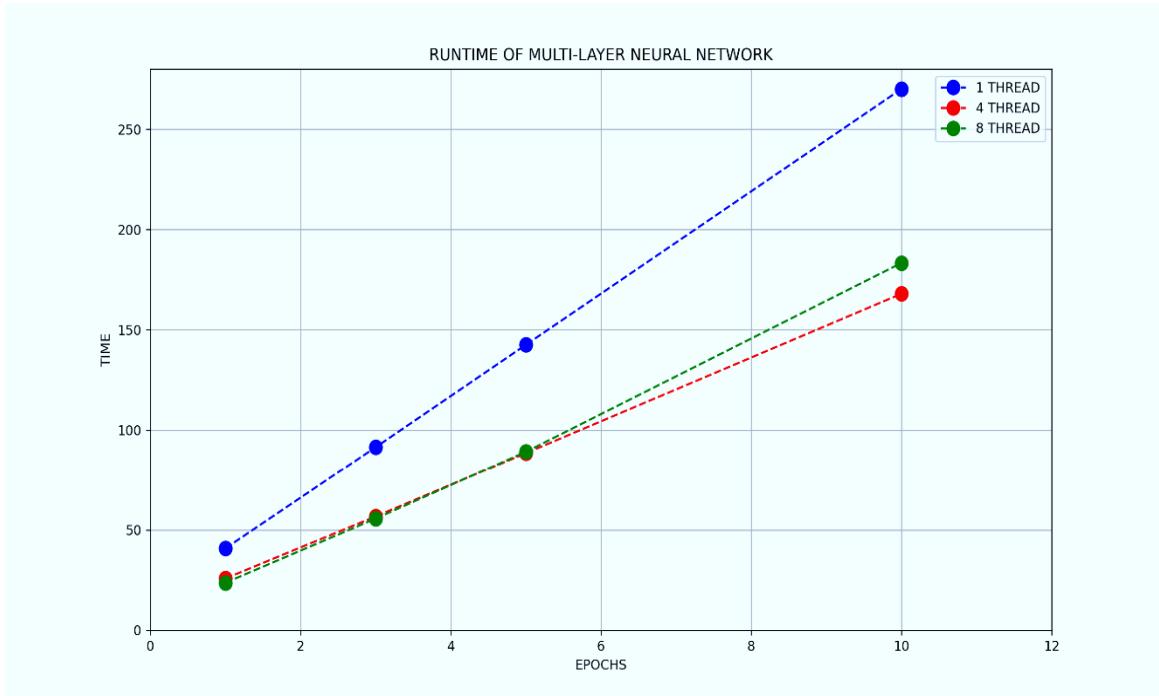
Ma trận lỗi cho mẫu dữ liệu kiểm tra:

786.00	4.00	12.00	33.00	1.00	1.00	179.00	0.00	2.00	0.00
7.00	958.00	1.00	12.00	3.00	2.00	6.00	0.00	0.00	0.00
20.00	8.00	744.00	15.00	72.00	4.00	111.00	0.00	8.00	0.00
51.00	17.00	13.00	855.00	24.00	3.00	29.00	0.00	4.00	0.00
8.00	3.00	145.00	55.00	813.00	0.00	105.00	0.00	12.00	0.00
2.00	2.00	1.00	6.00	1.00	904.00	2.00	50.00	6.00	41.00
97.00	8.00	68.00	18.00	80.00	1.00	537.00	0.00	17.00	0.00
1.00	0.00	0.00	2.00	1.00	53.00	0.00	889.00	7.00	49.00
27.00	0.00	15.00	3.00	5.00	9.00	30.00	3.00	943.00	1.00
1.00	0.00	1.00	1.00	0.00	23.00	1.00	58.00	1.00	909.00

Độ chính xác trong mẫu dữ liệu huấn luyện: 85,5%

Độ chính xác trong mẫu dữ liệu kiểm tra: 83,2%

Độ chính xác chung: 85,2%



Biểu đồ so sánh thời gian chạy của mạng lưới thần kinh trên 1, 4, 8 luồng

Chúng ta có thể nhận thấy sự khác biệt rõ ràng về thời gian giữa chạy đơn luồng so với 4 và 8 luồng, với số lần truyền dữ liệu tăng dần thì thời gian chênh lệch giữa thực thi chương trình trên đơn luồng so với đa luồng cụ thể là 4 và 8 luồng ngày càng lớn.

Với 100 lần truyền dữ liệu (100 epochs)

Chương trình song song sử dụng 4 luồng:

Thời gian thực thi: 2962,47s

Ma trận lỗi cho mẫu dữ liệu huấn luyện:

5563.00	21.00	76.00	115.00	24.00	0.00	542.00	1.00	35.00	2.00
6.00	5830.00	6.00	27.00	8.00	0.00	13.00	0.00	7.00	1.00
43.00	9.00	5265.00	18.00	289.00	1.00	289.00	0.00	38.00	1.00
144.00	106.00	78.00	5606.00	123.00		2.00	125.00	1.00	24.00 0.00
27.00	10.00	380.00	156.00	5377.00	1.00	267.00	0.00	17.00	0.00
3.00	1.00	6.00	2.00	0.00	5856.00	3.00	46.00	38.00	27.00
177.00	17.00	170.00	58.00	162.00	0.00	4721.00	2.00	46.00	0.00
1.00	1.00	2.00	2.00	0.00	81.00	1.00	5798.00	22.00	104.00
31.00	5.00	17.00	14.00	17.00	17.00	38.00	12.00	5765.00	3.00
5.00	0.00	0.00	2.00	0.00	42.00	1.00	140.00	8.00	5862.00

Ma trận lỗi cho mẫu dữ liệu kiểm tra:

836.00	7.00	14.00	39.00	3.00	1.00	181.00	0.00	6.00	1.00
0.00	969.00	1.00	10.00	0.00	0.00	5.00	0.00	0.00	0.00
12.00	3.00	783.00	11.00	75.00	1.00	87.00	0.00	10.00	0.00
34.00	14.00	21.00	888.00	25.00	1.00	35.00	0.00	7.00	1.00
2.00	3.00	99.00	34.00	818.00	0.00	78.00	0.00	3.00	0.00
1.00	1.00	1.00	2.00	1.00	924.00	1.00	26.00	13.00	16.00
103.00	3.00	76.00	14.00	73.00	1.00	593.00	0.00	17.00	0.00
1.00	0.00	0.00	0.00	1.00	40.00	0.00	910.00	3.00	37.00
10.00	0.00	5.00	2.00	4.00	6.00	20.00	0.00	938.00	0.00
1.00	0.00	0.00	0.00	0.00	26.00	0.00	64.00	3.00	945.00

Độ chính xác trong mẫu dữ liệu huấn luyện: 92.7%

Độ chính xác trong mẫu dữ liệu kiểm tra: 86%

Độ chính xác chung: 91.8%

Chúng ta có thể thấy khi tăng số lần truyền dữ liệu thì độ chính xác tăng dần từ độ chính xác chung là 75.9% với 1 lần truyền dữ liệu tới độ chính xác chung là 91.8% với 100 lần truyền dữ liệu chứng tỏ chương trình của chúng ta đã được xây dựng mạng lưới thần kinh trên chương trình đúng cách.

TÀI LIỆU THAM KHẢO

- [1] What is the Fashion MNIST Dataset? | Pinecone
- [2] Fashion MNIST | Kaggle
- [3] Fashion MNIST Visualization | TensorFlow
- [4] Deep Learning Image Classification for Fashion Design - Mohammad Farukh Hashmi (14/6/2022)
- [5] Introduction to Parallel Computing | Springer Nature Switzerland AG (2018)
- [6] Perceptrons | Marvin Minsky, Seymour Papert (15/1/1969)
- [7] Machine Learning - Tom M. Mitchell (1/3/1997)
- [8] Make your own neural network (Tarig Rashid)