

# THE COMPUTER VISION JOURNEY 1

## From Linear Regression to CNNs

---

May 18, 2024

Prepared by: Phan Kỳ Nhân - 2412432



Ho Chi Minh City University of Technology

# Mục lục

<b>1</b>	<b>Giới thiệu</b>	<b>4</b>
<b>2</b>	<b>Linear Regression</b>	<b>5</b>
2.1	Motivation & Dataset . . . . .	5
2.2	Bài toán con nhỏ nhất . . . . .	5
2.2.1	Loss Function: Mean Squared Error (MSE) . . . . .	7
2.2.2	Optimization: Learning Rate và Stochastic Gradient Descent (SGD)	8
2.2.3	Tóm tắt To-Do List cho Linear Regression với 1 feature . . . . .	9
2.3	Tổng quát hóa (Vectorization) . . . . .	10
2.4	Implementation . . . . .	12
<b>3</b>	<b>Logistic Regression</b>	<b>14</b>
3.1	Motivation & Dataset . . . . .	14
3.2	Activation Funtion . . . . .	14
3.2.1	Hàm Logistic (Sigmoid Function) . . . . .	16
3.2.2	Hyperbolic Tangent Function – tanh . . . . .	17
3.3	Loss Function . . . . .	17
3.3.1	Limitation of MSE . . . . .	17
3.3.2	Binary Cross Entropy (BCE) . . . . .	19
3.4	Tổng quát hóa + Vectorization . . . . .	21
3.5	Implementation . . . . .	24
<b>4</b>	<b>SoftMax Regression</b>	<b>26</b>
4.1	Motivation . . . . .	26
4.2	Hướng tiếp cận và Bài toán tổng quát . . . . .	26
4.2.1	Activation Function: SoftMax . . . . .	27
4.2.2	Loss Function: Cross Entropy . . . . .	28
4.3	Tổng quát hóa + Vectorization . . . . .	30
4.4	Implementation . . . . .	34
<b>5</b>	<b>MLP</b>	<b>36</b>
5.1	Motivation and Datasets . . . . .	36
5.1.1	Cách tiếp cận thứ nhất: Tập dữ liệu phân bố theo hàm XOR . . . . .	36
5.1.2	Tiếp cận 2 . . . . .	37
5.1.3	Normalization . . . . .	39
5.1.4	Cách tiếp cận thứ ba: Các bài toán phức tạp hơn . . . . .	43
5.2	MLP và cách train mô hình . . . . .	45
5.2.1	Layers . . . . .	45
5.2.2	Nodes . . . . .	47
5.2.3	Activation Function: ReLU . . . . .	49
5.2.4	Forward Pass . . . . .	50
5.2.5	Backpropagation . . . . .	53
5.3	Implementation . . . . .	55

<b>6 Advanced MLP</b>	<b>57</b>
6.1 Initialization . . . . .	57
6.1.1 Dying ReLU . . . . .	57
6.1.2 Gradient Exploding . . . . .	57
6.1.3 Gradient Vanishing với Sigmoid . . . . .	57
6.1.4 Relation Concept . . . . .	58
6.1.5 Xavier Initialization . . . . .	61
6.1.6 He Initialization . . . . .	62
6.2 Optimization . . . . .	62
6.2.1 Limitation of SGD . . . . .	63
6.2.2 Learning Rate Decay . . . . .	68
6.2.3 Adagrad . . . . .	71
6.2.4 RMSProp . . . . .	73
6.2.5 Momentum . . . . .	75
6.2.6 Adam . . . . .	76
6.3 Generate advance MLP . . . . .	76
<b>7 Convolutional Neural Networks (CNNs)</b>	<b>77</b>
7.1 Motivation and Dataset . . . . .	77
7.2 Hạn chế của MLP trong xử lý ảnh màu . . . . .	78
7.3 Giới thiệu về CNN . . . . .	79
7.4 CIFAR-10 dưới góc nhìn Machine Learning truyền thống . . . . .	80
7.4.1 Trích xuất đặc trưng: Hàm và bộ lọc . . . . .	80
7.4.2 Huấn luyện mô hình . . . . .	82
7.5 Triển khai CNN: Một hướng tiếp cận tích hợp trong học sâu . . . . .	82
7.5.1 Đặc điểm cốt lõi của CNN . . . . .	83
7.5.2 Sai bước (Stride) . . . . .	85
7.5.3 Chiều kênh trong tích chập ( <i>Channel Dimension in Convolution</i> ) . . . . .	85
7.6 Chuyển đổi MLP thành mô hình CNN: Áp dụng trên Fashion-MNIST . . . . .	88
7.6.1 Pooling . . . . .	89
7.6.2 Padding . . . . .	91
7.6.3 1x1 Convolution . . . . .	93
7.6.4 Implementation from Scratch . . . . .	94
<b>8 Pytorch</b>	<b>97</b>
8.1 Autograd . . . . .	97
<b>9 Advance CNN optimize</b>	<b>99</b>
9.1 Noise addiction . . . . .	99
9.2 Z1 Score _ Guassian Distribution . . . . .	99
9.3 Batch Normalization . . . . .	101
9.4 Dropout . . . . .	102
9.5 Weight Decay và Kernel Regularization . . . . .	103
9.6 Advanced Activation Function: Swish . . . . .	103
9.7 Skip connection . . . . .	104
<b>10 Tạm kết</b>	<b>106</b>

# 1 Giới thiệu

---

Trong kỷ nguyên số hiện nay, dữ liệu được tạo ra với tốc độ chưa từng có, mở ra những cơ hội to lớn để khai phá tri thức và tạo ra các giá trị đột phá. Học Máy (Machine Learning) và đặc biệt là Học Sâu (Deep Learning) đã nổi lên như những công cụ quyền năng nhất để biến tiềm năng từ dữ liệu thành hiện thực, thúc đẩy các cuộc cách mạng trong mọi lĩnh vực từ nhận dạng hình ảnh, xử lý ngôn ngữ tự nhiên, xe tự lái cho đến y học chính xác.

Tài liệu này được biên soạn với mục tiêu cung cấp một lộ trình học tập có hệ thống và toàn diện, dẫn dắt người đọc đi từ những viên gạch nền tảng nhất của học máy đến các kiến trúc mạng nơ-ron phức tạp và các kỹ thuật tối ưu hóa hiện đại. Chúng tôi tin rằng việc nắm vững bản chất toán học và cơ chế hoạt động đằng sau mỗi thuật toán là chìa khóa để có thể áp dụng và sáng tạo hiệu quả trong thực tế.

Hành trình của chúng ta sẽ bắt đầu với các mô hình kinh điển: **Linear Regression** và **Logistic Regression**, giúp xây dựng trực giác về các khái niệm cốt lõi như hàm mất mát (loss function), tối ưu hóa bằng gradient descent (tối ưu hóa bằng dốc), và vector hóa (vectorization). Tiếp đó, chúng ta sẽ mở rộng sang bài toán phân loại đa lớp với **SoftMax Regression**.

Cột mốc quan trọng tiếp theo là sự ra đời của **Mạng Nơ-ron Nhân tạo đa lớp (MLP)**, cánh cửa đưa chúng ta vào thế giới học sâu. Chúng ta sẽ mở rộng cấu trúc của MLP, cơ chế lan truyền xuôi (forward pass), và đặc biệt là thuật toán lan truyền ngược (backpropagation) – trái tim của việc huấn luyện mạng nơ-ron. Không chỉ dừng lại ở lý thuyết, chương **Advanced MLP** sẽ trang bị các kỹ thuật thiết yếu để xây dựng một mô hình mạnh mẽ, từ các phương pháp khởi tạo trọng số (He, Xavier) đến các thuật toán tối ưu hóa tiên tiến (Adagrad, RMSProp, Adam).

Khi đã có nền tảng vững chắc, chúng ta sẽ khám phá **Mạng Nơ-ron Tích chập (CNN)**, một kiến trúc đột phá được thiết kế chuyên biệt cho việc xử lý dữ liệu có cấu trúc lưới như hình ảnh. Các khái niệm như tích chập (convolution), pooling, và padding sẽ được diễn giải chi tiết.

Để biến lý thuyết thành sản phẩm, tài liệu sẽ dành một chương giới thiệu về **PyTorch**, một trong những framework học sâu phổ biến nhất hiện nay, cùng với cách hiện thực hóa các mô hình MLP và CNN. Cuối cùng, chúng ta sẽ hoàn thiện kiến thức bằng cách tìm hiểu các kỹ thuật tối ưu hóa và chính quy hóa cao cấp cho CNN như **Batch Normalization, Dropout, và Skip Connection**, những thành phần không thể thiếu trong các mô hình đạt hiệu năng đỉnh cao (state-of-the-art).

Tài liệu này hướng đến các bạn sinh viên, lập trình viên, nhà nghiên cứu và bất kỳ ai mong muốn có một cái nhìn sâu sắc và có hệ thống về lĩnh vực học sâu.

## 2 Linear Regression

---

### 2.1 Motivation & Dataset

Một phòng ban marketing thuộc công ty bán đồ điện tử đang đối mặt với nguy cơ giải thể sau khi nhiều chiến dịch quảng cáo không đạt được kỳ vọng KPI. Họ từng đầu tư ngân sách vào ba kênh truyền thông chính: **TV**, **radio**, và **newspaper**, nhưng không xác định được chiến lược phân bổ ngân sách tối ưu.

Trong nỗ lực phân tích và tối ưu hiệu suất các chiến dịch quảng cáo, phòng ban đã cung cấp cho chúng ta một tập dữ liệu gồm các chiến dịch trước đây, bao gồm chi phí bỏ ra trên từng kênh truyền thông và kết quả doanh số (*sales*) đạt được.

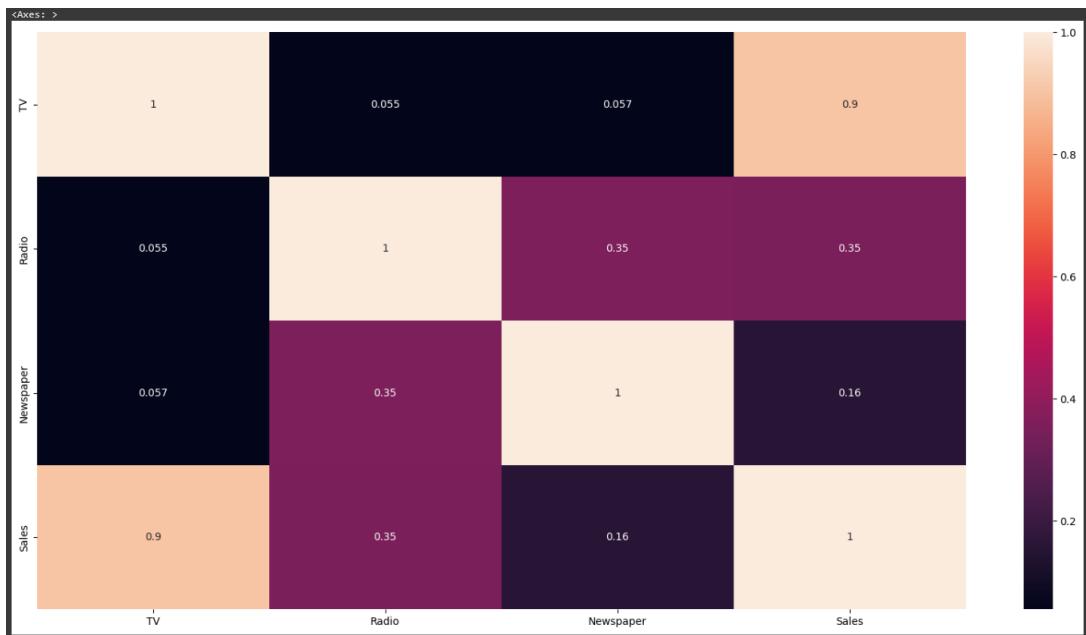
	TV	Radio	Newspaper	Sales
0	230.1	37.8	69.2	22.1
1	44.5	39.3	45.1	10.4
2	17.2	45.9	69.3	12.0
3	151.5	41.3	58.5	16.5
4	180.8	10.8	58.4	17.9
...	...	...	...	...
195	38.2	3.7	13.8	7.6
196	94.2	4.9	8.1	14.0
197	177.0	9.3	6.4	14.8
198	283.6	42.0	66.2	25.5
199	232.1	8.6	8.7	18.4

Hình 1: Advertising dataset được trực quan hóa bằng pandas

Đây là một ví dụ kinh điển khi bước đầu làm quen với **Linear Regression**. Tuy nhiên, trước khi xây dựng mô hình tổng quát, chúng ta sẽ tiếp cận bài toán dưới dạng đơn giản nhất – chỉ xét một đặc trưng (*feature*).

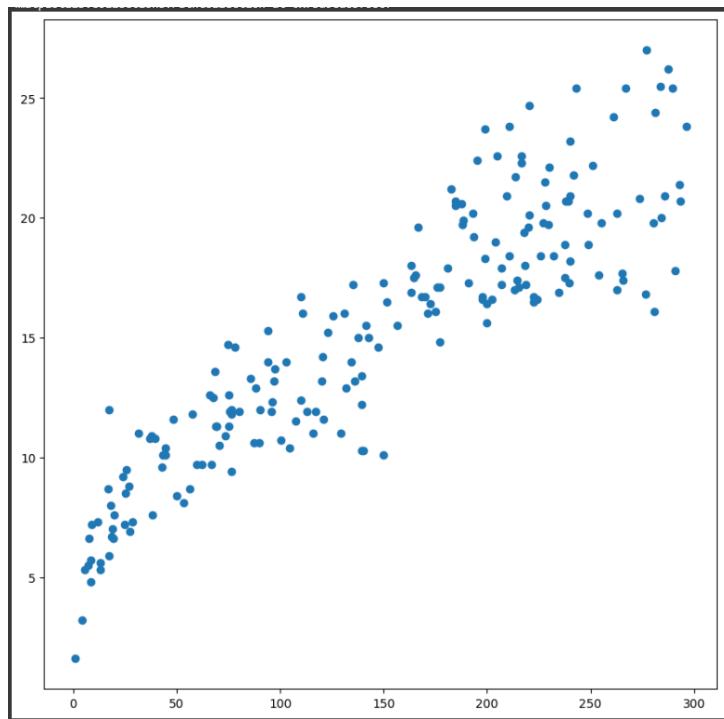
### 2.2 Bài toán con nhỏ nhất

Ta tiến hành trực quan hóa mối quan hệ giữa các đặc trưng thông qua biểu đồ *heatmap*:



Hình 2: Biểu đồ heatmap của Advertising dataset bằng seaborn

Từ hình trên, có thể thấy rằng chỉ số **TV** có correlation mạnh nhất với kết quả **sales**. Vì vậy, để tiếp cận bài toán con này một cách dễ hiểu nhất, ta sẽ chỉ sử dụng duy nhất feature **TV** để dự đoán sales.



Hình 3: Biểu đồ phân tán giữa TV và sales (200 chiến dịch marketing)

Nhiệm vụ đặt ra là tìm một đường thẳng (hay hyperplane trong trường hợp nhiều features) sao cho "fit" nhất với dữ liệu, từ đó có thể dự đoán **sales** dựa trên giá trị của **TV** thông qua đường thẳng đó.

Chúng ta biết rằng phương trình tổng quát của một đường thẳng là:

$$y = ax + b$$

Khi áp dụng vào bài toán này, ta có:

$$\text{sales} = w \cdot \text{TV} + b$$

trong đó  $w$  và  $b$  là hai tham số cần tìm của mô hình.

**Vấn đề đặt ra là:** *Làm thế nào để tìm được đường thẳng “fit” cho dữ liệu?* Khác với các bài toán đại số thông thường nơi chúng ta tìm giá trị  $x, y$  khi biết tham số, ở đây, chúng ta cần làm điều ngược lại: tìm tham số  $w, b$  sao cho đường thẳng đó tiến về với dữ liệu thực nghiệm.

Khi đã tìm được tham số phù hợp, ta có thể dự đoán **sales** tương ứng cho mọi giá trị **TV**.

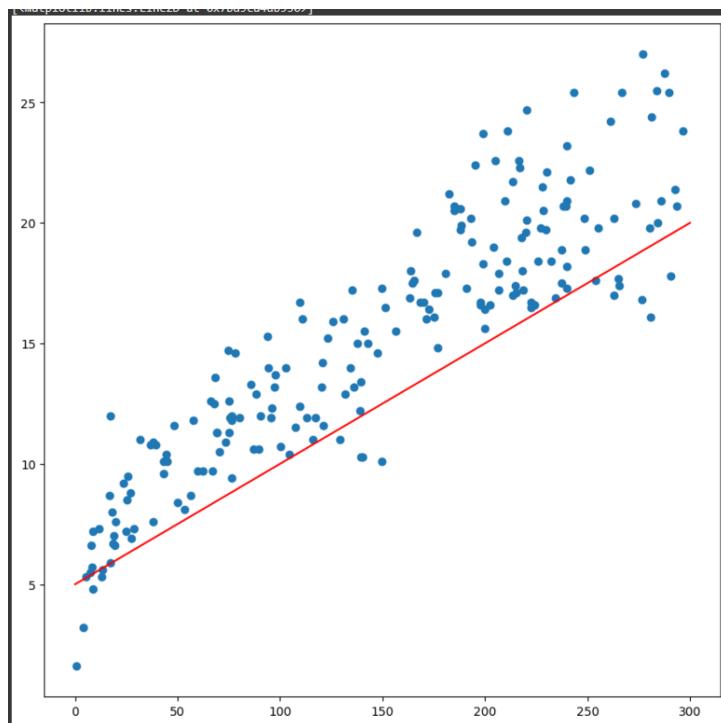
Để làm được điều đó, ta định nghĩa:

$$\hat{y} = w \cdot x + b$$

là **giá trị dự đoán** ( $\hat{y}$ ) của mô hình đối với đầu vào  $x = \text{TV}$ . Nếu mô hình dự đoán tốt,  $\hat{y}$  sẽ gần với giá trị thực tế  $y$ , và sự khác biệt giữa chúng – gọi là **lỗi** (**loss**) – sẽ là đối tượng được phân tích trong phần tiếp theo.

### 2.2.1 Loss Function: Mean Squared Error (MSE)

Với hai trọng số ban đầu  $w$  và  $b$ , chúng ta chưa thể xác định liệu chúng có phù hợp hay không. Do đó, người ta sử dụng một hàm số gọi là **hàm mất mát** (**loss function**) nhằm đo lường mức độ sai lệch giữa giá trị dự đoán và giá trị thực tế.



Hình 4: 1 ví dụ minh họa cho giá trị  $w_k, b_k$

Giả sử với cặp giá trị trọng số  $w_k, b_k$ , ta vẽ được một đường thẳng đại diện cho mô hình dự đoán. Mỗi điểm dữ liệu sẽ có một sai số giữa giá trị dự đoán  $\hat{y}$  và giá trị thực tế  $y$ . Sai số này có thể được đo bằng các cách:

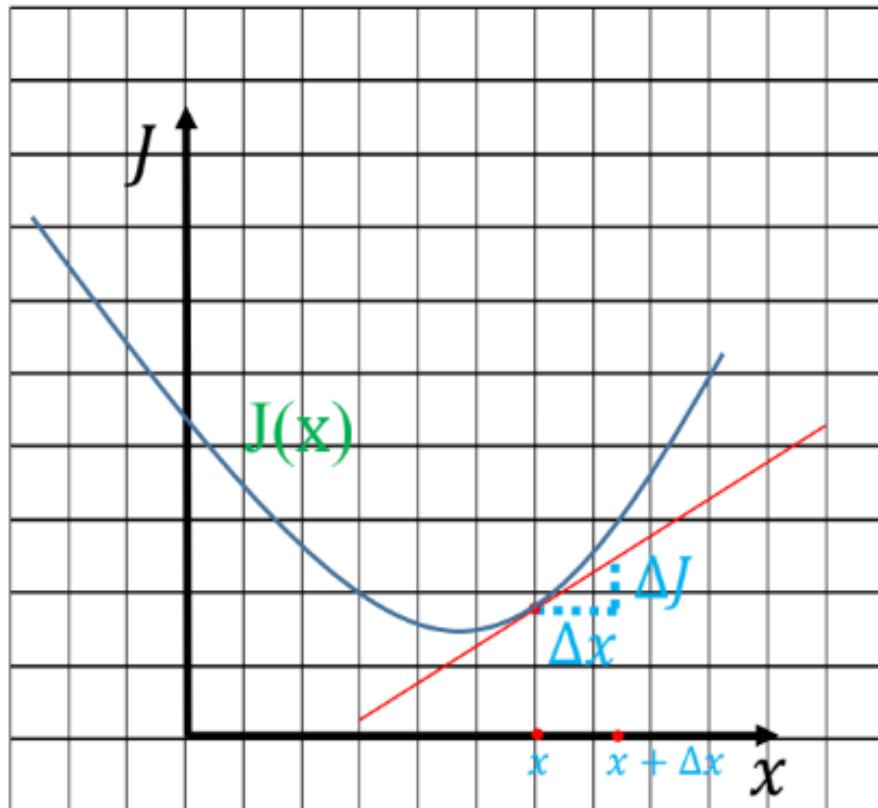
- Khoảng cách tuyệt đối: Loss =  $|\hat{y} - y|$
- Bình phương sai số: Loss =  $(\hat{y} - y)^2$

Trong bài toán này, ta chọn cách sử dụng bình phương sai số – gọi là **Mean Squared Error (MSE)** – do hàm này có dạng lồi (convex), rất thuận lợi cho việc tối ưu hóa (sẽ được trình bày ở phần sau). Tuy hàm trị tuyệt đối cũng có thể sử dụng, nhưng việc tối ưu hóa sẽ phức tạp hơn.

Tiếp theo, sau khi có được giá trị loss rồi thì chúng ta cần phải điều chỉnh các trọng số để hyperplane càng "fit" theo dữ liệu, để làm được điều đó chúng ta sẽ chuyển sang khái niệm tối ưu hóa.

### 2.2.2 Optimization: Learning Rate và Stochastic Gradient Descent (SGD)

Xét đồ thị  $J(x)$  và tiếp tuyến tại điểm  $x$ :



Hình 5: Đồ thị hàm mất mát  $J(x)$  và tiếp tuyến tại điểm  $x$

Về mặt hình học, đạo hàm của một hàm số tại điểm  $x$  chính là hệ số góc của tiếp tuyến tại điểm đó, được tính bởi:

$$f'(x) = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} = \tan(\alpha)$$

Từ đó, ta biết rằng đạo hàm biểu thị tốc độ và hướng thay đổi của hàm số. Áp dụng vào bài toán Linear Regression, ta có hàm mất mát  $\mathcal{L}(w, b) = (\hat{y} - y)^2$ , với  $\hat{y} = wx + b$ . Đây là một hàm lồi nên ta có thể sử dụng Gradient Descent để tìm cực tiểu toàn cục.

**Ý tưởng chính:** Ta coi các trọng số  $w, b$  là một điểm trên đồ thị hàm mất mát. Ta biết được giá trị đạo hàm sẽ chuyển dấu từ '-' sang '+' khi đi qua điểm cực tiểu; Do đó, nếu tính được đạo hàm tại điểm đó, ta có thể điều chỉnh các trọng số theo hướng **ngược dấu đạo hàm** để tiến dần đến điểm cực tiểu.

Sau khi xác định được hướng, việc tiếp theo chúng ta cần làm là xác định độ lớn của bước di chuyển. Liệu có thể di chuyển một lượng đúng bằng giá trị đạo hàm không? Câu trả lời là **không**, bởi vì nếu đạo hàm có giá trị quá lớn, điểm đang khảo sát có thể “bay” ra khỏi vùng tối ưu, thậm chí vượt khỏi cả vùng khảo sát.

Do đó, chúng ta cần thêm một hệ số để điều chỉnh hiện tượng này. Cuối cùng, ta có công thức cập nhật tham số như sau:

$$w \leftarrow w - \eta \cdot \frac{\partial \mathcal{L}}{\partial w} \quad \text{và} \quad b \leftarrow b - \eta \cdot \frac{\partial \mathcal{L}}{\partial b}$$

Trong đó,  $\eta$  là **learning rate** — một siêu tham số (hyperparameter) quan trọng quyết định độ lớn của bước nhảy trong quá trình tối ưu.

**Công thức đạo hàm:** Giả sử  $\hat{y} = f(w, b) = wx + b$ , và loss được định nghĩa là  $\mathcal{L} = g(\hat{y}) = (\hat{y} - y)^2$ .

Từ mối quan hệ chuỗi giữa các hàm, để cập nhật giá trị cho các tham số, ta cần tính đạo hàm riêng của  $\mathcal{L}$  theo từng tham số. Cụ thể:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w} = 2x(\hat{y} - y) \\ \frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b} = 2(\hat{y} - y) \end{aligned}$$

**Chọn Learning Rate phù hợp:** Việc lựa chọn giá trị  $\eta$  là rất quan trọng:

- Nếu quá nhỏ: quá trình học sẽ diễn ra rất chậm hoặc không hội tụ.
- Nếu quá lớn: mô hình có thể dao động zigzag hoặc phân kỳ.

Trong thực tế,  $\eta = 0.01$  là giá trị thường được đề xuất, tuy nhiên cần thử nghiệm và phân tích đặc trưng dữ liệu (data visualization & analysis) để lựa chọn giá trị tối ưu nhất.

### 2.2.3 Tóm tắt To-Do List cho Linear Regression với 1 feature

1. Lấy một cặp dữ liệu huấn luyện ( $x, y$ )
2. Tính giá trị dự đoán:  $\hat{y} = wx + b$
3. Tính hàm mất mát:  $L = (\hat{y} - y)^2$

4. Tính gradient:

$$\frac{\partial L}{\partial w} = 2x(\hat{y} - y) \quad , \quad \frac{\partial L}{\partial b} = 2(\hat{y} - y)$$

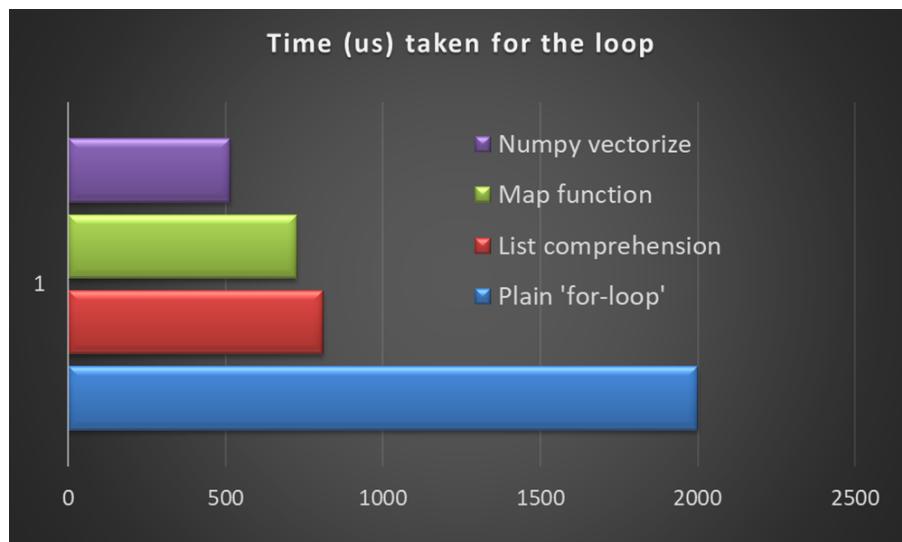
5. Cập nhật trọng số:

$$w = w - \eta \cdot \frac{\partial L}{\partial w} \quad , \quad b = b - \eta \cdot \frac{\partial L}{\partial b}$$

### 2.3 Tổng quát hóa (Vectorization)

Vậy là chúng ta đã đi qua ý tưởng của bài toán Linear Regression với một feature. Tuy nhiên, trong thực tế, các bài toán thường có nhiều features hơn, ví dụ như bài toán *Advertising* ban đầu. Ngoài ra, thay vì sử dụng phương pháp **stochastic** (cập nhật theo từng mẫu), chúng ta có thể sử dụng **mini-batch** hoặc **batch gradient descent** để tránh hiện tượng *zigzag* khi cập nhật tham số, giúp mô hình hội tụ ổn định hơn.

Moreover, như đã biết, Python không được tối ưu cho vòng lặp — điều này không gây vấn đề cho các mô hình đơn giản như Linear Regression, nhưng sẽ là điểm nghẽn khi làm việc với các mô hình có hàng triệu tham số trong những mô hình sẽ được đề cập sắp tới và cả tương lai. Đó là lý do chúng ta sẽ dùng NumPy và áp dụng concept của các phép toán ma trận trong đại số tuyến tính để giúp tăng tốc nhờ phép biến đổi vector hoá.



Hình 6: So sánh tốc độ giữa các kiểu lặp và khử lặp (vector hóa)

Không mất tính tổng quát, giả sử:

- $m$ : số lượng mẫu trong một batch (batch size)
- $n$ : số tham số (tức là số đặc trưng +1 để tính thêm bias)

Ta định nghĩa:

$$\mathbf{X} \in \mathbb{R}^{m \times n}, \quad \boldsymbol{\theta} \in \mathbb{R}^{n \times 1}$$

Trong đó:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1,n-1} & 1 \\ x_{21} & x_{22} & \cdots & x_{2,n-1} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{m,n-1} & 1 \end{bmatrix}, \quad \boldsymbol{\theta} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{n-1} \\ b \end{bmatrix}$$

Khi đó:

$$\hat{y}^{(i)} = \hat{y}^{(i)} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_m \end{bmatrix} = X \cdot \boldsymbol{\theta} \quad (\text{kích thước: } m \times n \cdot n \times 1 = m \times 1)$$

Tiếp theo, ta có hàm mất mát:

$$L^{(i)} = (\hat{y}^{(i)} - y^{(i)})^2 = \begin{bmatrix} (\hat{y}_1 - y_1)^2 \\ (\hat{y}_2 - y_2)^2 \\ \vdots \\ (\hat{y}_m - y_m)^2 \end{bmatrix} = (\hat{y}^{(i)} - y^{(i)}) \circ (\hat{y}^{(i)} - y^{(i)})$$

Trong đó  $\circ$  là phép nhân từng phần tử (Hadamard).

Gradient theo từng tham số:

$$\frac{\partial L^{(i)}}{\partial \theta} = \begin{bmatrix} \frac{\partial L^{(i)}}{\partial w_n} \\ \frac{\partial L^{(i)}}{\partial w_{n-1}} \\ \vdots \\ \frac{\partial L^{(i)}}{\partial w_1} \\ \frac{\partial L^{(i)}}{\partial b} \end{bmatrix} = 2 \cdot X^T \cdot (\hat{y}^{(i)} - y^{(i)})$$

Cuối cùng, cập nhật tham số và nhảy sang epoch tiếp theo:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \frac{\eta}{m} \cdot \frac{\partial L^{(i)}}{\partial \theta}$$

**Ghi chú:** Việc thêm cột toàn 1 vào  $X$  là để gom bias  $b$  vào vector tham số  $\boldsymbol{\theta}$ , nhờ đó các phép toán vector hóa trở nên đồng nhất, tiện lợi khi lập trình và dễ dàng mở rộng mô hình hơn.

**Lưu ý.** Sau một quá trình chứng minh, ta thu được công thức tổng quát như sau:

1. Lấy một minibatch gồm  $m$  cặp dữ liệu huấn luyện  $(\mathbf{x}^{(i)}, y^{(i)})$ , với  $i = 1, \dots, m$ .

Tạo ma trận đặc trưng  $\mathbf{X} \in \mathbb{R}^{m \times n}$  bằng cách nối ma trận đặc trưng ban đầu  $\tilde{\mathbf{X}} \in \mathbb{R}^{m \times (n-1)}$  với một cột toàn số 1 (để mô hình hóa bias):

$$\mathbf{X} = [\tilde{\mathbf{X}} \ 1]$$

2. Tính vector dự đoán:

$$\hat{\mathbf{y}} = \mathbf{X} \cdot \boldsymbol{\theta}$$

3. Tính vector măt măt (theo từng phần tử):

$$L = (\hat{\mathbf{y}} - \mathbf{y})^2$$

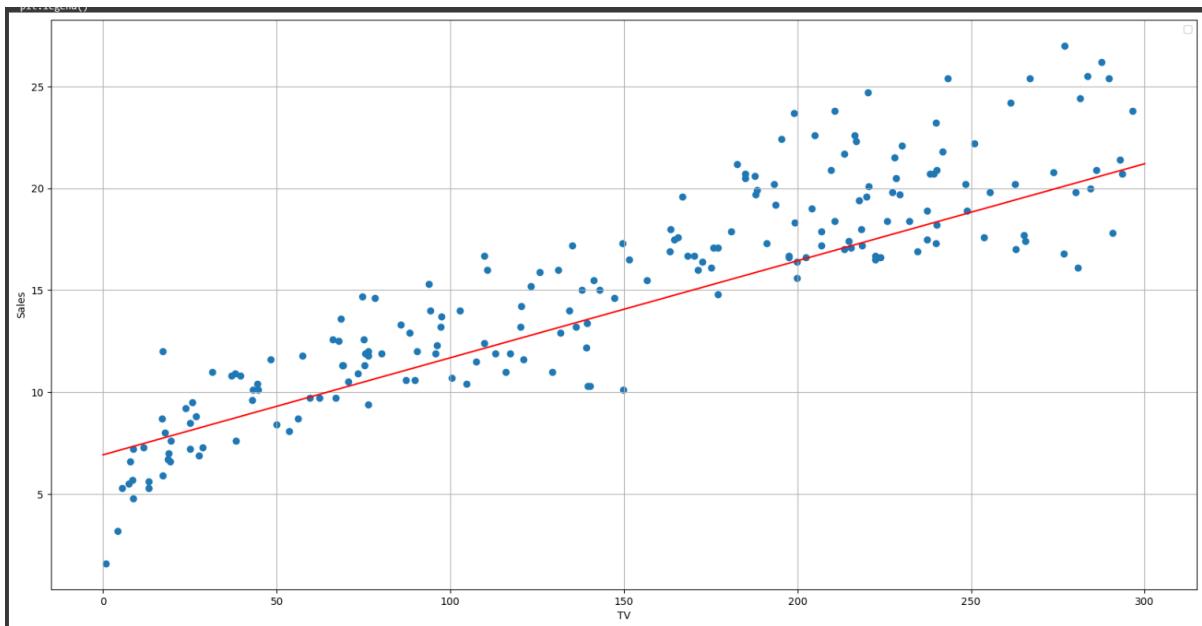
4. Tính gradient của hàm măt măt theo vector tham số:

$$\frac{\partial L}{\partial \boldsymbol{\theta}} = 2\mathbf{X}^\top \cdot (\hat{\mathbf{y}} - \mathbf{y})$$

5. Cập nhật vector tham số theo thuật toán Gradient Descent:

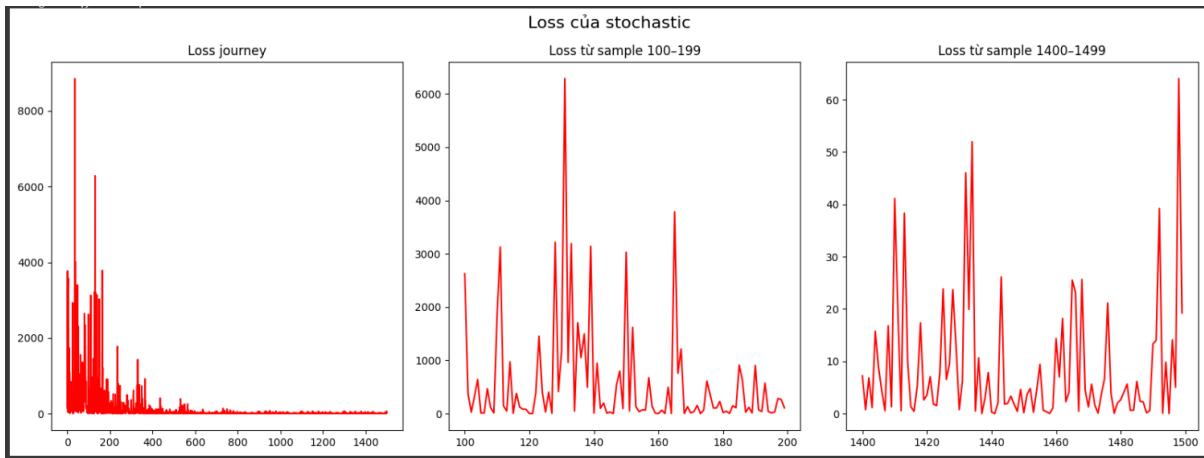
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\eta}{m} \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}, \quad \text{trong đó } \eta \text{ là learning rate.}$$

## 2.4 Implementation



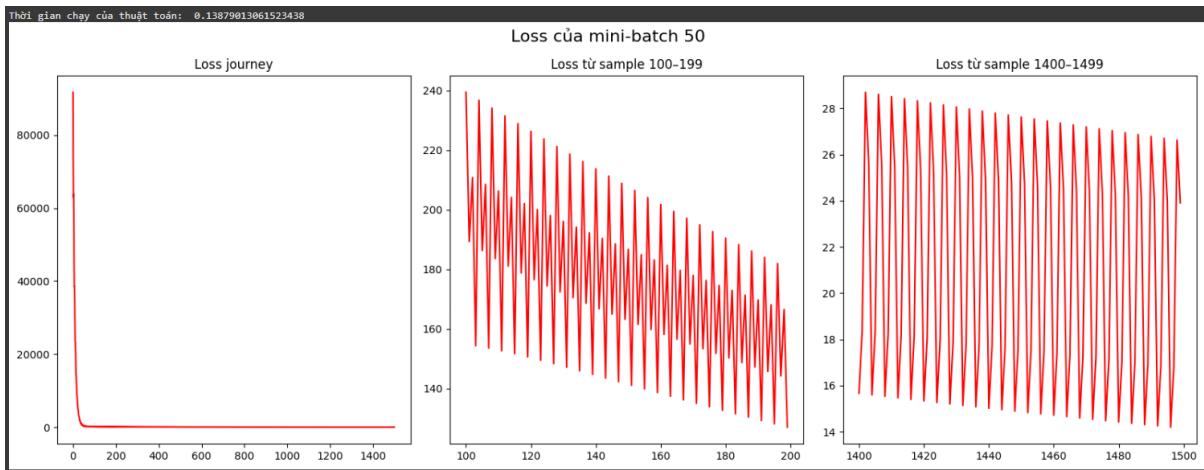
Hình 7: Đường Linear Regression khớp với dataset của bài toán con nhỏ nhất (TVs vs. Sales)

Sau quá trình huấn luyện, ta thu được một đường thẳng thể hiện mối quan hệ tuyến tính giữa TV và Sales.



Hình 8: Biến thiên loss theo epoch khi train kiểu Stochastic

**Nhận xét:** Khi sử dụng phương pháp huấn luyện *Stochastic Gradient Descent* (SGD), thời gian thực thi là 0.8 giây cho 500 epoch. *loss function* giảm theo kỳ vọng. Tuy nhiên, do thuật toán cập nhật tham số sau mỗi mẫu dữ liệu đơn lẻ, nên đường cong loss xuất hiện hiện tượng dao động mạnh (zigzag), đặc biệt vẫn còn khá lớn kể cả ở các epoch cuối. Đây là đặc trưng của kiểu train Stochastic do tính chất cập nhật "tức thời" và mang tính nhiễu cao.



Hình 9: Biến thiên loss theo epoch khi train kiểu Mini-batch size = 50

**Nhận xét:** Khi huấn luyện với *Mini-batch Gradient Descent* (batch size = 50), thời gian thực thi giảm xuống còn 0.1 giây cho 500 epoch — nhanh hơn 8 lần so với Stochastic, nhờ việc tận dụng tính toán ma trận thay vì lặp qua từng mẫu riêng lẻ.

Mặc dù đường cong loss vẫn có zigzag nhưng lại mang tính chu kỳ và kiểm soát được — đặc trưng của mini-batch khi trung hoà nhiễu trong cập nhật. Điều này cho thấy mini-batch là một điểm cân bằng giữa hiệu quả tính toán và độ ổn định trong quá trình hội tụ.

Source code( Google Colab) : [Linear Regression](#)

## 3 Logistic Regression

---

### 3.1 Motivation & Dataset

Sau khi giúp phòng Marketing tránh khỏi nguy cơ giải thể nhờ áp dụng Linear Regression vào bài toán quảng cáo, bạn đã nhận được nhiều sự chú ý và lời mời từ các lĩnh vực khác. Lần này, một nhà thực vật học tìm đến bạn với mong muốn giải quyết một bài toán phân loại hoa.

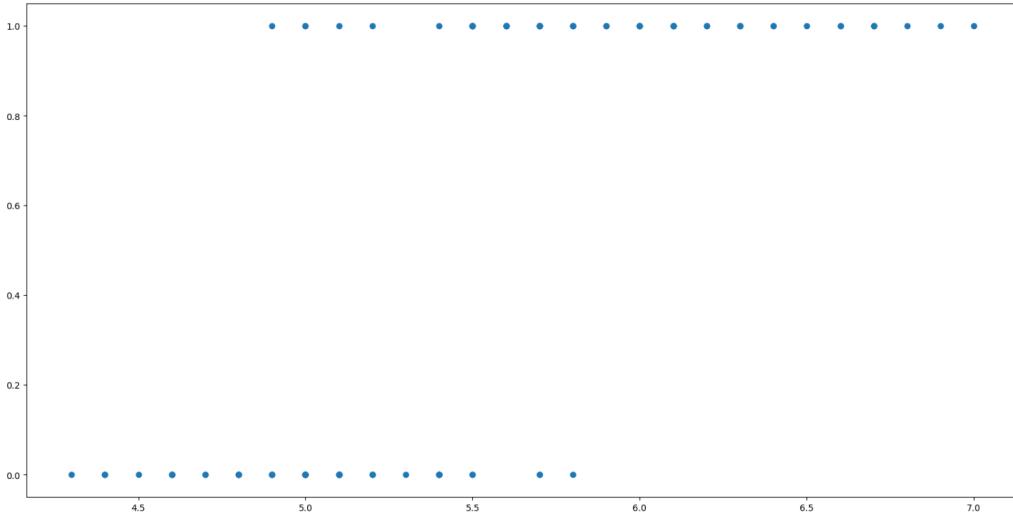
Trong quá trình nghiên cứu các loài hoa Iris, người này thu thập được nhiều dữ liệu về kích thước và chiều dài của các cánh hoa (sepal và petal). Vấn đề đặt ra là: liệu có thể xác định một ngưỡng cụ thể nào đó từ các đặc trưng hình thái này để phân biệt hai loài hoa Iris khác nhau?

	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
95	5.7	3.0	4.2	1.2	1
96	5.7	2.9	4.2	1.3	1
97	6.2	2.9	4.3	1.3	1
98	5.1	2.5	3.0	1.1	1
99	5.7	2.8	4.1	1.3	1
100 rows × 5 columns					

Hình 10: Dataset hoa Iris

### 3.2 Activation Function

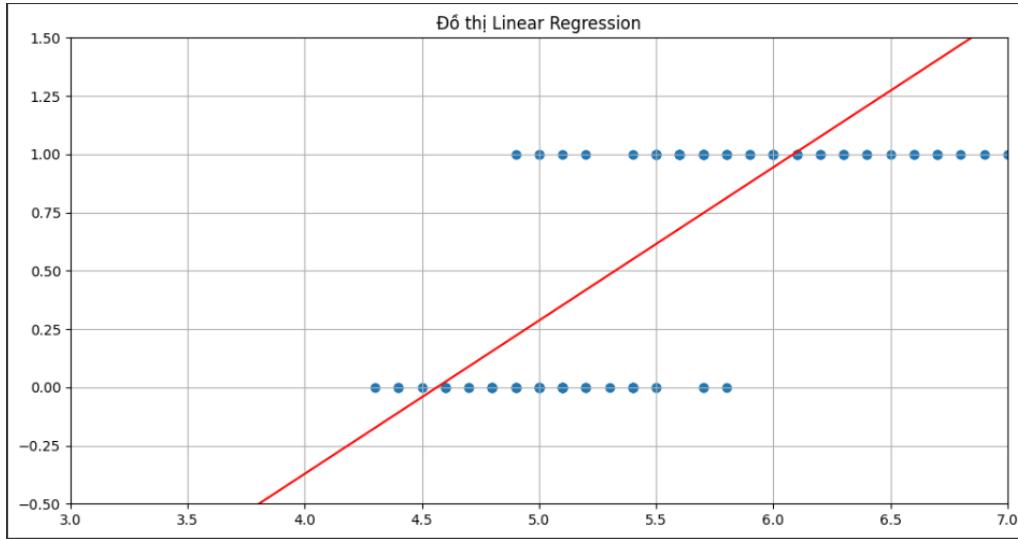
Tương tự như cách tiếp cận với Linear Regression, bạn quyết định bắt đầu từ một bài toán đơn giản: chỉ sử dụng một feature để giảm số chiều và độ phức tạp, nhằm đưa ra lời giải tổng quát hơn cho vấn đề phân loại.



Hình 11: Phân bố dữ liệu theo chiều dài cánh Petal và loại hoa

Với cách tiếp cận này, tôi thấy thử áp dụng Linear Regression để mô hình hóa mối quan hệ giữa chiều dài Petal và loài hoa cũng là một idea không tệ vì căn bản cũng có các điểm phân bố và chúng ta cần tìm đường thẳng "fit" với các phân bố đó, sau đó chúng ta sử dụng một ngưỡng (threshold) như 0.5 để phân loại

Cũng áp dụng các concept của Linear Regression, từ tính giá trị dự đoán cho đến hàm loss rồi optimize các tham số, chúng ta có được kết quả dưới đây



Hình 12: Đường hồi quy tuyến tính trên dữ liệu hoa Iris

Tuy kết quả thu được từ *Linear Regression* không quá tệ, nhưng chúng ta có thể cảm thấy có điều gì đó chưa thật sự ổn: đầu ra của mô hình là toàn bộ tập số thực  $\mathbb{R}$ , trong khi bài toán phân loại lại chỉ quan tâm đến các giá trị trong đoạn  $[0, 1]$ .

Một cách khắc phục đơn giản là cắt ngưỡng: gán các giá trị nhỏ hơn 0 thành 0, các giá trị lớn hơn 1 thành 1, và lấy mốc 0.5 làm ranh giới phân loại. Tuy nhiên, phương pháp này lại khá "hard", không linh hoạt với những trường hợp ngoại lệ (outliers) và có thể dẫn đến sai lệch trong phân loại.

**Giải pháp.** Chúng ta cần một hàm ánh xạ liên tục từ  $\mathbb{R}$  về đoạn  $[0, 1]$ , vừa duy trì tính liên tục, vừa mang ý nghĩa xác suất — giúp quá trình dự đoán trở nên "soft" hơn. Đây cũng là lý do các nhà nghiên cứu phát triển các **activation functions**, nhằm biến đổi mô hình tuyến tính thành phi tuyến để phù hợp hơn với từng loại bài toán.

Trong Logistic Regression, chúng ta sử dụng một hàm đặc biệt gọi là **sigmoid**. Đây là tên gọi chung của những hàm số có dạng đường cong hình chữ S.

### 3.2.1 Hàm Logistic (Sigmoid Function)

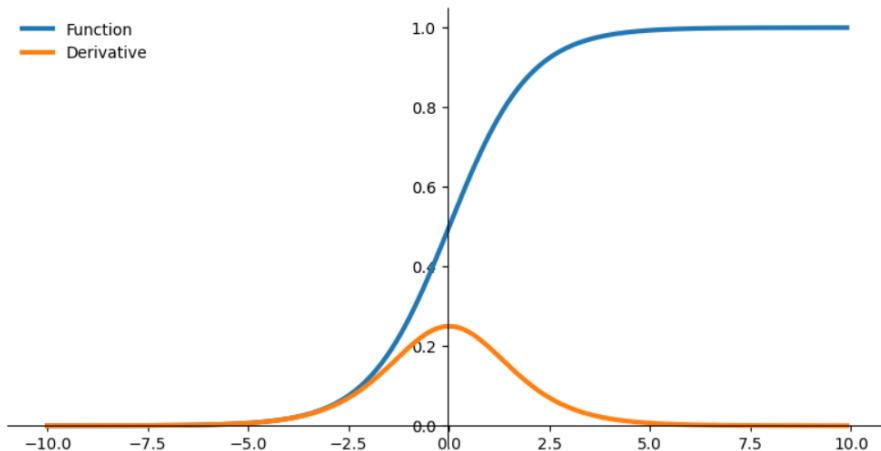
Logistic Function là hàm sigmoid phổ biến nhất nên thường được gọi ngắn gọn là sigmoid luôn. Hàm sigmoid được định nghĩa như sau:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

#### Đạo hàm của hàm sigmoid:

Ta tiến hành lấy đạo hàm theo từng bước:

$$\begin{aligned} \text{sigmoid}'(x) &= \frac{d}{dx} (1 + e^{-x})^{-1} = -(1 + e^{-x})^{-2} \cdot (-e^{-x}) \\ &= \frac{e^{-x}}{(1 + e^{-x})^2} = \frac{1}{1 + e^{-x}} \left(1 - \frac{1}{1 + e^{-x}}\right) \\ &= \text{sigmoid}(x) \cdot (1 - \text{sigmoid}(x)) \end{aligned}$$



Hình 13: Hàm sigmoid và đạo hàm của nó

#### Ưu điểm của sigmoid:

- Biến đổi toàn bộ trực thực  $\mathbb{R}$  về đoạn  $[0, 1]$ , rất phù hợp để biểu diễn xác suất.
- Đạo hàm đơn giản và dễ tính, giúp tối ưu hóa mô hình nhanh chóng.
- Hàm sigmoid là một hàm đơn điệu tăng và có giới hạn trên và dưới (bounded), giúp mô hình ổn định hơn.

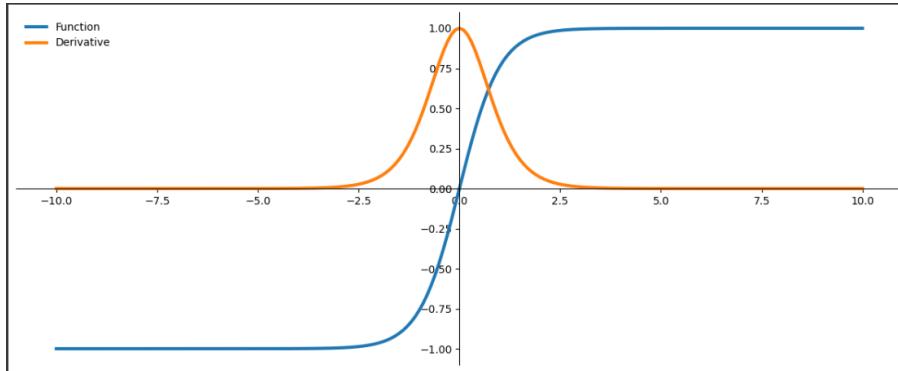
### 3.2.2 Hyperbolic Tangent Function – tanh

Một hàm khác cũng có khả năng biến đổi trực thực về một miền nhỏ hơn là hàm **tanh**, được định nghĩa như sau:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

**Đạo hàm của hàm tanh:**

$$\begin{aligned}\tanh'(x) &= \left( \frac{e^x - e^{-x}}{e^x + e^{-x}} \right)' \\ &= \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2} \\ &= \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2} \\ &= 1 - \tanh^2(x)\end{aligned}$$



Hình 14: Hàm tanh và đạo hàm của nó

## 3.3 Loss Function

### 3.3.1 Limitation of MSE

Chúng ta xây dựng hàm loss nhằm mô tả mức độ phù hợp giữa dự đoán và giá trị thực tế. Trực giác cho thấy một mô hình dự đoán chạy "hoàn hảo" là khi:

- Nếu  $y = 1$  thì  $\hat{y} \rightarrow 1$
- Nếu  $y = 0$  thì  $\hat{y} \rightarrow 0$

Vậy liệu ta có thể sử dụng MSE như trong bài toán hồi quy tuyến tính không? Câu trả lời là **không**, và điều này được chứng minh như sau:

**Lý do không dùng MSE:** Trong bài toán Linear Regression, đầu ra  $\hat{y}$  là một hàm tuyến tính theo đầu vào. Khi đó:

$$\mathcal{L}_{\text{MSE}} = (y - \hat{y})^2$$

là một hàm bậc hai convex (lồi), dễ dàng tối ưu bằng gradient descent.

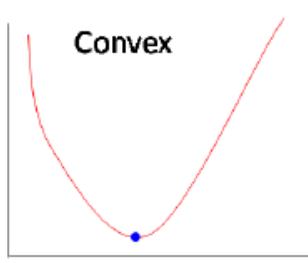
Tuy nhiên, trong Logistic Regression, đầu ra  $\hat{y}$  được tính thông qua hàm sigmoid nũa:

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}} \quad \text{với } z = \theta^T x$$

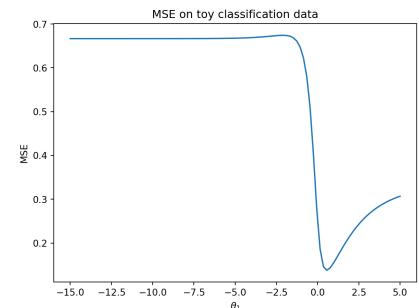
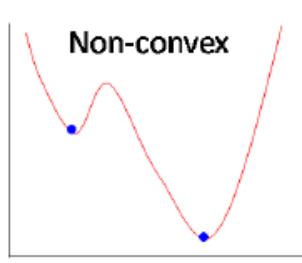
Do đó khi sử dụng MSE, ta có:

$$\mathcal{L}(z) = (y - \sigma(z))^2$$

Do  $\sigma(z)$  là một hàm phi tuyến, nên  $\mathcal{L}(z)$  trở thành **hàm phi tuyến bậc cao và không凸**.



Hàm non-convex khó tối ưu



Hàm non-convex không hội tụ

**Phân tích toán học chi tiết:** Ta khảo sát tính convex bằng cách phân tích đạo hàm bậc hai của  $\mathcal{L}(z)$  theo tham số  $\theta_i$ .

Xét đạo hàm bậc nhất

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \theta_i} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial \theta_i} \\ &= 2 \cdot (y - \hat{y}) \cdot \hat{y}(1 - \hat{y}) \cdot x_i \\ &= 2x_i(-\hat{y}^3 + \hat{y}^2 - y\hat{y} + y\hat{y}^2) \end{aligned}$$

Đạo hàm bậc hai:

$$\begin{aligned} \frac{\partial^2 \mathcal{L}}{\partial \theta_i^2} &= \frac{\partial}{\partial \theta_i} [2x_i(-\hat{y}^3 + \hat{y}^2 - y\hat{y} + y\hat{y}^2)] \\ &= 2x_i[-3\hat{y}^2x_i\hat{y}(1 - \hat{y}) + 2x_i\hat{y}\hat{y}(1 - \hat{y}) - yx_i\hat{y}(1 - \hat{y}) + 2x_iy\hat{y}\hat{y}(1 - \hat{y})] \\ &= 2x_i^2\hat{y}(1 - \hat{y})[-3\hat{y}^2 + 2\hat{y} - y + 2y\hat{y}] \end{aligned}$$

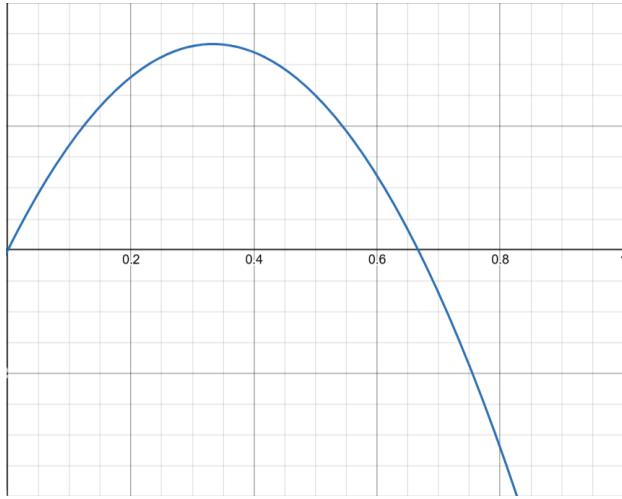
Xét hàm số trong dấu ngoặc:

$$f(\hat{y}) = -3\hat{y}^2 + 2\hat{y} - y + 2y\hat{y}$$

Ta khảo sát theo hai trường hợp:

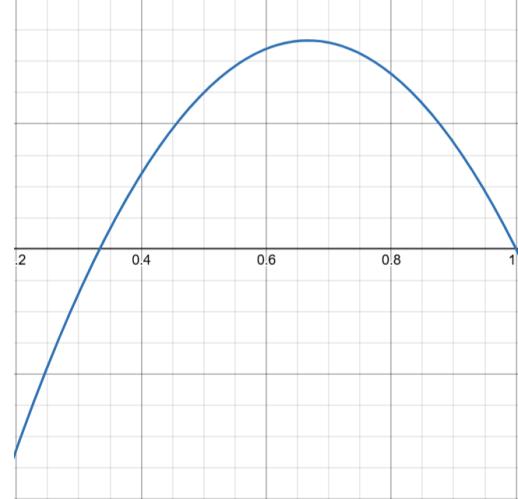
Trường hợp  $y = 0$ :

$$f(\hat{y}) = -3\hat{y}^2 + 2\hat{y}$$



Trường hợp  $y = 1$ :

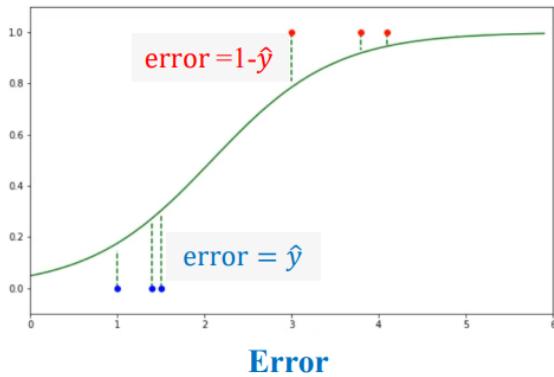
$$f(\hat{y}) = -3\hat{y}^2 + 4\hat{y} - 1$$



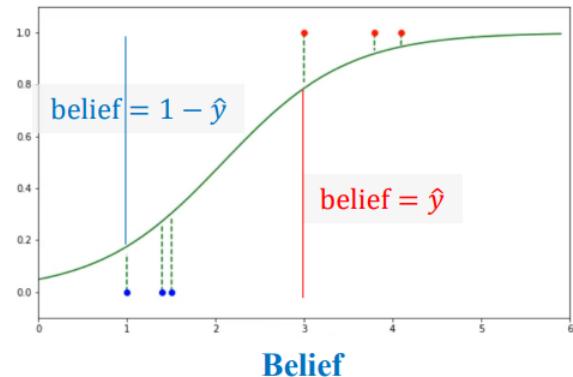
Do đó chúng ta thấy được với mọi trường hợp của  $y$  thì đạo hàm bậc 2 của loss luôn tồn tại điểm  $< 0$  và vì thế hàm không lồi, nên chúng ta không thể sử dụng được MSE.

### 3.3.2 Binary Cross Entropy (BCE)

Vì lý do trên nên ta bắt buộc phải xây dựng 1 hàm loss mới để có thể sử dụng được trong bài toán này, ta sẽ đi vào các khái niệm sau:



Hình 15: Tiếp cận loss theo error



Hình 16: Tiếp cận loss theo belief

Như chứng minh ở trên, nếu ta cứ tiếp tục tiếp cận loss theo error (khoảng cách từ giá trị  $\hat{y} \rightarrow y$ ) thì chúng ta có thể sẽ không train được tối ưu. Vì thế, chúng ta chuyển nó sang một giá trị xác suất gọi là *belief* — định nghĩa là độ tin tưởng của mô hình cho rằng đầu vào sẽ thuộc lớp  $i$ .

Tất nhiên, như định nghĩa, chúng ta muốn:

- Khi  $y = 1$  thì  $\hat{y} \rightarrow 1$
- Khi  $y = 0$  thì  $\hat{y} \rightarrow 0$

Vì belief là một xác suất nên chạy từ  $0 \rightarrow 1$ , ta chuyển từ error sang belief thông qua:

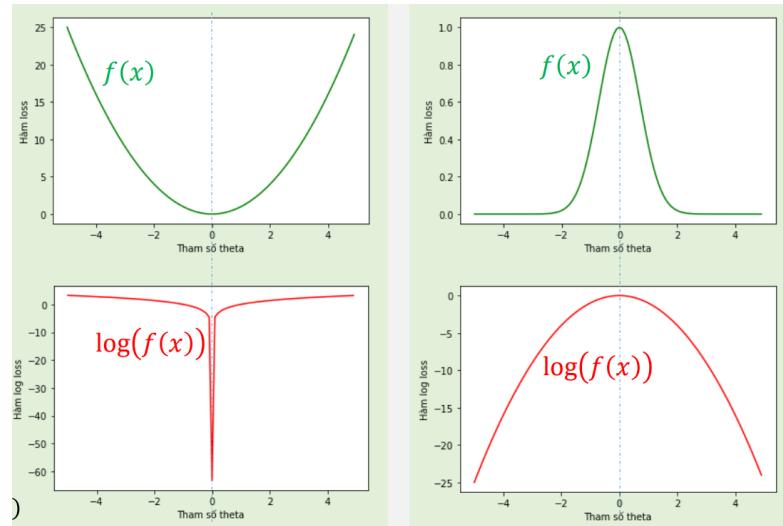
$$\text{belief} = \begin{cases} \hat{y}, & \text{if } y = 1 \\ 1 - \hat{y}, & \text{if } y = 0 \end{cases}$$

Khử rẽ nhánh, ta viết lại:

$$P = \hat{y}^y \cdot (1 - \hat{y})^{1-y}$$

Chúng ta sử dụng log để tiếp tục phân tách hàm này nhờ các tính chất:

- log kéo giá trị hàm xuống nhưng giữ tính đơn điệu của hàm số
- Điểm cực trị không đổi, chi phí tính toán giảm
- Biến tích thành tổng:  $\log(ab) = \log a + \log b$



Hình 17: Hàm loss và log loss

Từ đó ta có:

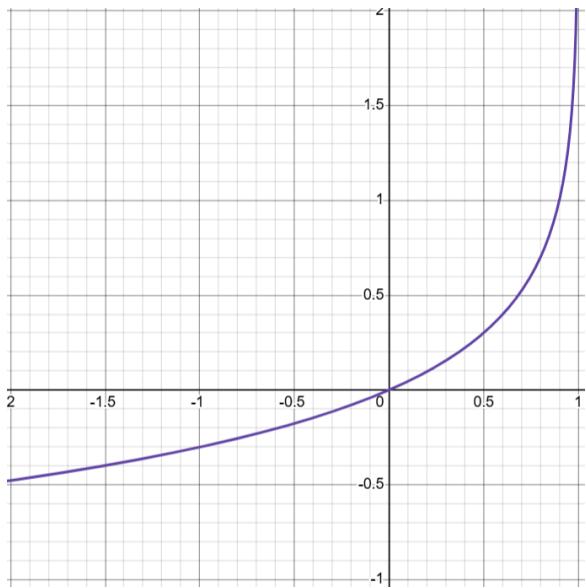
$$\log P = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

Mục tiêu ban đầu:

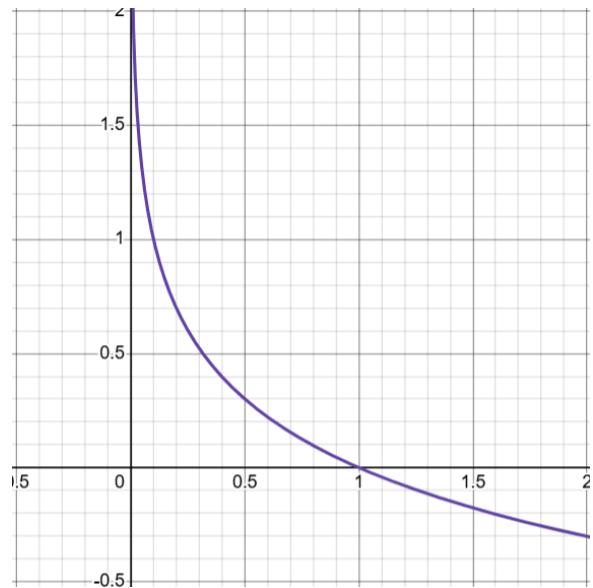
$$\min(\text{error}) = \max(\text{belief}) = \max(\log \text{belief}) = \min(-\log \text{belief})$$

Do đó ta đặt hàm loss mới:

$$\text{Loss} = -\log P = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$



Hình 18: Hàm  $-\log(1 - \hat{y})$



Hình 19: Hàm  $-\log(\hat{y})$

**Khi  $y = 0$ :**

$$\text{Loss} = -\log(1 - \hat{y}) \rightarrow \text{rất lớn nếu } \hat{y} \rightarrow 1$$

**Khi  $y = 1$ :**

$$\text{Loss} = -\log(\hat{y}) \rightarrow \text{rất lớn nếu } \hat{y} \rightarrow 0$$

Ta thấy chúng ta đã xây dựng xong hàm loss theo mong muốn của chúng ta là: khi tiến gần về y thì nó giảm không tăng rất lớn; nhưng chúng ta cần phải trải qua kiểm tra đạo hàm bậc 2 để xác định tính khả thi khi sử dụng hàm loss này

**Bài kiểm tra cuối cùng:** Ta kiểm tra tính convex của hàm loss theo cách tiếp cận mới.

Tính đạo hàm cấp 2 theo ma trận:

$$\frac{\partial \hat{y}}{\partial z} = \hat{y}(1 - \hat{y}) \quad , \quad \frac{\partial z}{\partial \theta_i} = x_i \quad , \quad \frac{\partial L}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_i} = x_i(\hat{y} - y)$$

Đạo hàm bậc hai:

$$\frac{\partial^2 \mathcal{L}}{\partial \theta_i^2} = x_i^2 \cdot \hat{y}(1 - \hat{y}) \geq 0$$

Do đó, hàm loss là **convex** và thích hợp để train dữ liệu.

Ta gọi cách đặt hàm loss như trên là **Binary Cross Entropy**, và nó là lựa chọn phổ biến cho các bài toán phân loại nhị phân.

### 3.4 Tổng quát hóa + Vectorization

Vậy qua những concept mà chúng ta đã tìm hiểu được từ những ví dụ vừa rồi, ta thấy rằng Logistic Regression cũng chỉ được xây dựng dựa trên Linear Regression nhưng chúng ta thêm activation function là sigmoid cho bước logit z sang , do đó kéo theo sự thay đổi

của hàm loss và cách cập nhật đạo hàm, do đó chúng ta sẽ cùng nhau xây dựng 1 công thức tổng quát nhất để có thể giải quyết mọi bài toán Logistic regression

Không mất tính tổng quát, giả sử:

- $m$ : số lượng mẫu trong một batch (batch size)
- $n$ : số tham số (tức là số đặc trưng +1 để tính thêm bias)

Ta định nghĩa:

$$\mathbf{X} \in \mathbb{R}^{m \times n}, \quad \boldsymbol{\theta} \in \mathbb{R}^{n \times 1}$$

Trong đó:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1,n-1} & 1 \\ x_{21} & x_{22} & \cdots & x_{2,n-1} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{m,n-1} & 1 \end{bmatrix}, \quad \boldsymbol{\theta} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{n-1} \\ b \end{bmatrix}$$

Khi đó:

$$\mathbf{z}^{(i)} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix} = \mathbf{X} \cdot \boldsymbol{\theta} \quad (\text{kích thước: } m \times n \cdot n \times 1 = m \times 1)$$

Từ đó, ta tính được :

$$\hat{\mathbf{y}}^{(i)} = \frac{1}{1 + e^{-z_i}} = \sigma(z_i) = \begin{bmatrix} \sigma(z_1) \\ \sigma(z_2) \\ \vdots \\ \sigma(z_m) \end{bmatrix} = \sigma(\mathbf{z}_i) \quad (\text{kích thước: } m \times n \cdot n \times 1 = m \times 1)$$

Vì ta đang dùng minibatch có kích thước  $m$ , nên hàm mất mát sẽ là trung bình của các mất mát đơn lẻ trên từng mẫu:

$$\begin{aligned} \mathcal{L}^{(i)}(\hat{\mathbf{y}}^{(i)}, y^{(i)}) &= \frac{1}{m} \sum_{j=1}^m \mathcal{L}^{(j)}(\hat{\mathbf{y}}^{(j)}, y^{(j)}) \\ &= -\frac{1}{m} \sum_{j=1}^m [y^{(j)} \log(\hat{y}^{(j)}) + (1 - y^{(j)}) \log(1 - \hat{y}^{(j)})] \\ &= -\frac{1}{m} [\mathbf{y}^\top \log(\hat{\mathbf{y}}) + (\mathbf{1} - \mathbf{y})^\top \log(1 - \hat{\mathbf{y}})] \end{aligned}$$

Từ phần chứng minh tính lồi của hàm mất mát, ta đã có đạo hàm với từng mẫu riêng lẻ (Stochastic):

$$\frac{\partial \mathcal{L}}{\partial \theta_i} = x_i(\hat{y} - y)$$

Tuy nhiên, với minibatch kích thước  $m$ , ta cần điều chỉnh một chút và sử dụng đại số tuyến tính. Khi đó, gradient của hàm mất mát theo toàn bộ minibatch là:

$$\nabla_{\theta} \mathcal{L} = \frac{1}{m} X^{\top} (\hat{\mathbf{y}} - \mathbf{y}) \quad (\text{kích thước: } n \times m \cdot m \times 1 = n \times 1 = \text{shape}(\theta))$$

Trong đó:

- $X \in \mathbb{R}^{m \times n}$ : ma trận dữ liệu đầu vào, trong đó mỗi hàng là một mẫu (có  $n$  đặc trưng).
- $\hat{\mathbf{y}} \in \mathbb{R}^{m \times 1}$ : vectơ đầu ra dự đoán, là kết quả áp dụng hàm sigmoid lên đầu ra tuyến tính.
- $\mathbf{y} \in \mathbb{R}^{m \times 1}$ : vectơ nhãn thực tế tương ứng với các mẫu đầu vào.

Cuối cùng, ta cập nhật tham số theo thuật toán Gradient Descent:

$$\theta := \theta - \eta \cdot \nabla_{\theta} \mathcal{L}$$

Trong đó  $\eta$  là learning rate .

**Lưu ý.** Sau một quá trình chứng minh, ta thu được quy trình của 1 bài toán Logistic Regression như sau:

1. Lấy một minibatch gồm  $m$  cặp dữ liệu huấn luyện  $(\mathbf{x}^{(i)}, y^{(i)})$ , với  $i = 1, \dots, m$ .

Tạo ma trận đặc trưng  $\mathbf{X} \in \mathbb{R}^{m \times n}$  bằng cách nối ma trận đặc trưng ban đầu  $\tilde{\mathbf{X}} \in \mathbb{R}^{m \times (n-1)}$  với một cột toàn số 1 (để mô hình hóa bias):

$$\mathbf{X} = [\tilde{\mathbf{X}} \ \mathbf{1}]$$

2. Compute output  $\hat{y}$

$$z = \mathbf{X}\theta, \quad \hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$

3. Compute loss

$$L(y, \hat{y}) = \frac{1}{m} \left( -y^{\top} \log \hat{y} - (1 - y)^{\top} \log(1 - \hat{y}) \right)$$

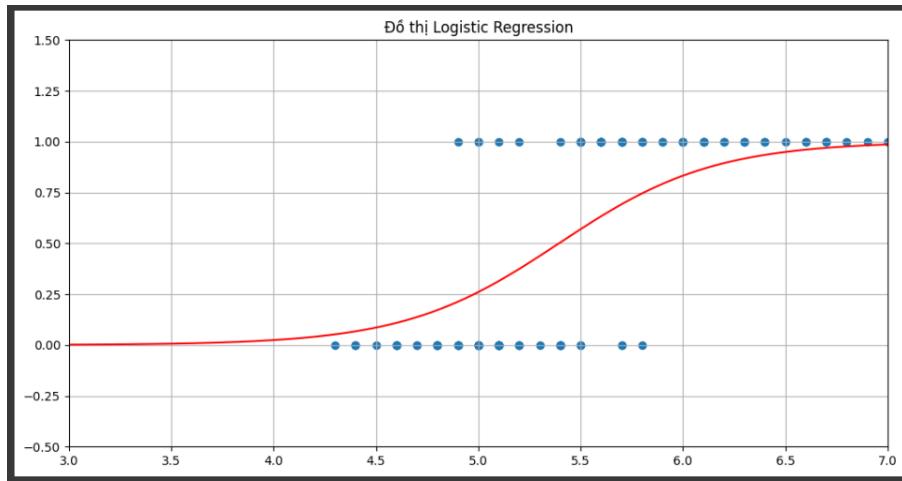
4. Compute nabla

$$\nabla_{\theta} L = \frac{1}{m} X^{\top} (\hat{y} - y)$$

5. Cập nhật thông số

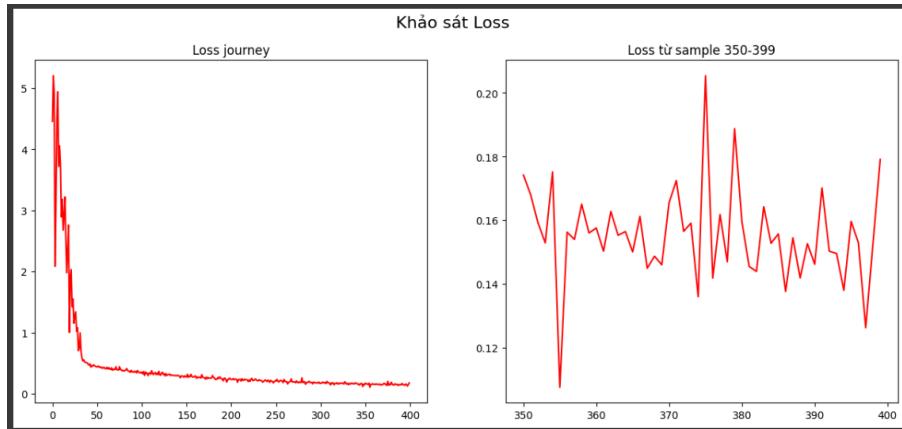
$$\theta := \theta - \eta \nabla_{\theta} L$$

### 3.5 Implementation



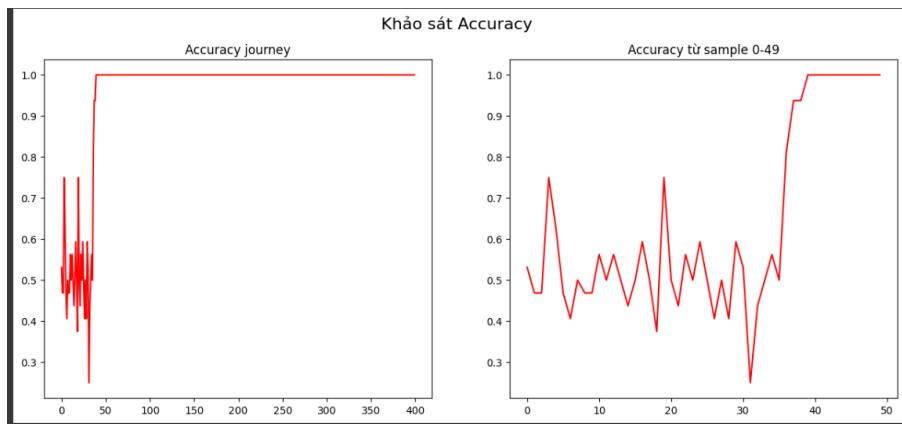
Hình 20: Đường biên phân lớp của Logistic Regression trên tập dữ liệu Iris\_2D

**Nhận xét:** Với bài toán con đơn giản nhất, tôi cố tình lựa chọn hai đặc trưng (feature) có corelation thấp nhất để thể hiện độ "hard" của cách linear regression. Tiếp theo, mô hình được huấn luyện trên bài toán tổng quát với 4 features và 2 label của iris\_2D dataset. Kết quả thu được thể hiện qua loss và accuracy như sau:



Hình 21: Biểu đồ hàm mất mát (loss) của mô hình theo số epoch

**Nhận xét:** Hàm mất mát giảm rất nhanh trong giai đoạn đầu, và sau khoảng 350 epoch thì giá trị loss chỉ còn khoảng 0.2 — tương đối nhỏ, cho thấy mô hình hội tụ tốt.



Hình 22: Biểu đồ độ chính xác (accuracy) của mô hình theo số epoch

**Nhận xét:** Độ chính xác của mô hình tăng nhanh và đạt cực đại (1.0) ngay từ khoảng epoch thứ 50. Điều này cho thấy mô hình có khả năng học rất nhanh trên tập dữ liệu đơn giản, đồng thời phản ánh tính phân lớp rõ rệt trong không gian đặc trưng của dữ liệu.

Source code (Google Colab): [Logistic Regression](#)

## 4 SoftMax Regression

---

### 4.1 Motivation

Sau khi giải quyết được vấn đề nan giải bấy lâu, nhà thực vật học quyết định thưởng cho mình một chuyến đi du lịch. Trong chuyến đi này, nhà thực vật học đã phát hiện ra một giống hoa *Iris* mới và thu thập rất nhiều đặc trưng (feature) về loại hoa này để đem về làm quà lưu niệm cho bạn. Dataset của *Iris* được cập nhật như sau:

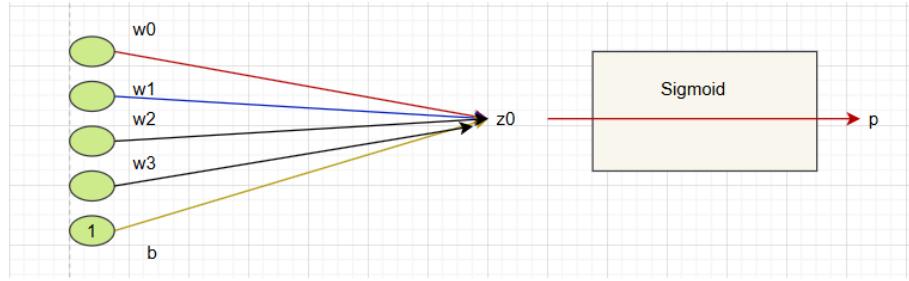
(150, 5)					
	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	5.1	3.5	1.4	0.2	0
10	5.4	3.7	1.5	0.2	0
20	5.4	3.4	1.7	0.2	0
30	4.8	3.1	1.6	0.2	0
40	5.0	3.5	1.3	0.3	0
50	7.0	3.2	4.7	1.4	1
60	5.0	2.0	3.5	1.0	1
70	5.9	3.2	4.8	1.8	1
80	5.5	2.4	3.8	1.1	1
90	5.5	2.6	4.4	1.2	1
100	6.3	3.3	6.0	2.5	2
110	6.5	3.2	5.1	2.0	2
120	6.9	3.2	5.7	2.3	2
130	7.4	2.8	6.1	1.9	2
140	6.7	3.1	5.6	2.4	2

Hình 23: Dataset mới của Iris sau khi bỏ sung loài thứ ba

Bạn nhận ra bài toán đã trở nên phức tạp hơn vì có sự xuất hiện của loại thứ ba , khiến cho việc sử dụng hàm sigmoid trở nên khó khăn và có thể không còn chính xác cao khi dùng phương pháp *one-vs-rest*, vì sigmoid chỉ biểu diễn xác suất nhị phân (0 và 1). Khi bài toán có đến ba loại, hoặc tổng quát hơn là  $k$  loại, thì hàm sigmoid bộc lộ rõ hạn chế. Vậy nên, chúng ta cần tìm một *activation function* mới giúp lựa chọn một xác suất phù hợp từ nhiều giá trị đầu ra ban đầu.

### 4.2 Hướng tiếp cận và Bài toán tổng quát

Chúng ta sẽ bắt đầu xây dựng một mô hình Logistic Regression như đã đề cập ở trên:



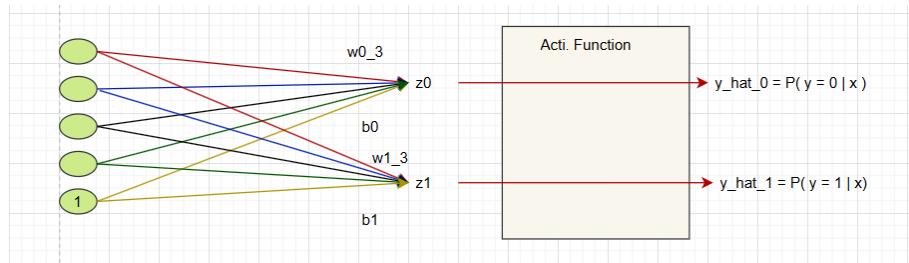
Hình 24: Mô hình bài toán Logistic Regression cho Iris\_2D dataset

Ta sẽ bàn thêm một chút về công thức Binary Cross-Entropy (BCE):

$$\text{Loss} = -\log P = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

Từ ý tưởng xây dựng trên niềm tin (belief), ta có thể hiểu  $\hat{y}$  là xác suất mà  $x$  (giá trị đầu vào) thuộc về lớp 1, tức là  $P(y = 1 | x)$ . Đương nhiên  $1 - \hat{y}$  sẽ là xác suất  $x$  thuộc về lớp 0, tức là  $P(y = 0 | x)$ . Giá trị 1 là do tổng xác suất bằng 1 và mô hình này chỉ có một node đầu ra  $\hat{y}$ , do đó ta chỉ có một con số để biểu diễn cho cả hai trường hợp.

Ý tưởng tối ưu (Optimize) của bài toán là đi tìm giá trị tham số sao cho xác suất  $\hat{y}$  tiến gần về 1. Tuy nhiên, theo cách xây dựng trên, các trọng số chỉ được thay đổi sao cho tốt cho xác suất  $P(\text{label} = 1 | x)$ , còn lớp 0 thì “dùng chung” trọng số đó nên chỉ được tính là ăn theo. Vậy, hiểu được ý tưởng như trên, chúng ta muốn tính xác suất của từng lớp và sau đó so sánh — tương tự như ý tưởng trong bài toán *Naive Bayes*. Nếu làm được điều đó, chúng ta hoàn toàn có thể mở rộng sang bài toán nhiều loại (multi-class classification). Chúng ta sẽ xây dựng lại mô hình như sau:



Hình 25: Mô hình Softmax Regression với nhiều node đầu ra

Chúng ta sẽ thêm nhiều node đầu ra, trong đó nhiệm vụ của mỗi node là xây dựng một bộ trọng số riêng chỉ để tối ưu cho category nhất định. Ví dụ, node  $z_0$  sẽ tối ưu cho class 0 và  $z_1$  tối ưu cho class 1. Khi mỗi node có tham số riêng, chúng chỉ cần tập trung xử lý và học bộ tham số của mình.

Tuy nhiên, trong mô hình này, các node đầu ra sẽ tạo thành một vector  $\mathbf{z} = [z_0, z_1, \dots, z_{k-1}]$ . Chúng ta mong muốn đầu ra của mô hình là một vector xác suất tương ứng — đại diện cho xác suất mẫu dữ liệu thuộc về từng class — từ đó giúp đưa ra quyết định phân loại chính xác nhất.

#### 4.2.1 Activation Function: SoftMax

Chúng ta cần một mô hình xác suất sao cho với mỗi đầu vào  $\mathbf{x}$ , giá trị  $\hat{y}_i$  thể hiện xác suất để  $\mathbf{x}$  thuộc về lớp  $i$ . Do đó, các điều kiện cần thiết là:

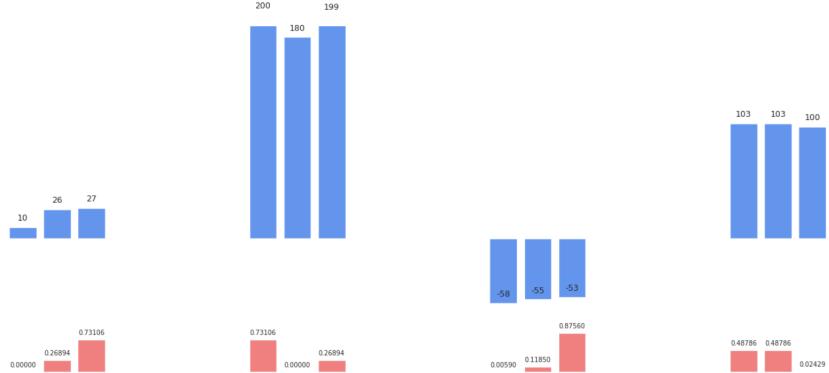
- $\hat{y}_i > 0$  với mọi  $i$

- $\sum_{i=1}^k \hat{y}_i = 1$

Để thỏa mãn các điều kiện trên, ta cần một cách chuyển đổi các giá trị  $z_i$  (đầu ra tuyến tính từ mô hình  $z_i = \mathbf{w}_i^T \mathbf{x}$ ) thành các xác suất  $\hat{y}_i$ . Bên cạnh hai điều kiện nêu trên, ta cũng yêu cầu thêm một tính chất tự nhiên khác. "Giá trị  $z_i$  càng lớn thì xác suất  $a_i$  càng cao."

Tính chất này gợi ý rằng act.func cần là một hàm đồng biến theo  $z_i$ . Tuy nhiên, vì  $z_i$  có thể nhận giá trị âm hoặc dương, ta cần một hàm biến đổi smooth để thuận tiện cho việc lấy đạo hàm trong quá trình huấn luyện. Một lựa chọn hợp lý là hàm mũ:  $e^{z_i}$ , cũng là để tối đa sự chênh lệch Hàm số này luôn dương và đồng biến theo  $z_i$ . Để đảm bảo tổng tất cả các xác suất  $\hat{y}_i$  bằng 1, ta chuẩn hóa các giá trị này như sau:

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^k e^{z_j}}, \quad \text{với } i = 1, 2, \dots, k$$



Hình 26: Output của hàm softMax

Hàm số trên được gọi là **hàm SoftMax**. Hàm này có các đặc tính sau:

- $\hat{y}_i > 0$  với mọi  $i$ ,

- $\sum_{i=1}^k \hat{y}_i = 1$ ,

- Nếu  $z_i > z_j$  thì  $\hat{y}_i > \hat{y}_j$ .

Với cách định nghĩa này, không có xác suất nào tuyệt đối bằng 0 hoặc 1, mặc dù một số giá trị có thể rất gần 0 hoặc rất gần 1 nếu  $z_i$  khác biệt lớn so với các  $z_j$  còn lại.

#### 4.2.2 Loss Function: Cross Entropy

Ở bài toán BCE, ta biết rằng nó chỉ được xây dựng với 1 node nên giá trị xác suất của 1 class sẽ ăn theo class còn lại, ta có:

$$\hat{y} = P(y = 1 | \mathbf{x}), \quad 1 - \hat{y} = P(y = 0 | \mathbf{x})$$

Tuy nhiên, trong cách thiết lập tổng quát hơn với bài toán phân loại đa lớp và concept của SoftMax Function, mô hình sẽ sinh ra một vector  $\hat{\mathbf{y}} = [\hat{y}_1, \hat{y}_2, \dots, \hat{y}_k]$  với  $k$  là số lượng

lớp. Mỗi phần tử  $\hat{y}_i$  là xác suất dự đoán rằng  $\mathbf{x}$  thuộc lớp  $i$ . Do đó, xác suất của một lớp bất kỳ không phụ thuộc vào giá trị label của lớp khác, và hàm mất mát cần được tổng quát hóa từ công thức Binary Cross Entropy thành **Cross Entropy** dành cho nhiều lớp:

$$\text{Loss} = -\log P = -y_1 \log(\hat{y}_1) - y_0 \log(\hat{y}_0)$$

Công thức trên vẫn chỉ áp dụng cho bài toán nhị phân với hai lớp. Để xử lý tốt hơn các bài toán phân loại với nhiều lớp, một phương pháp phổ biến là sử dụng biểu diễn nhãn đầu ra dưới dạng **one-hot encoding** thay vì các giá trị rời rạc.

**One-Hot Encoding** Trong cách biểu diễn này, thay vì dùng một số nguyên duy nhất để đại diện cho lớp (ví dụ  $y = 1$ ), ta dùng một vector nhị phân có độ dài bằng số lượng lớp  $k$ , trong đó chỉ có một phần tử bằng 1 tương ứng với lớp đúng, còn lại là 0.

$$\text{Ví dụ: } y = 1 \Rightarrow \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Tương ứng, đầu ra  $\hat{\mathbf{y}}$  là một vector xác suất từ hàm SoftMax:

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \hat{y}_2 \end{bmatrix}$$

Với cách biểu diễn này, việc tính toán hàm mất mát trở nên thuận lợi hơn khi sử dụng đại số tuyến tính.

**Loss Function (tổng quát hóa):** Hàm mất mát tổng quát được định nghĩa như sau:

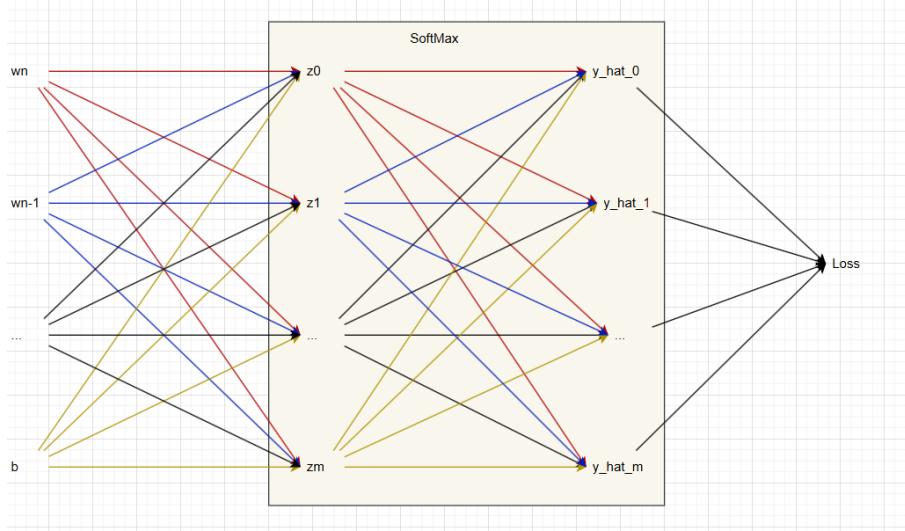
$$\text{Loss} = - \sum_{i=1}^k y_i \log(\hat{y}_i) = -\mathbf{y}^T \log(\hat{\mathbf{y}})$$

Trong đó:

- $\mathbf{y}$  là vector one-hot biểu diễn nhãn đúng (ground truth),
- $\hat{\mathbf{y}}$  là vector xác suất đầu ra sau khi qua hàm SoftMax.

Hàm mất mát Cross Entropy đo lường độ sai khác giữa phân phối xác suất dự đoán và phân phối thực tế, và là một trong những hàm mất mát phổ biến nhất trong các bài toán phân loại nhiều lớp.

### 4.3 Tổng quát hóa + Vectorization



Hình 27: Mô hình cho bài toán SoftMax Regression tổng quát

Không mất tính tổng quát, giả sử:

- \$m\$: số lượng mẫu trong một batch (batch size)
- \$n\$: số đặc trưng đầu vào (features), cộng thêm +1 để tính bias
- \$k\$: số lượng lớp (categories) trong nhãn đầu ra

Khi đó:

$$\mathbf{X} \in \mathbb{R}^{m \times n}, \quad \boldsymbol{\Theta} \in \mathbb{R}^{n \times k}$$

Với:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1,n-1} & 1 \\ x_{21} & x_{22} & \cdots & x_{2,n-1} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{m,n-1} & 1 \end{bmatrix} \quad (\text{thêm } 1 \text{ ở cuối để tính bias})$$

Ta cần xây dựng ma trận trọng số \$\boldsymbol{\Theta}\$, trong đó mỗi cột tương ứng với trọng số và bias của một lớp:

$$\theta_i = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,n-1} \\ b_i \end{bmatrix} \quad \text{với } i = 1, 2, \dots, k$$

Khi đó ma trận tổng thể là:

$$\boldsymbol{\Theta} = [\theta_1 \quad \theta_2 \quad \cdots \quad \theta_k] = \begin{bmatrix} w_{1,1} & w_{2,1} & \cdots & w_{k,1} \\ w_{1,2} & w_{2,2} & \cdots & w_{k,2} \\ \vdots & \vdots & \ddots & \vdots \\ b_1 & b_2 & \cdots & b_k \end{bmatrix} \in \mathbb{R}^{n \times k}$$

Bước tính toán tiếp theo là tính tích:

$$\mathbf{Z} = \mathbf{X} \cdot \Theta \quad (\text{kích thước: } m \times n \cdot n \times k = m \times k)$$

Trong đó,  $\mathbf{Z}_{ij}$  chính là logit (đầu vào chưa chuẩn hoá) của mẫu thứ  $i$  tại lớp thứ  $j$ .

Sau khi tính được đầu ra  $\mathbf{Z} = \mathbf{X} \cdot \Theta \in \mathbb{R}^{m \times k}$ , ta tiến hành chuẩn hóa đầu ra này bằng hàm Softmax, áp dụng theo từng hàng của ma trận đầu ra:

$$\hat{y}_{ij} = \frac{e^{z_{ij}}}{\sum_{t=1}^k e^{z_{it}}} \quad \text{với } i = 1, \dots, m \text{ và } j = 1, \dots, k$$

Kết quả thu được là một ma trận  $\hat{\mathbf{Y}} \in \mathbb{R}^{m \times k}$ , trong đó mỗi hàng tương ứng với một mẫu trong batch, và là một vector xác suất, tổng các phần tử trên mỗi hàng bằng 1:

$$\sum_{j=1}^k \hat{y}_{ij} = 1 \quad \forall i = 1, \dots, m$$

Tiếp theo, giả sử vector nhãn  $\mathbf{y}$  đã được one-hot hoá, tức là:

$$\mathbf{y}_{ij} = \begin{cases} 1 & \text{nếu mẫu } i \text{ thuộc lớp } j \\ 0 & \text{ngược lại} \end{cases} \Rightarrow \mathbf{Y} \in \mathbb{R}^{m \times k}$$

Khi đó, Loss function cho toàn bộ batch sẽ là trung bình của Binary Cross-Entropy (BCE) giữa nhãn thật và xác suất dự đoán:

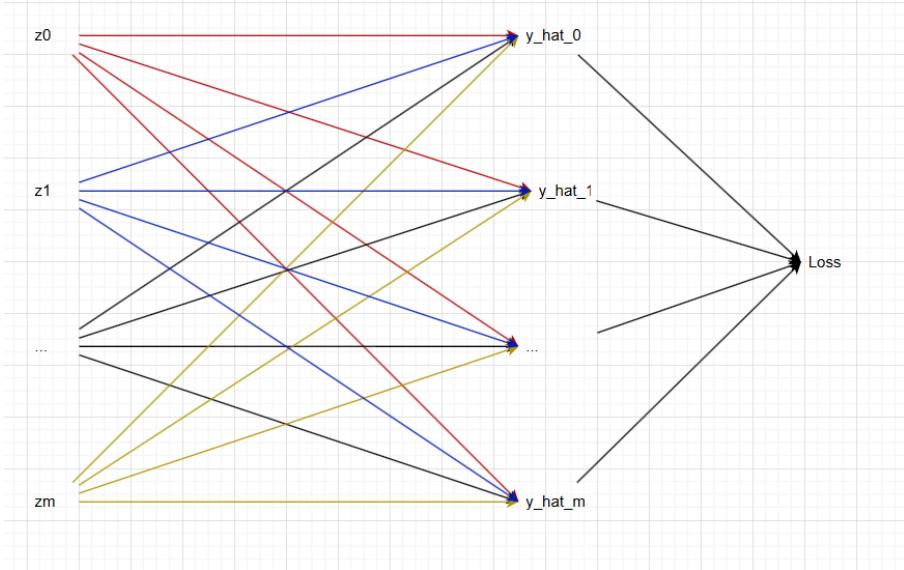
$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m \sum_{j=1}^k y_{ij} \log(\hat{y}_{ij})$$

Hoặc có thể viết gọn hơn dưới dạng tích vô hướng:

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m \mathbf{y}_i^T \log(\hat{\mathbf{y}}_i)$$

**Tính đạo hàm của Loss theo đầu ra trước softmax ( $\mathbf{Z}$ )** Ta cần tính gradient của loss theo ma trận  $\mathbf{Z}$ , tức là  $\frac{\partial \mathcal{L}}{\partial \mathbf{Z}}$ . Chú ý rằng do softmax là một hàm nhiều biến (vector function), nên việc đạo hàm không đơn giản như các ví dụ trước. Mỗi phần tử  $z_{ij}$  ảnh hưởng đến toàn bộ vector  $\hat{\mathbf{y}}_i$ , do đó cần xét tất cả các đường dẫn đạo hàm từ  $L \rightarrow \hat{y}_{ik} \rightarrow z_{ij}$ .

Cây đạo hàm như sau (hình minh họa):



Hình 28: Cây đạo hàm với Softmax

$$\begin{aligned}\frac{\partial L}{\partial z_i} &= \sum_{k=1}^K \frac{\partial L}{\partial \log(\hat{y}_k)} \frac{\partial \log(\hat{y}_k)}{\partial z_i} \\ &= \sum_{k=1}^K \cdot \frac{-y_k}{\hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial z_i}\end{aligned}$$

Chúng ta sẽ tính đạo hàm của  $\frac{\partial \hat{y}_k}{\partial z_i}$

- Với  $i \neq j$ : đạo hàm dạng  $(1/u)'$

$$\frac{\partial \hat{y}_i}{\partial z_j} = -\frac{e^{z_i}}{\left(\sum_j e^{z_j}\right)^2} \cdot e^{z_j} = -\hat{y}_i \hat{y}_j$$

- Với  $i = j$ : đạo hàm dạng  $(u/v)'$

$$\frac{\partial \hat{y}_i}{\partial z_i} = \frac{e^{z_i} \cdot \sum_j e^{z_j} - e^{z_i} \cdot e^{z_i}}{\left(\sum_j e^{z_j}\right)^2} = \hat{y}_i(1 - \hat{y}_i)$$

Thay vào từng trường hợp:

$$\begin{aligned}
\frac{\partial L}{\partial z_i} &= -y_i(1 - \hat{y}_i) - \sum_{k \neq i} \frac{-y_k}{\hat{y}_k}(-\hat{y}_k \hat{y}_i) \\
&= -y_i(1 - \hat{y}_i) + \sum_{k \neq i} y_k \hat{y}_i \\
&= -y_i + y_i \hat{y}_i + \sum_{k \neq i} y_k \hat{y}_i \\
&= \hat{y}_i \left( y_i + \sum_{k \neq i} y_k \right) - y_i \\
&= \hat{y}_i - y_i
\end{aligned}$$

Vậy ta suy ra đạo hàm tổng quát của loss theo ma trận đầu ra trước softmax:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Z}} = \hat{\mathbf{Y}} - \mathbf{Y} \in \mathbb{R}^{m \times k}$$


---

Ta có đầu ra trước softmax là:

$$\mathbf{Z} = \mathbf{X}\Theta \quad \text{với } \mathbf{X} \in \mathbb{R}^{m \times n}, \quad \Theta \in \mathbb{R}^{n \times k}$$

Với mẫu thứ  $i$ , ta có:

$$\frac{\partial z^{(i)}}{\partial \theta^{(i)}} = \mathbf{x}^{(i)}$$

**Suy ra gradient toàn bộ theo ma trận trọng số:**

$$\nabla_{\Theta} L = \mathbf{X}^T (\hat{\mathbf{Y}} - \mathbf{Y}) \quad (\text{vì } \hat{\mathbf{Y}} - \mathbf{Y} \in \mathbb{R}^{m \times k}, \mathbf{X}^T \in \mathbb{R}^{n \times m})$$

$$\Rightarrow \nabla_{\Theta} L \in \mathbb{R}^{n \times k} = \text{shape của } \Theta$$

Ta đã hoàn thành bước lan truyền ngược (backpropagation) để tính gradient theo trọng số. Sau đó, thuật toán sẽ cập nhật lại các giá trị của  $\Theta$ :

$$\Theta := \Theta - \eta \cdot \nabla_{\Theta} \mathcal{L}$$

Trong đó  $\eta$  là learning rate.

**Lưu ý.** Sau một quá trình chứng minh, ta thu được quy trình của 1 bài toán Logistic Regression như sau:

1. Lấy một minibatch gồm  $m$  cặp dữ liệu huấn luyện  $(\mathbf{x}^{(i)}, y^{(i)})$ , với  $i = 1, \dots, m$ .

Tạo ma trận đặc trưng  $\mathbf{X} \in \mathbb{R}^{m \times n}$  bằng cách nối ma trận đặc trưng ban đầu  $\tilde{\mathbf{X}} \in \mathbb{R}^{m \times (n-1)}$  với một cột toàn số 1 (để mô hình hóa bias):

$$\mathbf{X} = [\tilde{\mathbf{X}} \ 1]$$

2. Compute output  $\hat{y}$

$$\mathbf{Z} = \mathbf{X} \cdot \Theta, \quad \hat{\mathbf{Y}} = \text{softmax}(\mathbf{Z})$$

3. Compute loss

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^m \mathbf{y}_i^T \log(\hat{\mathbf{y}}_i)$$

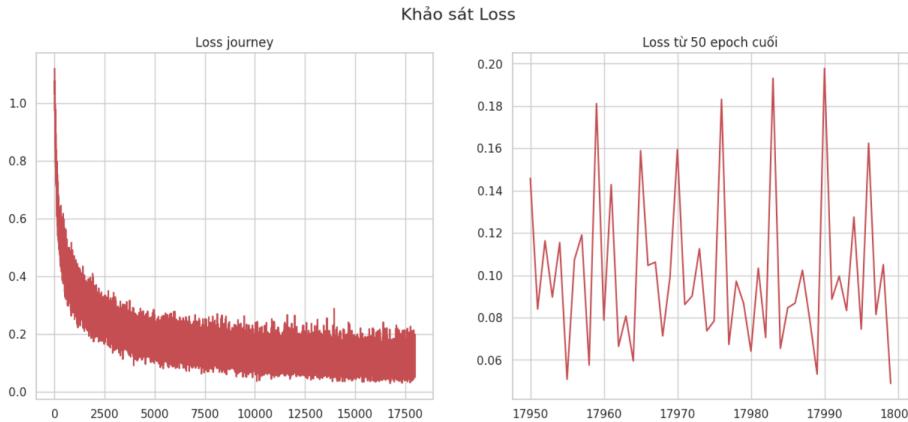
4. Compute nabla

$$\nabla_{\Theta} \mathcal{L} = \mathbf{X}^T (\hat{\mathbf{Y}} - \mathbf{Y})$$

5. Cập nhật thông số

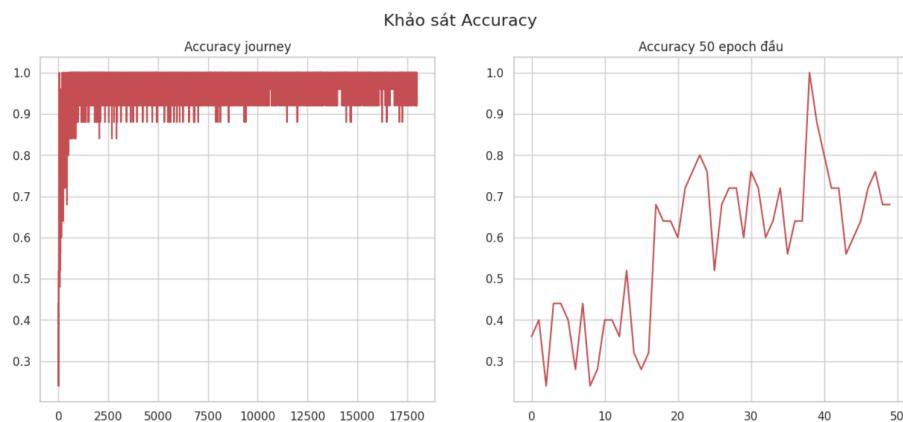
$$\Theta := \Theta - \eta \cdot \nabla_{\Theta} \mathcal{L}$$

#### 4.4 Implementation



Hình 29: Khảo sát hàm loss của bài toán SoftMax

**Nhận xét:** Hàm mất mát giảm rất nhanh trong giai đoạn đầu, và trong 50 epoch cuối thì giá trị loss chỉ còn khoảng 0.2 — tương đối nhỏ, cho thấy mô hình hội tụ tốt.



Hình 30: Khảo sát accuracy

**Nhận xét:** Độ chính xác của mô hình tăng nhanh và dao động quang vùng cực đại ngay từ khoảng epoch thứ 50. Điều này cho thấy mô hình có khả năng học dữ liệu tuy

nhiên khi tăng số category lên thì mô hình dễ bị "dao động", đồng thời phản ánh tính phân lớp rõ rệt trong không gian đặc trưng của dữ liệu.

**Source code (Google Colab):** [SoftMax Regression](#)

## 5 MLP

### 5.1 Motivation and Datasets

Có sourcode chứng minh limitation của SoftMax ở cuối mục 5.1, các thực nghiệm ấy sẽ được trình bày trong 5.1.2 và 5.1.3

#### 5.1.1 Cách tiếp cận thứ nhất: Tập dữ liệu phân bố theo hàm XOR

Nội dung mục 5.1.1 tham khảo từ: [Machine Learning Cơ bản](#)

**Các hàm tuyến tính phân biệt được (linearly separable):**

Các hàm logic như NOT, AND, OR, cũng như các tập dữ liệu như `iris_2class`, đều là các bài toán phân loại tuyến tính — tức tồn tại một siêu phẳng (hyperplane) có thể phân tách hai lớp một cách rõ ràng. Ví dụ, bảng chân trị của hàm AND:

$x_1$	$x_2$	$Y$
0	0	0
0	1	0
1	0	0
1	1	1

Dễ thấy rằng chỉ có điểm  $(1, 1)$  cho đầu ra là 1, do đó ta có thể dễ dàng vẽ một đường thẳng tách biệt điểm này khỏi ba điểm còn lại. Với hàm kích hoạt đơn giản như hàm dấu:

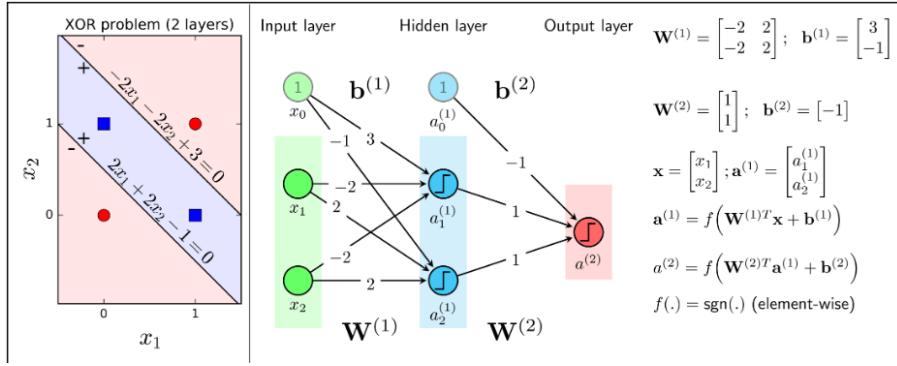
$$a = \text{sgn}(w^T x + b),$$

mô hình PLA có thể tìm được các trọng số  $w$  và độ dời  $b$  thỏa điều kiện phân biệt hai lớp. **Hàm XOR không tuyến tính phân biệt được:**

Bảng chân trị của XOR:

$x_1$	$x_2$	$Y$
0	0	0
0	1	1
1	0	1
1	1	0

Nếu biểu diễn các điểm này trên mặt phẳng 2D, không tồn tại một đường thẳng nào có thể phân chia chính xác hai lớp 0 và 1. Điều này cho thấy bài toán là *không tuyến tính phân biệt được*. Để giải quyết bài toán này, ta cần giới thiệu một tầng ẩn (hidden layer) với các nút trung gian  $a_1, a_2$  đóng vai trò trung gian xử lý đặc trưng, trước khi chuyển kết quả đến một đơn vị PLA khác ở tầng output. Tổng thể mô hình này được gọi là **Multi-layer Perceptron (MLP)** — gồm một tầng input, ít nhất một tầng ẩn và một tầng output.



Hình 31: Mô hình MLP biểu diễn hàm XOR

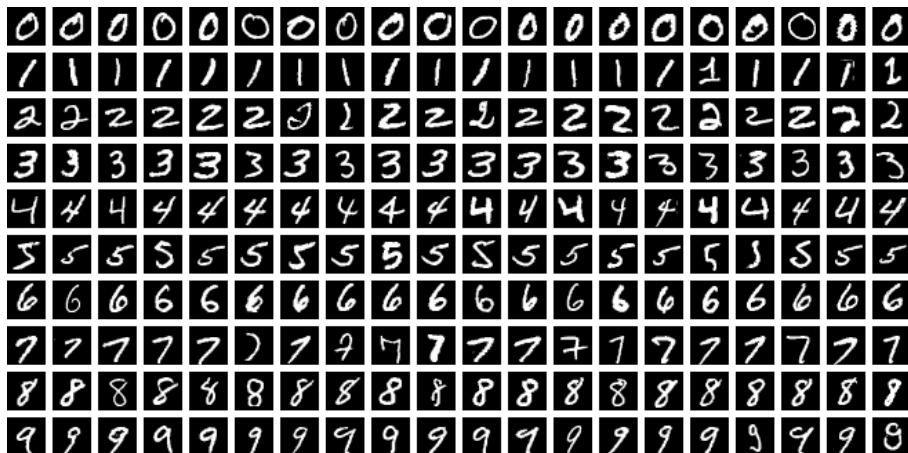
## 2.

### 5.1.2 Tiếp cận 2

Sau khi bạn đã giải quyết bài toán phân loại hoa Iris cho nhà thực vật học, bạn trở nên nổi tiếng và lọt vào mắt xanh của giới tinh anh. Một ngày nọ, một *shark* đến ngỏ lời muốn bạn giúp giải quyết vấn đề cho tập đoàn triệu đô của anh ta. Được biết doanh nghiệp của anh chuyên sản xuất thời trang, tuy nhiên do khâu quản lý yếu kém của đơn vị logistics, các mặt hàng bị ứ đọng và lắn lộn với nhau, dẫn đến việc tồn nhiều thời gian phân loại trước khi đến tay khách hàng.

Bằng một cách nào đó, doanh nghiệp đã thu thập được ảnh xám của 10 loại mặt hàng, mỗi loại gồm 6000 tấm ảnh kích thước  $28 \times 28$ . Doanh nghiệp mong muốn bạn huấn luyện một mô hình sao cho khi đưa một ảnh mới vào, hệ thống có thể nhận diện đúng loại mặt hàng và phân loại vào 10 khu vực riêng biệt để tránh tình trạng lắn lộn như hiện tại

Bài toán ở trên là cách diễn đạt sinh động của bài toán phân loại dữ liệu **Fashion-MNIST**, một biến thể của tập dữ liệu MNIST: tập các ảnh  $28 \times 28$  chứa chữ số từ 0 đến 9, dễ phân biệt bằng mắt thường. Tuy nhiên, với máy tính, mỗi ảnh chỉ là một ma trận  $28 \times 28$  chứa các giá trị `uint8` và nhiệm vụ của ta là huấn luyện mô hình sao cho máy có thể hiểu và phân loại được dữ liệu này.



Hình 32: Dataset MNIST

**Cách tổ chức dataset** Vì máy tính hoạt động hiệu quả hơn với dữ liệu một chiều trong RAM, nên các ảnh sẽ được *flatten* thành vector một chiều. Với ảnh  $28 \times 28$ , ta sẽ cần một mảng 784 byte. Tuy nhiên, nếu không nhớ kích thước ban đầu hoặc chia sẻ cho người khác, rất khó để xác định lại kích thước ảnh. Do đó, người ta đã thêm một phần đầu mô tả cấu trúc dữ liệu như sau:

- **4 byte**: *magic number* – dùng để xác định định dạng file
- **4 byte**: số lượng ảnh
- **4 byte**: số dòng (chiều cao ảnh)
- **4 byte**: số cột (chiều rộng ảnh)

Từ byte thứ 16 trở đi là dữ liệu ảnh.

Đối với phần nhãn (*label*) – vì chỉ là văn bản – nên phần header chỉ gồm:

- **4 byte**: *magic number*
- **4 byte**: số lượng nhãn

Các byte còn lại chứa thông tin nhãn tương ứng.

---

### TRAINING SET LABEL FILE (train-labels-idx1-ubyte):

```
[offset] [type]      [value]      [description]
0000    32 bit integer  0x00000801(2049)  magic number (MSB first)
0004    32 bit integer  60000          number of items
0008    unsigned byte   ??             label
0009    unsigned byte   ??             label
.....
xxxx    unsigned byte   ??             label

The labels values are 0 to 9.
```

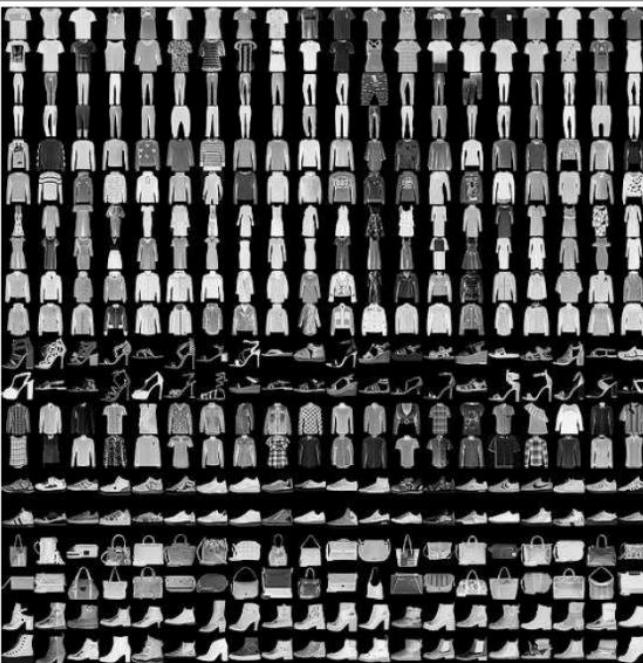
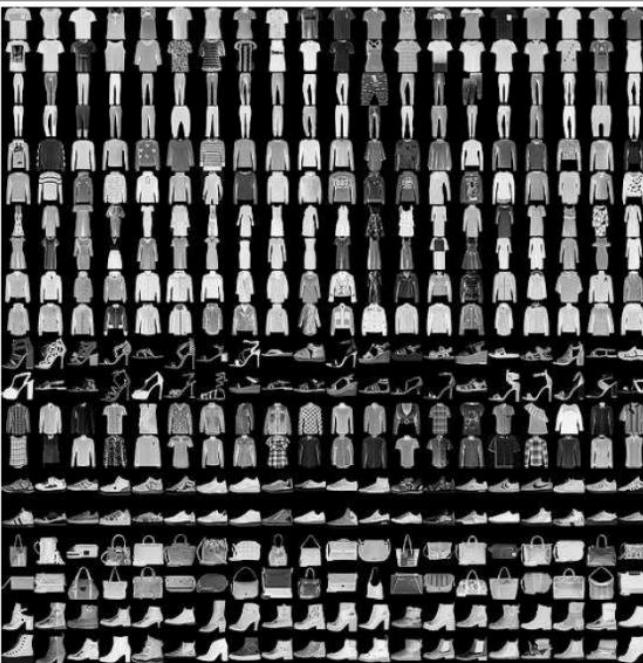
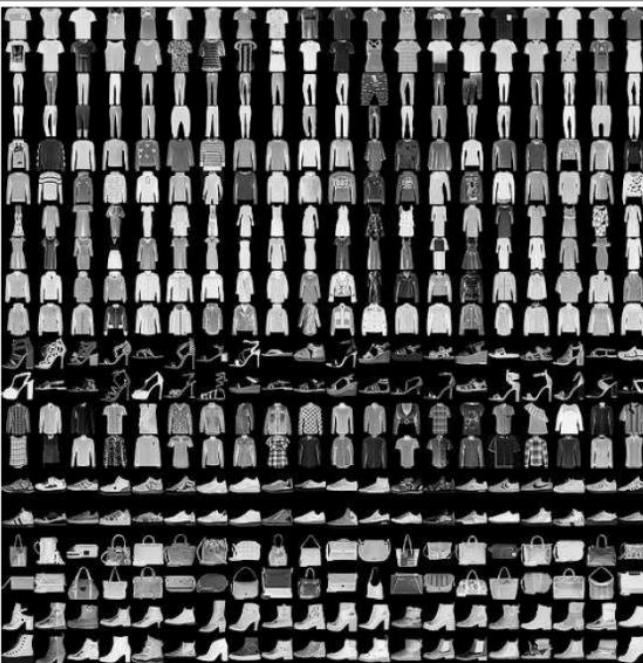
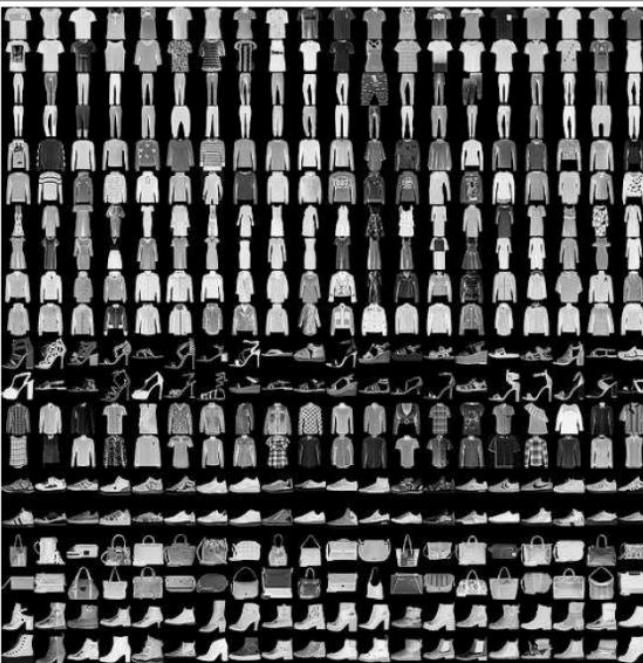
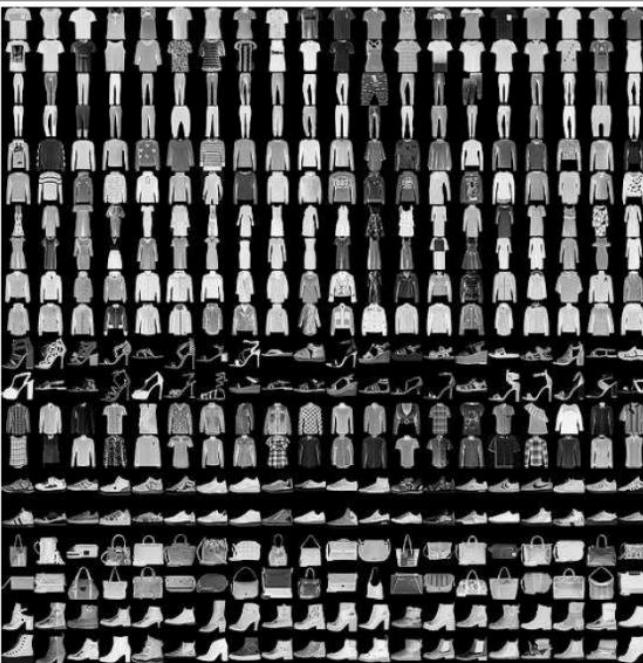
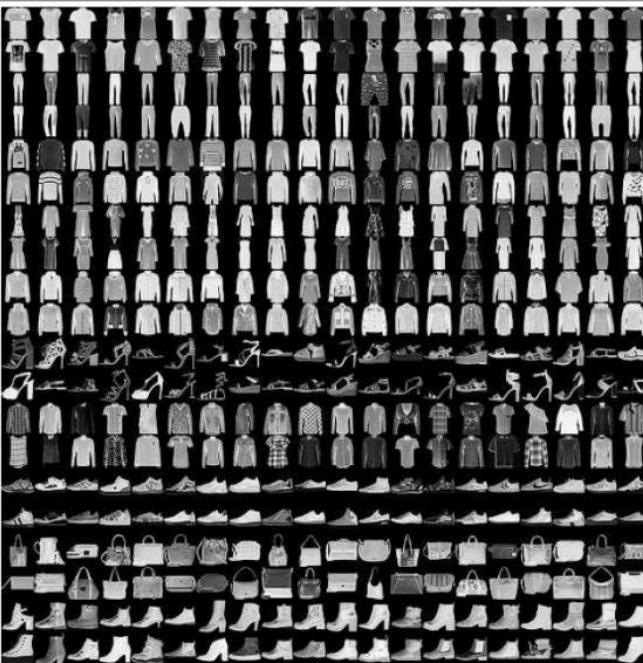
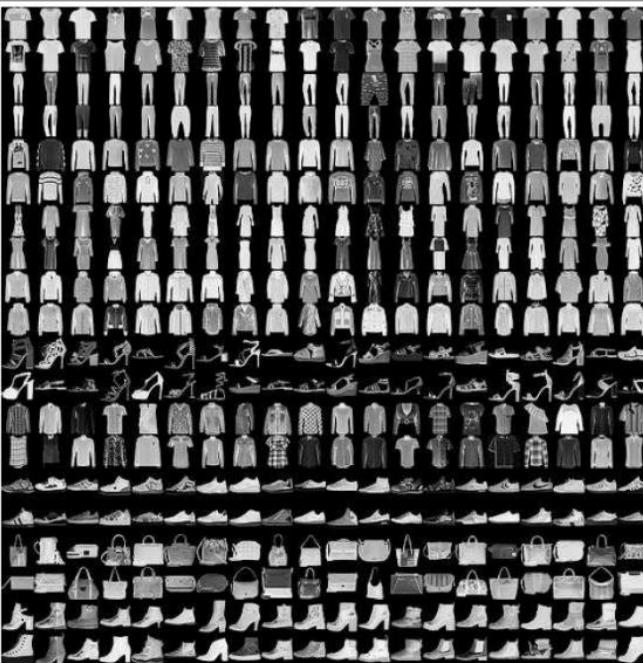
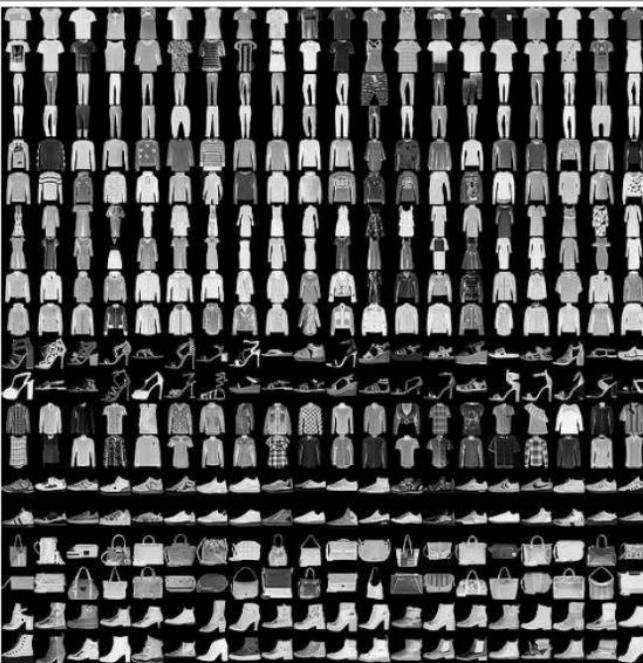
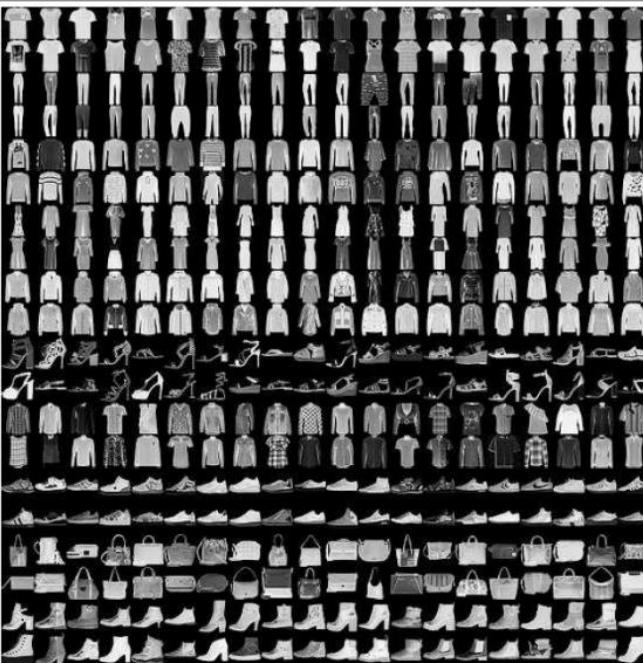
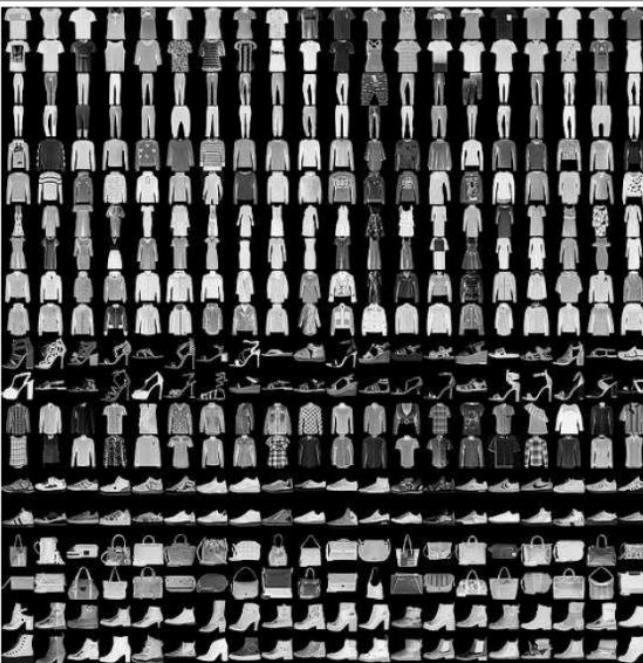
### TRAINING SET IMAGE FILE (train-images-idx3-ubyte):

```
[offset] [type]      [value]      [description]
0000    32 bit integer  0x00000803(2051)  magic number
0004    32 bit integer  60000          number of images
0008    32 bit integer  28              number of rows
0012    32 bit integer  28              number of columns
0016    unsigned byte   ??             pixel
0017    unsigned byte   ??             pixel
.....
xxxx    unsigned byte   ??             pixel
```

---

Hình 33: Cách tổ chức dataset MNIST

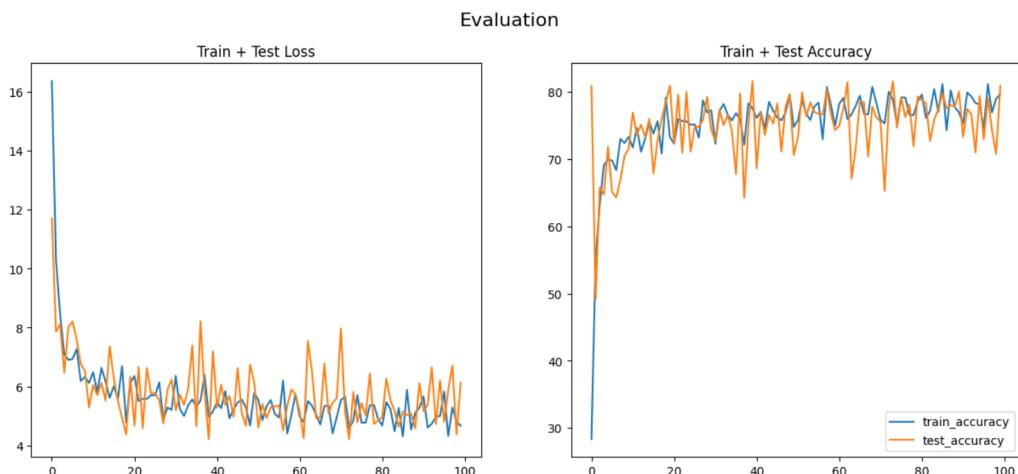
Tuy nhiên, người ta cho rằng tập dữ liệu chỉ gồm chữ số là quá nhảm chán. Do đó, họ đã xây dựng **Fashion-MNIST** – một tập dữ liệu với cấu trúc tương tự nhưng các ảnh là hình ảnh quần áo, giúp sinh động và thực tế hơn.

Label	Description	Examples
0	T-Shirt/Top	
1	Trouser	
2	Pullover	
3	Dress	
4	Coat	
5	Sandals	
6	Shirt	
7	Sneaker	
8	Bag	
9	Ankle boots	

Hình 34: Fashion-MNIST dataset

Bạn có thể thấy rằng ảnh thực chất chỉ là ma trận các con số. Khi được *flatten*, mỗi ảnh trở thành một vector gồm 784 phần tử kiểu `uint8`. Nếu coi mỗi phần tử là một đặc trưng (feature), và biết rằng có 10 loại mặt hàng (10 lớp), ta có thể huấn luyện mô hình phân loại dựa trên tập dữ liệu này.

Triển khai ý tưởng trên, ta thu được biểu đồ *loss* và *accuracy* sau:



Hình 35: Evaluate loss and accuracy

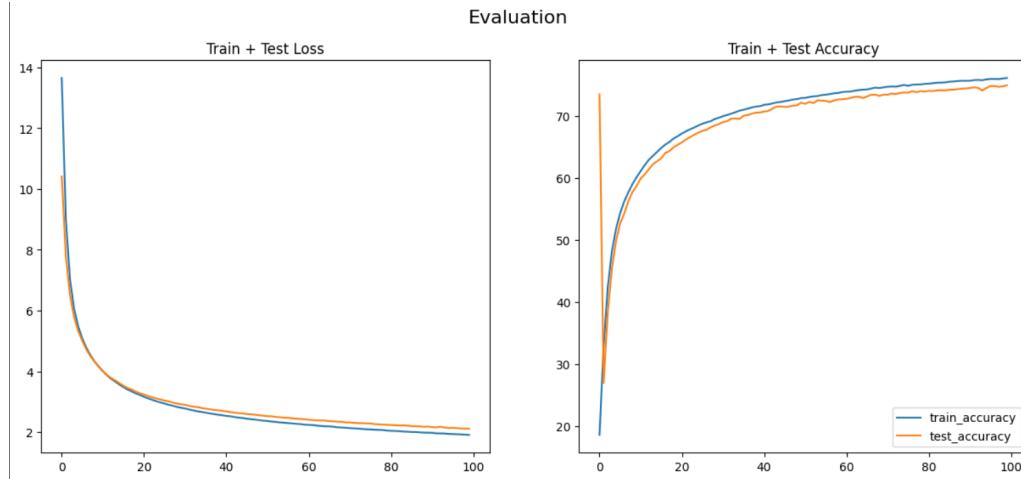
### 5.1.3 Normalization

Nhìn vào đồ thị, ta thấy hai hàm dao động mạnh mặc dù đã huấn luyện với `batch size` lớn là 5000. Nguyên nhân là do các concept trước đây thường sử dụng dữ liệu nhỏ ( $X$  nhỏ), dẫn đến đạo hàm  $\nabla_{\theta}L$  cũng nhỏ. Tuy nhiên, ở đây, dữ liệu phân bố từ 0 đến 255

nên nếu chọn learning rate như cũ, đạo hàm sẽ lớn, khiến tham số  $\theta$  cập nhật với bước nhảy lớn, gây dao động.

### Giải pháp:

- **Giảm learning rate (lr):** giúp lượng cập nhật nhỏ hơn và quá trình huấn luyện ổn định hơn.



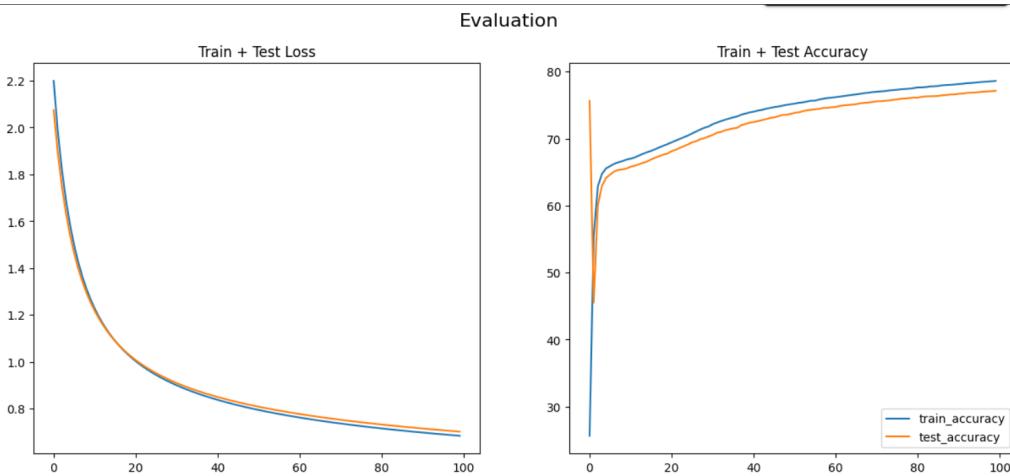
Hình 36: Evaluate khi ta giảm lr còn 0.0001

Đã thấy khi giảm learning rate, độ dao động đã giảm đi đáng kể. Tuy nhiên, mức chính xác vẫn còn hạn chế do chưa xử lý được gốc rễ vấn đề từ phân bố dữ liệu.

```
Epoch [90/100], Loss: 1.9816, Accuracy: 75.64%, Test Loss: 2.1849, Test Accuracy: 74.48%
Epoch [91/100], Loss: 1.9799, Accuracy: 75.65%, Test Loss: 2.1677, Test Accuracy: 74.59%
Epoch [92/100], Loss: 1.9623, Accuracy: 75.75%, Test Loss: 2.1577, Test Accuracy: 74.50%
Epoch [93/100], Loss: 1.9601, Accuracy: 75.78%, Test Loss: 2.1825, Test Accuracy: 74.07%
Epoch [94/100], Loss: 1.9576, Accuracy: 75.74%, Test Loss: 2.1543, Test Accuracy: 74.49%
Epoch [95/100], Loss: 1.9438, Accuracy: 75.86%, Test Loss: 2.1381, Test Accuracy: 74.80%
Epoch [96/100], Loss: 1.9382, Accuracy: 75.94%, Test Loss: 2.1449, Test Accuracy: 74.79%
Epoch [97/100], Loss: 1.9322, Accuracy: 75.94%, Test Loss: 2.1346, Test Accuracy: 74.67%
Epoch [98/100], Loss: 1.9273, Accuracy: 75.92%, Test Loss: 2.1209, Test Accuracy: 74.74%
Epoch [99/100], Loss: 1.9183, Accuracy: 76.02%, Test Loss: 2.1166, Test Accuracy: 74.92%
Epoch [100/100], Loss: 1.9112, Accuracy: 76.09%, Test Loss: 2.1154, Test Accuracy: 74.60%
```

Hình 37: Performance tốt nhất với lr = 0.0001

- **Normalization:** một khái niệm mới nhằm chuyển dữ liệu từ không gian lớn về không gian quen thuộc, giúp tối ưu dễ hơn. Thực chất đây là thao tác *scale* dữ liệu. Một số cách chuẩn hóa phổ biến:
  - Scale về khoảng [0, 1]: Đây là cách đơn giản và dễ hiểu, phổ biến trong nhiều mô hình học sâu.



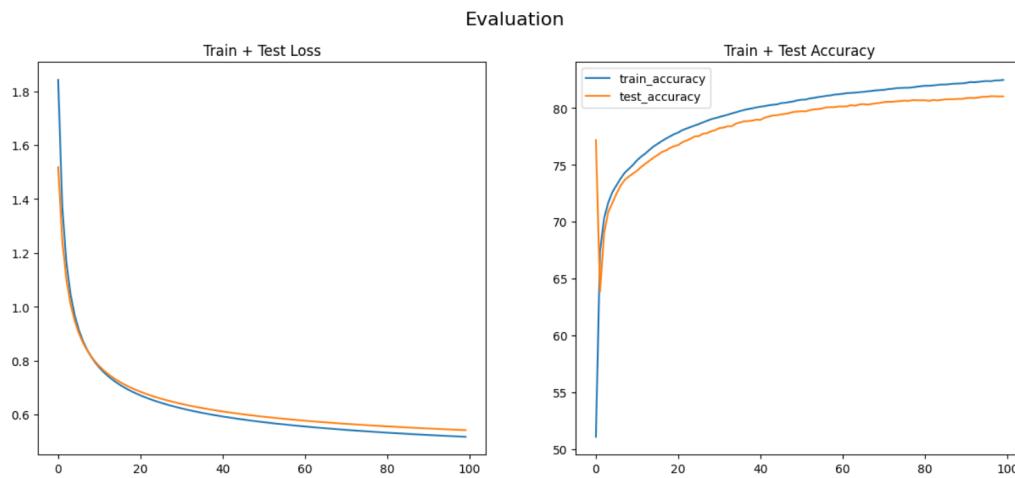
Hình 38: Evaluate với normalization  $[0, 1]$

Hiệu quả tốt hơn rõ rệt so với chỉ giảm learning rate, do model không còn bị "sốc" bởi các giá trị lớn.

```
Epoch [90/100], Loss: 0.6993, Accuracy: 78.09%, Test Loss: 0.7166, Test Accuracy: 76.70%
Epoch [91/100], Loss: 0.6977, Accuracy: 78.14%, Test Loss: 0.7150, Test Accuracy: 76.73%
Epoch [92/100], Loss: 0.6961, Accuracy: 78.19%, Test Loss: 0.7135, Test Accuracy: 76.82%
Epoch [93/100], Loss: 0.6945, Accuracy: 78.28%, Test Loss: 0.7119, Test Accuracy: 76.87%
Epoch [94/100], Loss: 0.6929, Accuracy: 78.31%, Test Loss: 0.7104, Test Accuracy: 76.88%
Epoch [95/100], Loss: 0.6914, Accuracy: 78.37%, Test Loss: 0.7089, Test Accuracy: 76.94%
Epoch [96/100], Loss: 0.6899, Accuracy: 78.42%, Test Loss: 0.7075, Test Accuracy: 77.02%
Epoch [97/100], Loss: 0.6884, Accuracy: 78.48%, Test Loss: 0.7060, Test Accuracy: 77.05%
Epoch [98/100], Loss: 0.6869, Accuracy: 78.52%, Test Loss: 0.7046, Test Accuracy: 77.09%
Epoch [99/100], Loss: 0.6855, Accuracy: 78.56%, Test Loss: 0.7032, Test Accuracy: 77.14%
Epoch [100/100], Loss: 0.6840, Accuracy: 78.61%, Test Loss: 0.7018, Test Accuracy: 77.18%
```

Hình 39: Performance tốt nhất với normalization  $[0, 1]$

- Scale về khoảng  $[-1, 1]$ : Phù hợp với các hàm kích hoạt như tanh, tạo sự cân bằng quanh 0 giúp hội tụ nhanh hơn trong nhiều trường hợp.



Hình 40: Evaluate với normalization  $[-1, 1]$

Chất lượng training tiếp tục được cải thiện, sai số dao động ít hơn.

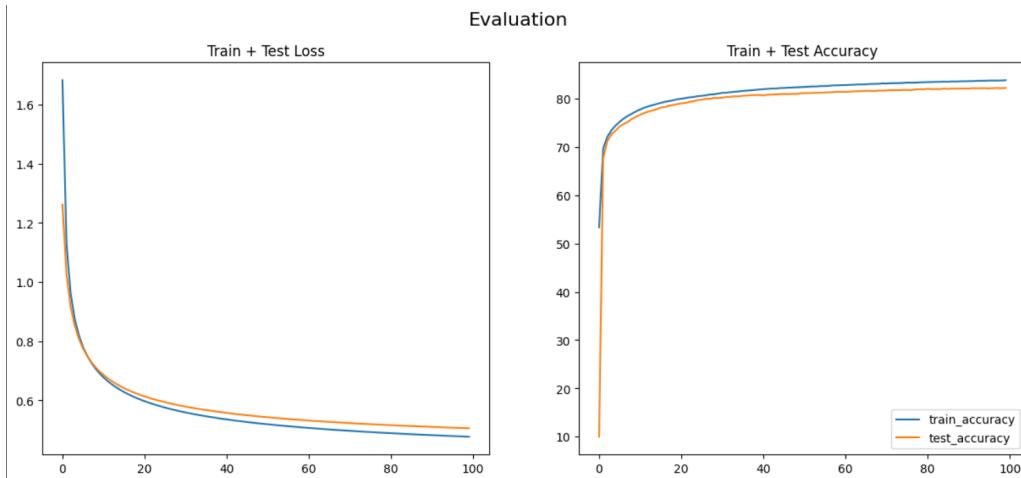
```

Epoch [90/100], Loss: 0.5251, Accuracy: 82.17%, Test Loss: 0.5492, Test Accuracy: 80.87%
Epoch [91/100], Loss: 0.5243, Accuracy: 82.21%, Test Loss: 0.5485, Test Accuracy: 80.90%
Epoch [92/100], Loss: 0.5235, Accuracy: 82.28%, Test Loss: 0.5478, Test Accuracy: 80.88%
Epoch [93/100], Loss: 0.5228, Accuracy: 82.27%, Test Loss: 0.5471, Test Accuracy: 80.93%
Epoch [94/100], Loss: 0.5220, Accuracy: 82.31%, Test Loss: 0.5464, Test Accuracy: 80.99%
Epoch [95/100], Loss: 0.5212, Accuracy: 82.35%, Test Loss: 0.5457, Test Accuracy: 80.99%
Epoch [96/100], Loss: 0.5205, Accuracy: 82.38%, Test Loss: 0.5450, Test Accuracy: 81.04%
Epoch [97/100], Loss: 0.5198, Accuracy: 82.36%, Test Loss: 0.5443, Test Accuracy: 81.03%
Epoch [98/100], Loss: 0.5191, Accuracy: 82.43%, Test Loss: 0.5437, Test Accuracy: 81.02%
Epoch [99/100], Loss: 0.5183, Accuracy: 82.43%, Test Loss: 0.5430, Test Accuracy: 81.02%
Epoch [100/100], Loss: 0.5176, Accuracy: 82.47%, Test Loss: 0.5424, Test Accuracy: 81.04%

```

Hình 41: Performance tốt nhất với normalization  $[-1, 1]$

- Normalization bằng mean và variance (Standardization): Phương pháp chuẩn hóa dữ liệu theo phân phối chuẩn (Gaussian), đưa dữ liệu về phân bố với kỳ vọng bằng 0 và phương sai bằng 1.



Hình 42: Evaluate với chuẩn hóa theo phân phối chuẩn

Đây là phương pháp mạnh nhất, giúp mô hình học được sâu sắc hơn và ổn định nhất xuyên suốt quá trình huấn luyện.

```

Epoch [90/100], Loss: 0.4827, Accuracy: 83.59%, Test Loss: 0.5108, Test Accuracy: 82.08%
Epoch [91/100], Loss: 0.4821, Accuracy: 83.61%, Test Loss: 0.5100, Test Accuracy: 82.16%
Epoch [92/100], Loss: 0.4815, Accuracy: 83.62%, Test Loss: 0.5096, Test Accuracy: 82.14%
Epoch [93/100], Loss: 0.4809, Accuracy: 83.68%, Test Loss: 0.5091, Test Accuracy: 82.15%
Epoch [94/100], Loss: 0.4803, Accuracy: 83.69%, Test Loss: 0.5084, Test Accuracy: 82.15%
Epoch [95/100], Loss: 0.4798, Accuracy: 83.70%, Test Loss: 0.5079, Test Accuracy: 82.11%
Epoch [96/100], Loss: 0.4791, Accuracy: 83.74%, Test Loss: 0.5074, Test Accuracy: 82.17%
Epoch [97/100], Loss: 0.4786, Accuracy: 83.75%, Test Loss: 0.5070, Test Accuracy: 82.19%
Epoch [98/100], Loss: 0.4781, Accuracy: 83.73%, Test Loss: 0.5065, Test Accuracy: 82.16%
Epoch [99/100], Loss: 0.4775, Accuracy: 83.75%, Test Loss: 0.5061, Test Accuracy: 82.21%
Epoch [100/100], Loss: 0.4769, Accuracy: 83.81%, Test Loss: 0.5056, Test Accuracy: 82.21%

```

Hình 43: Performance tốt nhất với normalization theo phân phối chuẩn

**Nhận xét:** Trong các cách tối ưu, việc đơn thuần giảm learning rate tuy có hiệu quả nhất định nhưng vẫn là giải pháp đối phó. Trong khi đó, normalization không chỉ giúp ổn định gradient mà còn cải thiện đáng kể chất lượng huấn luyện. Đặc biệt, phương pháp normalization dựa trên mean và variance cho kết quả vượt trội, do tận dụng được tính chất phân phối tự nhiên của dữ liệu.

Dễ thấy rằng với mô hình hiện tại, độ chính xác (accuracy) đã đạt mức bão hòa khoảng 85%. Trong khi đó, với một bộ dữ liệu đơn giản hơn như Iris, mô hình có thể đạt đến 97% độ chính xác. Điều này cho thấy rằng khi áp dụng softmax đơn thuần trên dữ liệu ảnh (như Fashion-MNIST), hiệu quả của mô hình không còn tối ưu.

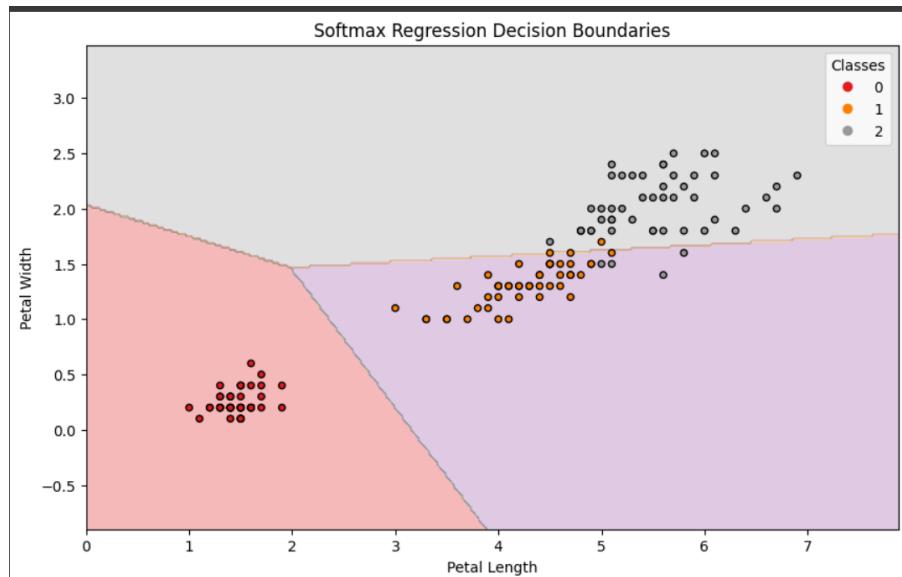
Nguyên nhân có thể xuất phát từ độ phức tạp của bài toán: trong khi dữ liệu Iris chỉ có 4 đặc trưng và 3 lớp đầu ra — tức là không gian đầu vào có chiều 4 và đầu ra có chiều 3 — thì đối với Fashion-MNIST, không gian đầu vào là 784 chiều (từ ảnh 28x28), đầu ra gồm 10 lớp. Việc chuyển từ không gian  $(4, 3)$  sang  $(784, 10)$  là một bước nhảy lớn về độ phức tạp, làm tăng mạnh kích thước không gian tìm kiếm của mô hình. Trong bối cảnh này, mô hình softmax đơn giản trở nên không đủ khả năng biểu diễn để học được các đặc trưng phức tạp từ ảnh.

Một cách trực quan, ta có thể chấp nhận rằng mô hình càng có nhiều tham số thì càng có khả năng học tốt hơn. Tuy nhiên, với dữ liệu từ Fashion-MNIST, số lượng đặc trưng đầu vào và số lớp đầu ra là cố định (lần lượt là 784 và 10, chưa kể bias). Do đó, cách khả thi duy nhất để tăng số lượng tham số của mô hình là chèn thêm các tầng trung gian (hidden layers) giữa tầng đầu vào và tầng đầu ra. Mỗi tầng trung gian có thể bao gồm nhiều nút (nodes), từ đó làm tăng số lượng tham số của toàn mô hình, cho phép nó học được các đặc trưng phức tạp hơn trong dữ liệu.

Chính điều này là tiền đề dẫn đến kiến trúc Multi-layer Perceptron (MLP) — một mạng nơ-ron nhiều tầng, trong đó mỗi tầng trung gian có khả năng học các biểu diễn phi tuyến (nonlinear representations) của dữ liệu, vượt ra ngoài giới hạn biểu diễn tuyến tính của softmax truyền thống.

#### 5.1.4 Cách tiếp cận thứ ba: Các bài toán phức tạp hơn

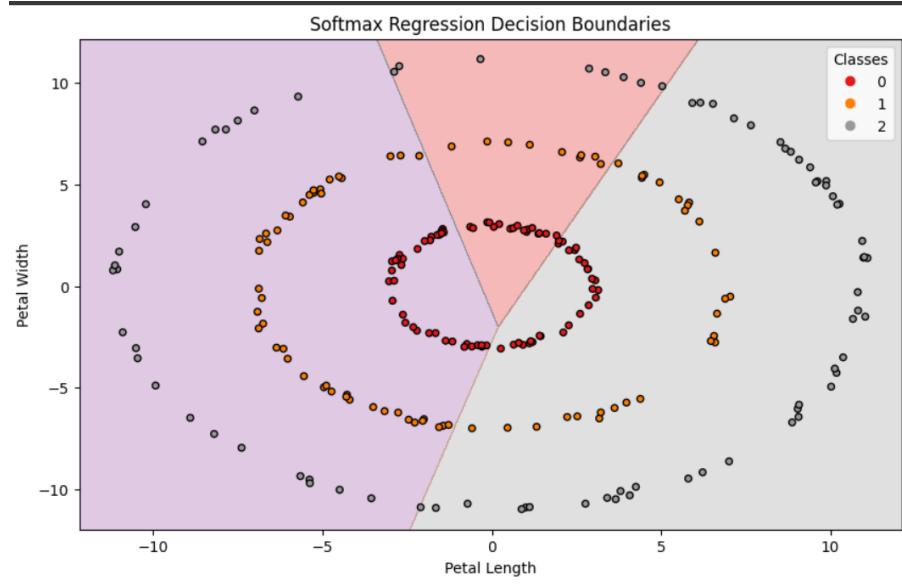
Đối với bài toán XOR, có thể giải quyết bằng các kỹ thuật như sigmoid one-vs-rest hoặc softmax nếu tổng quát hóa vùng phân hoạch, tuy nhiên điều này chỉ áp dụng hiệu quả cho các phân bố đơn giản. Để minh họa giới hạn của softmax, ta khảo sát hai tập dữ liệu dưới đây:



Hình 44: Phân hoạch lớp bằng softmax trên tập `iris_3class` (2D)

Trong trường hợp với 3 lớp, 2 thuộc tính đầu vào, softmax vẫn cho kết quả hợp lý và

có thể trực quan hóa được. Tuy nhiên, với các dữ liệu phức tạp hơn như tập donut — phân bố gồm ba vòng tròn đồng tâm đại diện cho ba lớp — softmax bắt đầu bộc lộ giới hạn.



Hình 45: Phân hoạch lớp bằng softmax trên tập dữ liệu donut

Rõ ràng trong trường hợp này, softmax không thể tạo ra ranh giới phân loại chính xác giữa các lớp. Điều đó cho thấy một hạn chế trong việc áp dụng trực tiếp mô hình tuyến tính vào các bài toán phân loại phi tuyến.

Một cách tiếp cận phổ biến trong các mô hình truyền thống như SVM là sử dụng *kernel trick* để ánh xạ dữ liệu từ không gian ban đầu sang một không gian chiều cao hơn, nơi các điểm dữ liệu có thể được phân tách tuyến tính. Tương tự, trong mạng neural, ta có thể thực hiện điều này bằng cách tăng số lượng tầng ẩn — tức là tăng số lượng tham số học — để cho phép mô hình học được các đặc trưng phức tạp hơn.

Tuy nhiên, chỉ riêng việc thêm nhiều tầng và nhiều node là chưa đủ. Nếu toàn bộ các tầng đều là tuyến tính, tức không sử dụng bất kỳ hàm kích hoạt phi tuyến nào, thì tổng thể mô hình vẫn chỉ tương đương một phép biến đổi tuyến tính duy nhất:

$$f(f(W_1x)) = W_2(W_1x) = (W_2W_1)x,$$

và như vậy, khả năng biểu diễn của mô hình không hề được mở rộng so với mô hình ban đầu.

Vì vậy, để mô hình có khả năng học các biểu diễn phi tuyến, ta bắt buộc phải chèn các hàm kích hoạt phi tuyến sau mỗi tầng, ví dụ như:

sigmoid, tanh, ReLU, etc.

Các hàm này sẽ phá vỡ tính tuyến tính của toàn mạng, cho phép mô hình có khả năng học được các đường phân chia phức tạp.

Một ví dụ minh họa là phân bố dữ liệu dạng **donut** (**vòng tròn rỗng giữa**). Đây là một bài toán mà không thể phân tách tuyến tính trong không gian hai chiều gốc, nhưng lại hoàn toàn có thể phân tách nếu ánh xạ dữ liệu qua một tầng phi tuyến. Điều này cho

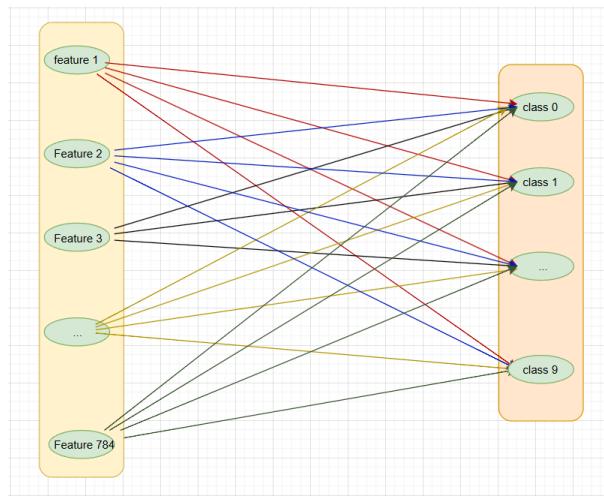
thấy rõ ràng vai trò thiết yếu của hàm kích hoạt trong việc mở rộng khả năng biểu diễn của mạng nơ-ron, đặc biệt đối với các bài toán có cấu trúc dữ liệu phức tạp.

**Source code (Google Colab):** [SoftMax Limitation](#)

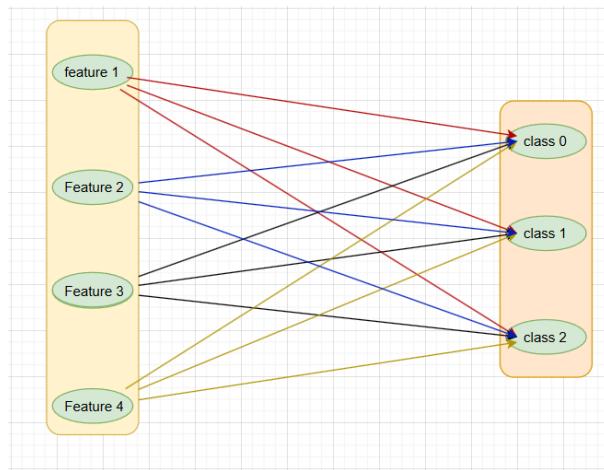
## 5.2 MLP và cách train mô hình

### 5.2.1 Layers

Tuy rằng chúng ta đã chỉ ra rằng một mô hình có nhiều tham số sẽ có khả năng học tốt hơn, dẫn đến nhu cầu thêm các tầng trung gian (hidden layers) để chứa nhiều node hơn, nhưng nếu chỉ xem layers như một công cụ để tăng tham số thì chưa thật sự đầy đủ về mặt khái niệm. Để hiểu sâu hơn vai trò của layers, ta sẽ khảo sát ý nghĩa của hidden layer trong mạng neural.



Hình 46: Mô hình rút gọn cho Fashion-MNIST (784 đầu vào, 10 lớp)



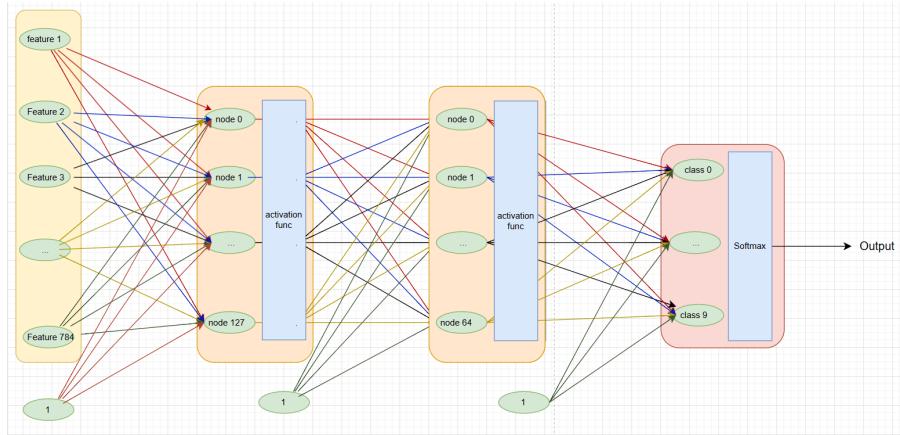
Hình 47: Mô hình rút gọn cho tập dữ liệu Irish (4 đầu vào, 3 lớp)

Gọi  $F_1$  là mô hình cho Fashion-MNIST với cấu trúc  $(784, 10)$  và  $F_2$  là mô hình cho Irish dataset với cấu trúc  $(4, 3)$ .

## Nhận xét:

- $F_1$  có số lượng đặc trưng (features) lớn hơn nhiều so với  $F_2$ .
- Tuy nhiên, trong  $F_2$ , mỗi đặc trưng mang tính trừu tượng cao và có khả năng đóng góp nhiều hơn vào việc phân lớp chính xác.

Lý do tập dữ liệu Irish cho kết quả tốt là bởi vì nó có ít đặc trưng (4 feature) nhưng mỗi feature lại mang thông tin quan trọng giúp phân biệt rõ ràng giữa các lớp (3 label). Trong khi đó, với Fashion-MNIST, input có tới 784 chiều (ảnh 28x28) và output là 10 lớp, thì mỗi node đầu ra phải tổng hợp thông tin từ toàn bộ 784 đặc trưng, một nhiệm vụ rất phức tạp. Điều này dẫn đến sai số cao nếu không có các tầng trung gian để "rút trích" những thông tin quan trọng hơn.



Hình 48: Mô hình MLP với 2 hidden layers: 128 và 64 nodes

Giả sử mô hình có hai tầng ẩn với 128 và 64 node nằm giữa input và output. Khi đó, ta có thể hình dung:

- Hidden layer 1 (128 node) có vai trò tổng hợp các đặc trưng hình học như đường nét, khói dạng (ví dụ: sandal có dạng hình chữ nhật ngang, áo thì hình dọc, v.v.).
- Hidden layer 2 (64 node) tiếp tục trừu tượng hóa từ đặc trưng trung gian ở layer trước (ví dụ: tổng hợp chi tiết hình tròn thành đặc trưng "bánh xe" nếu đang phân loại xe).

Do đó, ngoài **Input layer** và **Output layer**, một **Multi-layer Perceptron (MLP)** có thể có nhiều **Hidden layers**. Các hidden layer được đánh số theo thứ tự xuất hiện từ input đến output.

**Lưu ý:** Tổng số layer của MLP được tính bằng số hidden layers cộng thêm 1 (tầng output). Input layer không được tính trong tổng số này.

**Tăng số layer — khi nào hiệu quả?** Một câu hỏi tự nhiên được đặt ra là: nếu thêm nhiều layer sẽ làm tăng khả năng học, vậy chuyện gì sẽ xảy ra nếu ta liên tục thêm layer?

Chúng tôi thực hiện huấn luyện thực nghiệm trên mô hình với số hidden layers từ 1 đến 7, sử dụng cùng một hàm loss và learning rate. Kết quả cho thấy:

- Với 1–3 hidden layers, độ chính xác (accuracy) tăng đáng kể.
- Từ 4 layers trở đi, mô hình bắt đầu **bão hòa** — accuracy gần như không tăng thêm.
- Khi đạt tới 7 layers, mô hình không thể hội tụ — biểu đồ loss xuất hiện nhiều dao động (zigzag), dẫn đến huấn luyện thất bại.

Nguyên nhân của hiện tượng này liên quan đến việc các tham số được cập nhật quá lớn (do input là ảnh 0–255, learning rate 0.01). Khi số layer tăng, các tham số lan truyền và khuếch đại qua nhiều tầng, dẫn đến mất ổn định gradient, đặc biệt khi sử dụng ReLU mà không có kỹ thuật chuẩn hóa.

**Giải pháp:** Để xử lý các vấn đề như trên, các kỹ thuật như **Batch Normalization**, **Residual Connection** và các kiến trúc nâng cao (CNNs, ResNet) được áp dụng, sẽ được trình bày trong các phần tiếp theo.

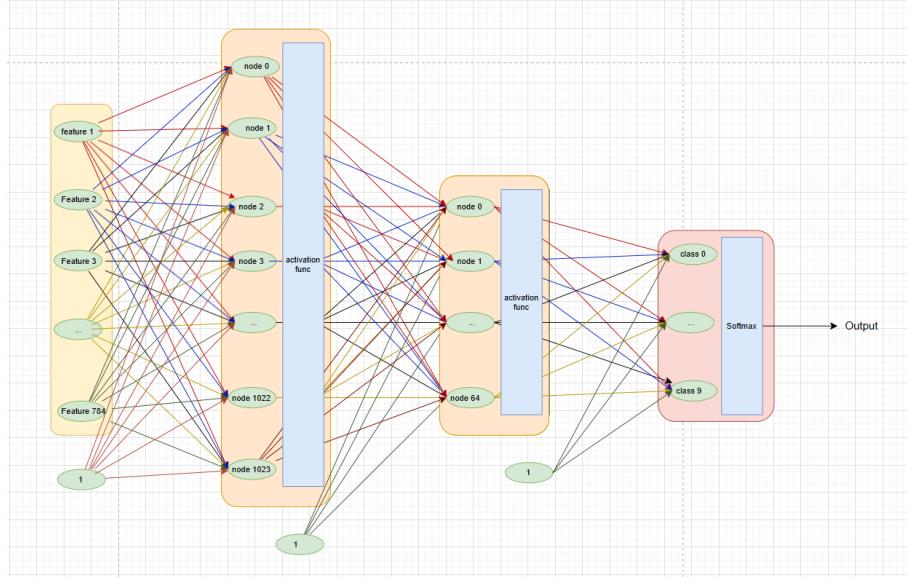
### 5.2.2 Nodes

Một nút hình tròn trong một layer được gọi là một **unit**. Các unit trong input layer, hidden layers, và output layer lần lượt được gọi là *input unit*, *hidden unit*, và *output unit*.

Đầu vào của mỗi unit trong hidden layer được ký hiệu là  $\mathbf{z}$ , trong khi đầu ra (sau khi áp dụng hàm kích hoạt) được ký hiệu là  $\mathbf{a}$  (viết tắt của *activation*). Điều này phản ánh giá trị đầu ra sau khi biến đổi phi tuyến được thực hiện trên  $\mathbf{z}$ .

Thông thường, số lượng node trong mỗi layer sẽ giảm dần theo thứ tự từ input layer đến output layer. Thiết kế này phản ánh quá trình tổng hợp thông tin một cách tự nhiên và phù hợp với trực giác: các layer ban đầu trích xuất đặc trưng chi tiết, trong khi các layer sau tổng hợp và ra quyết định. Hầu hết các bài báo trong lĩnh vực đều tuân theo mô hình này.

Tuy nhiên, điều đó không có nghĩa là việc thiết kế số lượng node phình to rồi thu hẹp dần (ví dụ như:  $64 \rightarrow 256 \rightarrow 64$ ) sẽ cho hiệu năng kém hơn. Trên thực tế, nếu một kiến trúc như vậy tạo ra kết quả tốt (SOTA – *state-of-the-art*) thì vẫn được công nhận là một đóng góp khoa học.



Hình 49: Model thiết kế nodes dạng phình ra

Một câu hỏi quan trọng được đặt ra tương tự như trong mục trước: điều gì sẽ xảy ra nếu ta tăng số lượng node trong một layer lên rất lớn?

Để khảo sát, ta thực hiện thực nghiệm với một mô hình chỉ có **một hidden layer**, lần lượt với số lượng node là: 64, 256, và 1024. Kết quả được trình bày trong các đồ thị sau.

Dễ thấy rằng với 256 node, mô hình đã có đủ khả năng để đưa ra quyết định phân loại. Việc tăng lên 1024 node không cải thiện độ chính xác đáng kể, thậm chí còn gây dư thừa và nhiều node sẽ có giá trị kích hoạt bằng 0. Điều này dẫn đến hiện tượng bão hòa (*saturation*) về độ chính xác trên tập huấn luyện (*train-acc*).

Giả sử một mô hình có  $N$  tham số, ta thường không thể biết chính xác liệu số lượng đó là thiếu hay dư. Vì vậy, một chiến lược thực tế là thiết kế dư số lượng node, sau đó để mô hình tự loại bỏ các node không cần thiết. Nếu không làm vậy, mô hình có thể rơi vào trạng thái *overfitting*.

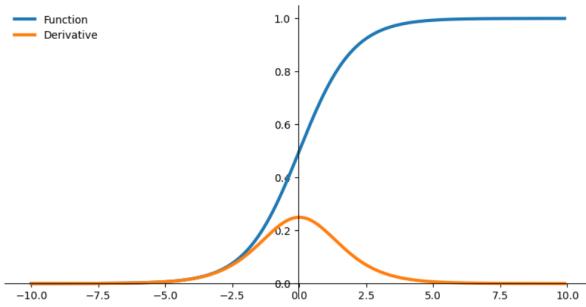
Trong thị giác máy tính (CV), đặc biệt là các bài toán như phát hiện vật thể (*object detection*), việc một số node có activation bằng 0 không gây ảnh hưởng nghiêm trọng. Điều này giống như việc ta che khuất một phần ảnh và yêu cầu mô hình vẫn phải dự đoán được nội dung còn lại, giúp tăng tính *robustness*.

Tuy nhiên, đối với các bài toán như sinh ảnh (GANs) hoặc các loại dữ liệu khác, việc quá nhiều node có activation bằng 0 là một vấn đề lớn. Do đó, nhiều hàm kích hoạt mới đã được phát triển để khắc phục vấn đề này, bao gồm:

- Leaky ReLU
- ELU (Exponential Linear Unit)
- PReLU (Parametric ReLU)
- SELU (Scaled ELU)
- Swish
- GeLU (Gaussian Error Linear Unit) – hiện là một trong những lựa chọn hiện đại nhất

### 5.2.3 Activation Function: ReLU

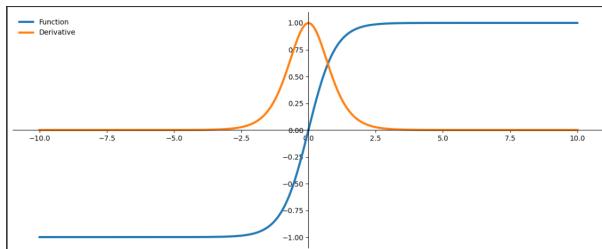
Như đã thảo luận trong mục "Tiếp cận 3", chúng ta cần một hàm kích hoạt (*activation function*) để biến đổi hàm tuyến tính ban đầu thành một hàm phi tuyến, nhằm tăng tính linh hoạt cho quá trình huấn luyện dữ liệu. Hai hàm kích hoạt phổ biến trong lịch sử là **sigmoid** và **tanh**, vốn thường được sử dụng trong các mô hình Logistic Regression. Sau đây là một số điểm tổng quan:



Hình 50: Hàm sigmoid và đạo hàm của nó

Hàm sigmoid có giá trị đầu ra nằm trong khoảng  $(0, 1)$ , gần 1 khi đầu vào lớn và gần 0 khi đầu vào nhỏ (rất âm). Một trong những lý do nó từng được sử dụng rộng rãi là vì đạo hàm của nó có dạng đẹp và đơn giản, thuận lợi cho việc tính toán gradient.

Tuy nhiên, hàm sigmoid có nhược điểm lớn là gây ra hiện tượng **saturation** và làm **gradient tiến dần về 0**. Cụ thể, khi  $|z|$  lớn, đạo hàm  $\sigma'(z)$  trở nên rất nhỏ, dẫn đến cập nhật trọng số trở nên không hiệu quả trong quá trình tối ưu hoá bằng phương pháp gradient descent. Ví dụ, khi  $z = 5$ , ta có  $\sigma(z) \approx 0.99$ , nhưng đạo hàm lúc này rất gần 0, khiến các tham số gần như không được cập nhật – mô hình không học được.



Hình 51: Hàm tanh và đạo hàm của nó

Hàm tanh có giá trị đầu ra nằm trong khoảng  $(-1, 1)$ , với các tính chất tương tự sigmoid. Mặc dù tâm đối xứng tại 0 giúp gradient cân bằng hơn, nhưng nó vẫn gặp vấn đề gradient nhỏ với đầu vào có trị tuyệt đối lớn.

Chính vì những hạn chế trên, một hàm kích hoạt mới đã được giới thiệu và sử dụng rộng rãi trong các mô hình hiện đại: **ReLU (Rectified Linear Unit)**.

$$f(z) = \max(0, z)$$

Ưu điểm lớn nhất của ReLU là tính đơn giản và khả năng hỗ trợ quá trình huấn luyện mạng nơ-ron sâu (*Deep Neural Networks*) hiệu quả hơn, đặc biệt là giúp giảm đáng kể thời gian hội tụ.

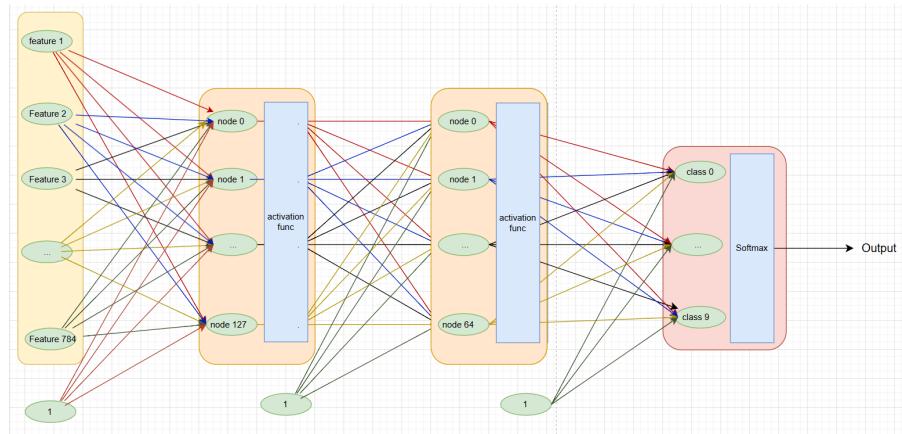
**Trực giác về ReLU:** Xét trong ngữ cảnh phân loại ảnh (ví dụ: phân biệt mèo và chó). Nếu sử dụng sigmoid, ta có thể hiểu rằng mô hình sẽ đưa ra xác suất xem ảnh đó là mèo hay không phải mèo – ví dụ: có tai nhọn, có râu, không có cánh,...

Tuy nhiên, với ReLU, đầu ra  $z$  có thể được hiểu như một **feature score**: nếu  $z > 0$ , thì đặc trưng đó ủng hộ cho việc phân loại là mèo, và  $z$  càng lớn thì mức độ "giống mèo" càng cao. Ngược lại, nếu  $z \leq 0$ , mô hình có thể hiểu đặc trưng đó là không liên quan hoặc phủ định hình ảnh con mèo.

Một đặc điểm mạnh mẽ nữa của ReLU là khả năng **bỏ qua các thông tin không quan trọng** (ví dụ như nền ảnh). Khi  $z \leq 0$ , tức feature không có ích cho việc phân loại, ReLU sẽ đưa giá trị kích hoạt về 0. Điều này không chỉ giúp giảm tính nhiễu mà còn đóng vai trò như một cơ chế chọn lọc đặc trưng.

Tóm lại, ReLU đóng vai trò quan trọng trong việc giải quyết vấn đề gradient nhỏ của sigmoid/tanh, đồng thời cung cấp một cách tiếp cận trực quan và hiệu quả hơn cho quá trình học của mạng nơ-ron hiện đại.

#### 5.2.4 Forward Pass



Hình 52: Mô hình MLP với 2 hidden layers: 128 và 64 nodes

Cũng giống như các mô hình trong học sâu (Deep Learning), quá trình **forward pass** trong MLP bắt đầu với dữ liệu đầu vào  $\mathbf{X}$  và các trọng số (weights)  $\boldsymbol{\theta}$ . Ta lần lượt tính:

- Tổng trọng số tuyến tính:  $z = \mathbf{W}x + b$
- Áp dụng hàm kích hoạt:  $a = \phi(z)$

Quy trình này được lặp lại qua từng tầng (layer) trong mạng. Với một mạng nhiều tầng (multi-layer), đầu ra của mỗi tầng đóng vai trò là đầu vào cho tầng tiếp theo. Việc chồng nhiều tầng phi tuyến (nhờ activation functions như ReLU) cho phép mạng học được các biểu diễn trừu tượng và phức tạp hơn của dữ liệu.

Ở tầng đầu ra (output layer), nếu bài toán là phân loại nhiều lớp (như nhận dạng ảnh với 10 lớp tương ứng với các chữ số 0–9), ta sử dụng **softmax** để chuyển đầu ra thành một phân phối xác suất:

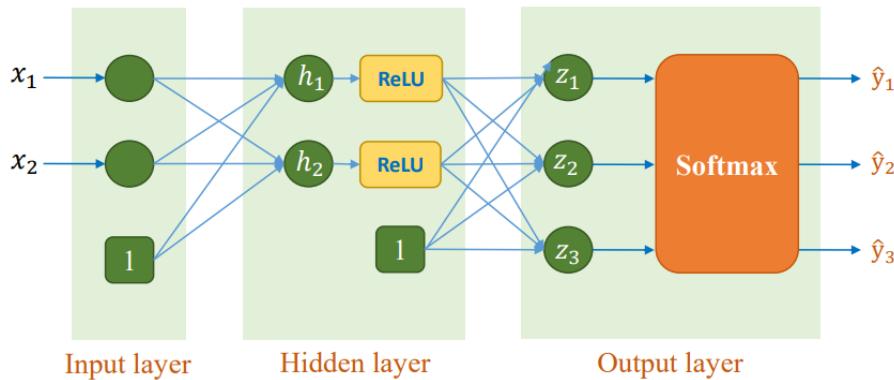
$$\hat{y}_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Với  $\hat{y}_i$  là xác suất mô hình dự đoán mẫu đầu vào thuộc lớp  $i$ . Mô hình sẽ chọn lớp có xác suất cao nhất làm dự đoán cuối cùng.

Sau khi thu được  $\hat{y}$ , ta so sánh với nhãn thật  $y$  để tính toán hàm mất mát (loss function), ví dụ như **Cross-Entropy Loss**. Giá trị loss này sẽ được sử dụng trong quá trình lan truyền ngược (backpropagation) để cập nhật các tham số mạng trong giai đoạn học.

Tóm lại, forward pass là bước lan truyền thông tin theo chiều xuôi, từ dữ liệu đầu vào cho tới đầu ra cuối cùng, và là bước đầu tiên trong quá trình huấn luyện một mạng MLP.

Giả sử ta có mô hình nhiều lớp như sau:



Hình 53: Mô hình MLP gồm một lớp ẩn và một lớp output

Không mất tính tổng quát, giả sử:

- $m$ : số lượng mẫu trong một batch (batch size)
- $n$ : số đặc trưng đầu vào (features), cộng thêm  $+1$  để tính bias
- $d^{(1)}$ : số neuron tại lớp ẩn thứ nhất
- $k$ : số lượng lớp (categories) trong nhãn đầu ra

Khi đó:

$$\mathbf{X} \in \mathbb{R}^{m \times n}$$

Với:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1,n-1} & 1 \\ x_{21} & x_{22} & \cdots & x_{2,n-1} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{m,n-1} & 1 \end{bmatrix} \quad (\text{thêm } 1 \text{ ở cuối để tính bias})$$

Với kiến trúc MLP gồm hai lớp tuyến tính (fully-connected), ta cần hai ma trận trọng số:

**Lớp ẩn:** Trọng số từ đầu vào đến lớp ẩn:

$$\mathbf{W}^{(1)} \in \mathbb{R}^{n \times d^{(1)}} \quad (\text{bao gồm bias})$$

Với mỗi cột  $W_{:,j}^{(1)}$  là vector trọng số ứng với neuron thứ  $j$  tại lớp ẩn:

$$W_{:,j}^{(1)} = \begin{bmatrix} w_{j,1}^{(1)} \\ w_{j,2}^{(1)} \\ \vdots \\ w_{j,n-1}^{(1)} \\ b_j^{(1)} \end{bmatrix}$$


---

**Lớp output:** Trọng số từ lớp ẩn (đã thêm bias) đến output:

$$\mathbf{W}^{(2)} \in \mathbb{R}^{(d^{(1)}+1) \times k}$$

Tương tự, mỗi cột  $W_{:,i}^{(2)}$  là vector trọng số cho class  $i$ :

$$W_{:,i}^{(2)} = \begin{bmatrix} w_{i,1}^{(2)} \\ w_{i,2}^{(2)} \\ \vdots \\ w_{i,d^{(1)}}^{(2)} \\ b_i^{(2)} \end{bmatrix}$$


---

Do đó, luồng truyền thẳng (forward) của mô hình là:

$$\mathbf{Z}^{(1)} = \mathbf{X}\mathbf{W}^{(1)} \quad (\text{kích hoạt ẩn})$$

$$\mathbf{A}^{(1)} = \phi(\mathbf{Z}^{(1)}) \quad (\text{áp dụng hàm kích hoạt, ví dụ ReLU})$$

$$\mathbf{A}_{\text{bias}}^{(1)} = [\mathbf{A}^{(1)} \quad \mathbf{1}] \in \mathbb{R}^{m \times (d^{(1)}+1)}$$

$$\mathbf{Z}^{(2)} = \mathbf{A}_{\text{bias}}^{(1)} \mathbf{W}^{(2)} \quad (\text{output trước softmax})$$

$$\hat{\mathbf{Y}} = \text{softmax}(\mathbf{Z}^{(2)})$$

Với nhãn thực tế  $\mathbf{Y}$ , ta chuyển đổi sang dạng one-hot:

$$\mathbf{Y} \leftarrow \text{one\_hot\_encoding}(\mathbf{Y})$$

Sau đó, ta tính hàm mất mát entropy chéo:

$$\mathcal{L} = \text{CrossEntropy}(\mathbf{Y}, \hat{\mathbf{Y}})$$


---

Như vậy, so với kiến trúc softmax tuyến tính trước đây chỉ có một ma trận trọng số, kiến trúc MLP đòi hỏi hai bước nhân ma trận và thêm hàm kích hoạt phi tuyến giữa các lớp.

### Trường hợp tổng quát nhiều lớp (multi-layer)

Giả sử mạng có  $L$  tầng ẩn, ta tiến hành lan truyền tiến (forward propagation) như sau:

$$\begin{aligned}\mathbf{A}^{(0)} &= \mathbf{X} \quad (\text{đầu vào ban đầu}) \\ \mathbf{A}_{\text{bias}}^{(0)} &= [\mathbf{A}^{(0)} \quad \mathbf{1}]\end{aligned}$$

Với mỗi tầng  $k = 1, 2, \dots, L - 1$ :

$$\begin{aligned}\mathbf{Z}^{(k)} &= \mathbf{A}_{\text{bias}}^{(k-1)} \mathbf{W}^{(k)} \quad (\text{tiền kích hoạt tầng } k) \\ \mathbf{A}^{(k)} &= \phi(\mathbf{Z}^{(k)}) \quad (\text{hàm kích hoạt, ví dụ ReLU}) \\ \mathbf{A}_{\text{bias}}^{(k)} &= [\mathbf{A}^{(k)} \quad \mathbf{1}] \in \mathbb{R}^{m \times (d^{(k)} + 1)}\end{aligned}$$

Tầng cuối cùng (tầng  $L$ ) không cần thêm bias sau hàm kích hoạt:

$$\begin{aligned}\mathbf{Z}^{(L)} &= \mathbf{A}_{\text{bias}}^{(L-1)} \mathbf{W}^{(L)} \quad (\text{tiền kích hoạt tầng cuối}) \\ \hat{\mathbf{Y}} &= \text{softmax}(\mathbf{Z}^{(L)}) \quad (\text{xác suất dự đoán đầu ra})\end{aligned}$$

Với nhãn thực tế  $\mathbf{Y}$ , ta chuyển đổi sang dạng one-hot:

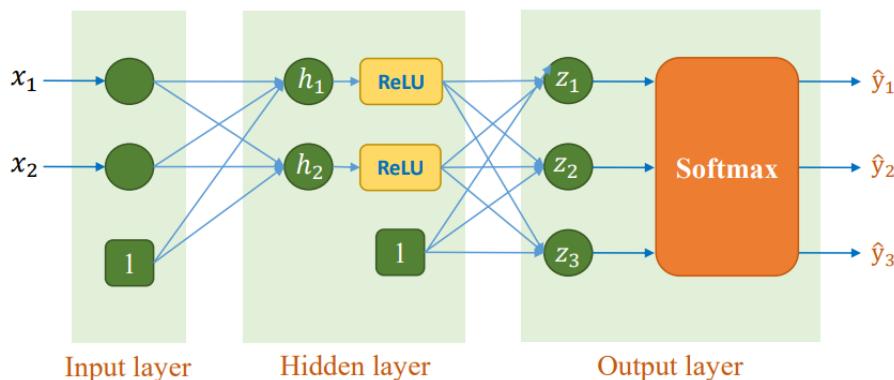
$$\mathbf{Y} \leftarrow \text{one\_hot\_encoding}(\mathbf{Y})$$

Sau đó, ta tính hàm mất mát entropy chéo:

$$\mathcal{L} = \text{CrossEntropy}(\mathbf{Y}, \hat{\mathbf{Y}})$$

Đoạn thể hiện toán hơi phức tạp nhưng nếu bạn đọc các *concept* trước và hiểu rõ về cách Linear, Logistic, Softmax Regression hoạt động thì công thức trên sẽ không quá khó hiểu.

#### 5.2.5 Backpropagation



Hình 54: Mô hình MLP gồm một lớp ẩn và một lớp output

**Mô hình 2 lớp (2-layer neural network):** Tương tự các mô hình trước, sau khi tính toán hàm mất mát, ta sử dụng đạo hàm của nó để cập nhật trọng số. Tuy nhiên, đối với MLP, mỗi layer sẽ có một ma trận trọng số riêng (kèm bias), và do đó ta cần tính đạo hàm với respect tới các tham số này tại từng layer. Các layer ẩn cũng chứa hàm kích hoạt (ví dụ: ReLU), nên việc truyền ngược sẽ cần tính đạo hàm qua activation như với Softmax. Ta sẽ xây dựng công thức tổng quát cho quá trình backpropagation của MLP.

**1. Gradient của hàm mất mát theo  $Z^{[L]}$  (lớp cuối cùng)** Giả sử lớp cuối cùng sử dụng Softmax và hàm mất mát là Cross-Entropy, ta có đạo hàm đơn giản:

$$\frac{\partial L}{\partial Z^{[L]}} = \hat{Y} - Y$$

Kích thước:  $\frac{\partial L}{\partial Z^{[L]}} \in \mathbb{R}^{N \times D^{[L]}}$ .

**2. Gradient của trọng số mở rộng lớp cuối cùng  $\tilde{W}^{[L]}$**  Tổng quát hóa công thức gradient quen thuộc:

$$\frac{\partial L}{\partial \tilde{W}^{[L]}} = \frac{1}{N} \cdot (\tilde{A}^{[L+1]})^T \cdot \frac{\partial L}{\partial Z^{[L]}}$$

Trong đó: -  $\tilde{A}^{[L-1]} \in \mathbb{R}^{N \times (D^{[L-1]}+1)}$  là output của lớp trước đã thêm cột bias. - Kết quả:  $\frac{\partial L}{\partial \tilde{W}^{[L]}} \in \mathbb{R}^{(D^{[L-1]}+1) \times D^{[L]}}$

Hàng cuối cùng của ma trận gradient tương ứng với đạo hàm theo bias.

**3. Truyền ngược qua các lớp ẩn ( $l = L-1, L-2, \dots, 1$ )** Giả sử:

$$Z^{[l+1]} = \tilde{A}^{[l]} \cdot \tilde{W}^{[l+1]} = [A^{[l]}, \mathbf{1}] \cdot \tilde{W}^{[l+1]}$$

Thì đạo hàm của hàm mất mát theo  $A^{[l]}$  là:

$$\frac{\partial L}{\partial A^{[l]}} = \frac{\partial L}{\partial Z^{[l+1]}} \cdot \left( \tilde{W}_{[:, D^{[l]}, :]}^{[l+1]} \right)^T$$

Trong đó: -  $\tilde{W}_{[:, D^{[l]}, :]}^{[l+1]}$  là  $\tilde{W}^{[l+1]}$  bỏ đi hàng bias (hàng cuối), do bias không truyền ngược. - Kích thước:

- $\frac{\partial L}{\partial Z^{[l+1]}} \in \mathbb{R}^{N \times D^{[l+1]}}$
- $\tilde{W}_{[:, D^{[l]}, :]}^{[l+1]} \in \mathbb{R}^{D^{[l]} \times D^{[l+1]}}$
- $\Rightarrow \frac{\partial L}{\partial A^{[l]}} \in \mathbb{R}^{N \times D^{[l]}}$

**Lý do loại bias:** Bias không kết nối với đầu ra thông qua các neuron (nó chỉ thêm trực tiếp), nên không tham gia vào quá trình truyền ngược qua ma trận.

**4. Gradient qua hàm kích hoạt (ví dụ: ReLU)** Vì activation là hàm đơn biến áp dụng theo từng phần tử, ta sử dụng phép nhân Hadamard:

$$\frac{\partial L}{\partial Z^{[l]}} = \frac{\partial L}{\partial A^{[l]}} \circ g'^{[l]}(Z^{[l]})$$

Với ReLU:

$$g'^{[l]}(Z_{ij}) = \begin{cases} 0 & \text{nếu } Z_{ij} \leq 0 \\ 1 & \text{nếu } Z_{ij} > 0 \end{cases}$$

## 5. Gradient của trọng số mờ rỗng lớp $l$

$$\frac{\partial L}{\partial \tilde{W}^{[l]}} = \frac{1}{N} \cdot (\tilde{A}^{[l-1]})^T \cdot \frac{\partial L}{\partial Z^{[l]}}$$

Trong đó: -  $\tilde{A}^{[l-1]} \in \mathbb{R}^{N \times (D^{[l-1]}+1)}$  -  $\frac{\partial L}{\partial Z^{[l]}} \in \mathbb{R}^{N \times D^{[l]}}$  -  $\Rightarrow \frac{\partial L}{\partial \tilde{W}^{[l]}} \in \mathbb{R}^{(D^{[l-1]}+1) \times D^{[l]}}$

## 6. Tổng quát: Trường hợp mạng nhiều lớp

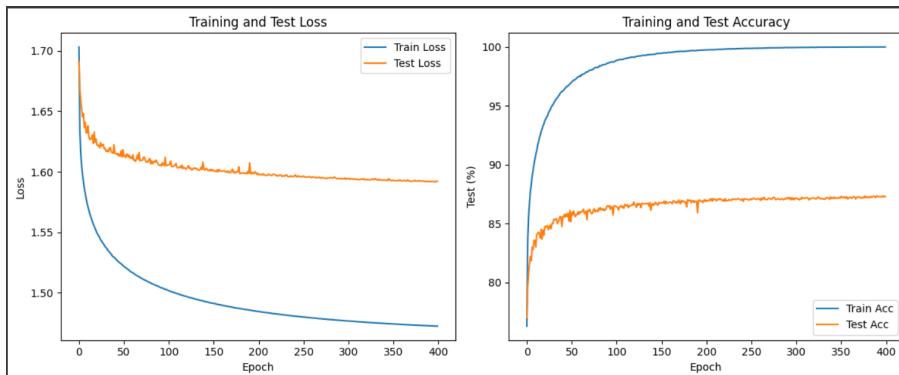
Giả sử mạng có  $L$  tầng ẩn, ta tiến hành backward propagation như sau:

$$\frac{\partial L}{\partial Z^{[L]}} = \hat{Y} - Y$$

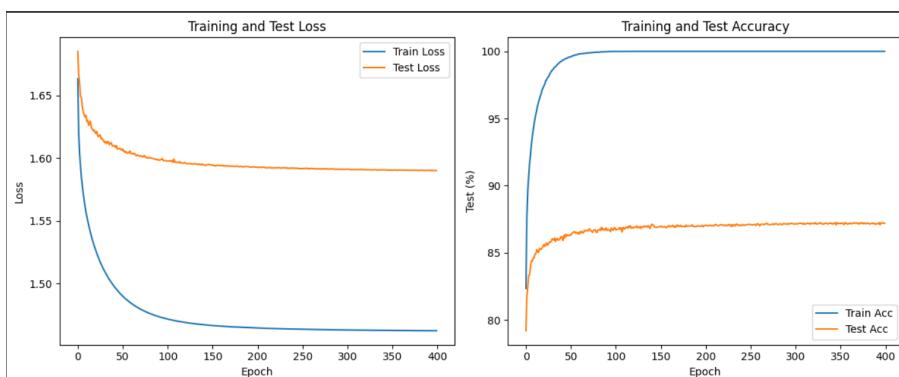
Với mỗi tầng  $l = L-1, L-2, \dots, 2, 1$ :

$$\begin{aligned}\frac{\partial L}{\partial Z^{[l]}} &= \frac{\partial L}{\partial A^{[l+1]}} \circ \phi'^{[l]}(Z^{[l]}) \\ \frac{\partial L}{\partial \tilde{W}^{[l]}} &= \frac{1}{N} \cdot (\tilde{A}^{[l]})^T \cdot \frac{\partial L}{\partial Z^{[l]}} \\ \frac{\partial L}{\partial A^{[l]}} &= \frac{\partial L}{\partial Z^{[l]}} \cdot \left( \tilde{W}_{[:, D^{[l-1]}, :]}^{[l]} \right)^T\end{aligned}$$

### 5.3 Implementation



Hình 55: Performance của 1 hidden layer 64



Hình 56: Performance của 3 hidden layer 512 - 256 - 128

Performance của 3 hidden layer 1024 - 512 - 256: thuận numpy nên không dùng GPU được -> chậm

**Source code (Google Colab):** [MLP](#)

## 6 Advanced MLP

---

### 6.1 Initialization

#### 6.1.1 Dying ReLU

Khi huấn luyện mạng MLP với hàm kích hoạt ReLU, việc khởi tạo ma trận trọng số ban đầu đóng vai trò vô cùng quan trọng. Nếu các trọng số được khởi tạo không hợp lý, một số nút có thể nhận đầu vào âm lớn khiến đầu ra ReLU bằng 0. Dựa vào cơ chế lan truyền ngược (backpropagation), ta thấy rằng các node đó sẽ có đạo hàm bằng 0, khiến cho gradient không được lan truyền về các lớp trước. Điều này làm cho các trọng số liên kết với node đó không được cập nhật — hiện tượng này được gọi là **Dying ReLU**.

(Minh họa hiện tượng Dying ReLU)

Khi một node đã bị "chết" (đầu ra bằng 0), nó sẽ không phục hồi dù quá trình huấn luyện tiếp diễn. Đây là lý do tại sao lựa chọn trọng số ban đầu rất quan trọng.

#### 6.1.2 Gradient Exploding

Một vấn đề ngược lại là **Gradient Exploding**, xảy ra khi các trọng số được khởi tạo quá lớn hoặc số tầng mạng quá nhiều khiến đạo hàm trở nên cực lớn khi lan truyền ngược. Gradient quá lớn làm cho mô hình cập nhật tham số một cách không ổn định, nhảy loạn hoặc thậm chí phân kỳ.

(ReLU không chuẩn hóa khiến quá trình huấn luyện bị mất kiểm soát)

Một số kỹ thuật để khắc phục tình trạng này bao gồm:

- Sử dụng **Batch Normalization** sau mỗi tầng.
- Áp dụng **Gradient Clipping** để giới hạn độ lớn của đạo hàm.

#### 6.1.3 Gradient Vanishing với Sigmoid

Trước khi có ReLU, hàm kích hoạt sigmoid là lựa chọn phổ biến. Tuy nhiên sigmoid cũng gây ra vấn đề **Gradient Vanishing**. Khi đầu ra của sigmoid tiến về 0 hoặc 1, đạo hàm gần bằng 0, khiến gradient giảm dần theo từng lớp — đặc biệt nghiêm trọng ở các mạng nhiều tầng.

(So sánh sigmoid có chuẩn hóa và không)

Ngay cả với một tầng ẩn, nếu khởi tạo trọng số quá lớn hoặc quá nhỏ, đạo hàm cũng sẽ gần 0:

- **Trọng số quá lớn:** sigmoid tiến nhanh về 0 hoặc 1  $\Rightarrow$  đạo hàm  $\approx 0$ .
- **Trọng số quá nhỏ:** tín hiệu yếu  $\Rightarrow$  mô hình học chậm.

Ngày nay, ReLU giúp giảm thiểu gradient vanishing, nhưng điều này không có nghĩa là có thể bỏ qua việc khởi tạo trọng số ban đầu.

Từ các ví dụ trên, ta thấy việc khởi tạo trọng số là yếu tố then chốt để tránh cả hai vấn đề:

- **ReLU**: tránh khởi tạo trọng số quá nhỏ (dễ chết).
- **Sigmoid**: tránh khởi tạo trọng số quá lớn (gradient nhỏ).

Vì vậy, cộng đồng học máy đã phát triển các phương pháp khởi tạo phù hợp với từng loại hàm kích hoạt, tiêu biểu là **Xavier Initialization** và **He Initialization**. Chúng ta sẽ tìm hiểu chúng trong phần tiếp theo.

#### 6.1.4 Relation Concept

**Mean (Kỳ vọng)** Chúng ta đã quen với việc hiểu **mean** là giá trị trung bình, ví dụ như trung bình điểm, trung bình cộng. Tuy nhiên, trong xác suất thống kê, khi nói đến **mean** thì ta đang nói đến *kỳ vọng* của một biến ngẫu nhiên  $X$ . Kỳ vọng không chỉ đơn thuần là giá trị trung bình mà là **giá trị trung bình có trọng số**, với trọng số là xác suất xảy ra của từng giá trị:

$$\mathbb{E}(X) = \sum_{i=1}^N X_i \cdot P_X(X_i)$$

Với  $P_X(X_i)$  là xác suất để biến ngẫu nhiên  $X$  nhận giá trị  $X_i$ . Nếu là biến liên tục, tổng trên được thay bằng tích phân.

**Variance (Phương sai)** Phương sai là độ đo thể hiện mức độ phân tán của các giá trị  $X_i$  so với kỳ vọng  $\mathbb{E}(X)$ . Nó được định nghĩa là kỳ vọng của bình phương sai lệch so với giá trị kỳ vọng:

$$\text{Var}(X) = \mathbb{E} [(X - \mathbb{E}(X))^2] = \sum_{i=1}^N (X_i - \mathbb{E}(X))^2 \cdot P_X(X_i)$$

Đây không phải công thức "thân thuộc" trong ứng dụng, nhưng ta có thể biến đổi nó:

$$\begin{aligned} \text{Var}(X) &= \sum_{i=1}^N (X_i - \mathbb{E}(X))^2 \cdot P_X(X_i) \\ &= \sum_{i=1}^N (X_i^2 - 2X_i \cdot \mathbb{E}(X) + \mathbb{E}(X)^2) \cdot P_X(X_i) \\ &= \sum X_i^2 P_X(X_i) - 2\mathbb{E}(X) \sum X_i P_X(X_i) + \mathbb{E}(X)^2 \sum P_X(X_i) \\ &= \mathbb{E}(X^2) - 2\mathbb{E}(X)^2 + \mathbb{E}(X)^2 \\ &= \mathbb{E}(X^2) - (\mathbb{E}(X))^2 \end{aligned}$$

**Variance of product of variables** Giả sử  $X$  và  $Y$  là hai biến ngẫu nhiên độc lập, ta có:

$$\begin{aligned}\text{Var}(XY) &= \mathbb{E}(X^2Y^2) - (\mathbb{E}(XY))^2 \\ &= \mathbb{E}(X^2)\mathbb{E}(Y^2) - (\mathbb{E}(X)\mathbb{E}(Y))^2 \\ &= [\text{Var}(X) + \mathbb{E}(X)^2][\text{Var}(Y) + \mathbb{E}(Y)^2] - \mathbb{E}(X)^2\mathbb{E}(Y)^2 \\ &= \text{Var}(X)\text{Var}(Y) + \text{Var}(X)\mathbb{E}(Y)^2 + \text{Var}(Y)\mathbb{E}(X)^2\end{aligned}$$

Công thức này có thể phức tạp, nhưng trong thực tế, chúng ta có thể đơn giản hóa bằng cách chọn các biến ngẫu nhiên có kỳ vọng bằng 0:

- Chọn  $\mathbb{E}(X) = 0$  bằng cách chuẩn hóa phân phối  $X$  thành phân phối chuẩn hóa, ví dụ như phân phối đều  $\mathcal{U}(-1, 1)$  - Chọn  $\mathbb{E}(Y) = 0$  bằng cách sử dụng zero-mean initialization hoặc chọn các phân phối đối xứng quanh 0

**Uniform Distribution** Với phân phối đều (uniform) trên đoạn  $[a, b]$ , mọi giá trị trong khoảng này đều có xác suất như nhau. Do đó:

$$f(x) = \frac{1}{b-a}, \quad \text{for } x \in [a, b]$$

Kỳ vọng:

$$\begin{aligned}\mathbb{E}(X) &= \int_a^b x \cdot \frac{1}{b-a} dx = \frac{1}{b-a} \int_a^b x dx \\ &= \frac{1}{b-a} \cdot \left[ \frac{x^2}{2} \right]_a^b = \frac{1}{b-a} \cdot \left( \frac{b^2 - a^2}{2} \right) = \frac{a+b}{2}\end{aligned}$$

Phương sai được chứng minh như sau (tham khảo hình ảnh):

$$\text{Var}(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2$$

$$\begin{aligned}\text{Var}(X) &= \int_a^b \left( x - \frac{a+b}{2} \right)^2 \cdot \frac{1}{b-a} dx \\ &= \frac{1}{b-a} \int_a^b \left( x^2 - x(a+b) + \frac{(a+b)^2}{4} \right) dx \\ &= \frac{1}{b-a} \left[ \frac{x^3}{3} - \frac{(a+b)}{2}x^2 + \frac{(a+b)^2}{4}x \right]_a^b \\ &= \frac{1}{b-a} \left( \frac{b^3 - a^3}{3} - \frac{(a+b)(b^2 - a^2)}{2} + \frac{(a+b)^2(b-a)}{4} \right) \\ &= \frac{(b-a)^2}{12}\end{aligned}$$

**Phân phối Gaussian** Mặc dù hàm mật độ xác suất (PDF) của phân phối Gaussian có biểu thức toán học phức tạp, nhưng thực nghiệm cho thấy rất nhiều hiện tượng trong xã hội (chẳng hạn như điểm số của một quần thể học sinh) tuân theo phân phối này.

Phân phối Gaussian còn gọi là phân phối chuẩn, được ký hiệu như sau:

$$X \sim \mathcal{N}(\mu, \sigma^2)$$

Hàm mật độ xác suất có dạng:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Không giống như phân phối đồng đều (uniform) vốn cần tính toán trung bình và phương sai một cách riêng biệt, phân phối Gaussian được đặc trưng hoàn toàn bởi hai tham số này: trung bình  $\mu$  và phương sai  $\sigma^2$ , và chúng xuất hiện trực tiếp trong công thức.

**Khai triển Maclaurin** Một lợi ích của việc đưa kỳ vọng (mean) về 0 là giúp đơn giản hóa biểu thức thông qua khai triển Maclaurin, một trường hợp đặc biệt của khai triển Taylor tại  $x = 0$ . Đây là một công cụ quan trọng trong giải tích để xấp xỉ các hàm phức tạp bằng chuỗi vô hạn các đa thức:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{n!} x^n = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \dots$$

Ý tưởng là thay vì xử lý trực tiếp một hàm phức tạp, ta có thể biểu diễn nó thông qua tổng các đạo hàm tại 0, giúp việc tính toán, phân tích hoặc xấp xỉ trở nên dễ dàng hơn.

**Ví dụ: Khai triển hàm tanh** Hàm tanh( $x$ ) có các giá trị đạo hàm tại 0 như sau:

$$\tanh(0) = 0, \quad \tanh'(0) = 1, \quad \tanh''(0) = 0, \quad \tanh'''(0) = -2$$

Khai triển Maclaurin của tanh( $x$ ):

$$\tanh(x) = x - \frac{x^3}{3!} + \dots$$

Ta thấy rằng các bậc cao hơn đóng góp ít dần, do đó trong nhiều trường hợp, có thể xấp xỉ:

$$\tanh(x) \approx x$$

Từ đó, nếu  $X$  là biến ngẫu nhiên với kỳ vọng bằng 0, ta có thể suy ra gần đúng:

$$\mathbb{V}[\tanh(X)] \approx \mathbb{V}[X]$$

(tham khảo: Giáo trình Giải tích 1 – Đại học Bách khoa)

**Ví dụ: Khai triển hàm sigmoid** Hàm sigmoid có biểu thức:

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

Với các đạo hàm tại 0:

$$\sigma(0) = \frac{1}{2}, \quad \sigma'(0) = \frac{1}{4}, \quad \sigma''(0) = 0$$

Khai triển Maclaurin:

$$\sigma(x) = \frac{1}{2} + \frac{x}{4} + \dots \Rightarrow \sigma(x) \approx \frac{1}{2} + \frac{x}{4}$$

Đây là một cách xấp xỉ tuyến tính đơn giản, thường được sử dụng để phân tích hoặc tối ưu trong học máy, đặc biệt khi biên độ của  $x$  nhỏ.

### 6.1.5 Xavier Initialization

Xét một hidden layer  $L_i$  bất kỳ, đầu vào là  $X_i$  và đầu ra là  $A_i$ . Ta mong muốn  $A_i$  không quá lớn (tránh *explosion*) và cũng không quá nhỏ (tránh *vanishing*). Điều kiện lý tưởng là:

$$\text{Var}(X_i) = \text{Var}(A_i)$$

vì phương sai đặc trưng cho độ phân tán dữ liệu – nếu phương sai quá nhỏ thì tín hiệu truyền qua mạng sẽ bị “xẹp lại”, còn quá lớn thì gây mất ổn định.

Xét công thức logit:

$$z_i = X \cdot W = x_1 w_1 + x_2 w_2 + \dots + x_n w_n + b$$

Giả sử:

-  $x_i$  là các biến ngẫu nhiên độc lập, lấy từ phân phối chuẩn hóa (do input đã được chuẩn hóa).

-  $w_i$  là các biến ngẫu nhiên độc lập, khởi tạo từ một phân phối đã chọn (Uniform hoặc Gaussian).

-  $b = 0$  (giá trị bias ban đầu thường đặt là 0).

Khi đó:

$$\text{Var}(z_i) = \text{Var}\left(\sum_{j=1}^n x_j w_j\right) = \sum_{j=1}^n \text{Var}(x_j w_j)$$

Do  $x_j$  và  $w_j$  độc lập và cùng phân phối:

$$= n \cdot \text{Var}(x_j) \cdot \text{Var}(w_j)$$

**Trường hợp activation là tanh** Từ khai triển Taylor:

$$\tanh(x) \approx x \Rightarrow \text{Var}(A_i) = \text{Var}(Z_i)$$

Do đó:

$$\text{Var}(X_i) = \text{Var}(Z_i) = n \cdot \text{Var}(x_i) \cdot \text{Var}(w_i) \Rightarrow \text{Var}(w_i) = \frac{1}{n}$$

- Nếu khởi tạo từ Uniform:  $\text{Var}(w_i) = \frac{(b-a)^2}{12} = \frac{1}{n}$ . Do  $\mathbb{E}(w_i) = 0 \Rightarrow a = -r, b = r$ , ta được:

$$2r = \sqrt{12 \cdot \frac{1}{n}} \Rightarrow r = \sqrt{\frac{3}{n}} \Rightarrow w_i \sim \mathcal{U}(-\sqrt{\frac{3}{n}}, \sqrt{\frac{3}{n}})$$

- Nếu khởi tạo từ Gaussian:  $w_i \sim \mathcal{N}(0, \frac{1}{n})$

**Trường hợp activation là sigmoid** Từ khai triển Taylor:

$$\sigma(x) \approx \frac{1}{2} + \frac{x}{4} \Rightarrow \text{Var}(\sigma(x)) \approx \frac{1}{16} \cdot \text{Var}(x)$$

Suy ra:

$$\text{Var}(Z_i) = n \cdot \text{Var}(x_i) \cdot \text{Var}(w_i) \Rightarrow \text{Var}(A_i) = \frac{1}{16} \cdot \text{Var}(Z_i) = \frac{n}{16} \cdot \text{Var}(x_i) \cdot \text{Var}(w_i) \Rightarrow \text{Var}(w_i) = \frac{16}{n}$$

- Nếu khởi tạo từ Uniform:

$$\frac{(2r)^2}{12} = \frac{16}{n} \Rightarrow r = \sqrt{\frac{48}{n}} = 4\sqrt{\frac{3}{n}} \Rightarrow w_i \sim \mathcal{U}(-4\sqrt{\frac{3}{n}}, 4\sqrt{\frac{3}{n}})$$

- Nếu khởi tạo từ Gaussian:  $w_i \sim \mathcal{N}(0, \frac{16}{n})$

### 6.1.6 He Initialization

He Initialization áp dụng khi activation là ReLU. Do  $\text{ReLU}(x) = \max(0, x)$  không khả vi tại 0 và khó khai triển Taylor, ta xét theo kỳ vọng.

Vì  $Z \sim \mathcal{N}(0, \sigma^2)$  nên:

$$\begin{aligned} \mathbb{E}[A] &= \mathbb{E}[\max(0, Z)] = \int_0^\infty z \cdot f(z) dz = \frac{1}{2}\mathbb{E}[Z] = 0 \quad (\text{do } \mathbb{E}[Z] = 0) \\ \text{Var}(A) &= \int_0^\infty z^2 f(z) dz = \frac{1}{2}\mathbb{E}[Z^2] = \frac{1}{2} \cdot \text{Var}(Z) \end{aligned}$$

Do đó:

$$\text{Var}(Z_i) = n \cdot \text{Var}(x_i) \cdot \text{Var}(w_i) \Rightarrow \text{Var}(A_i) = \frac{1}{2} \cdot \text{Var}(Z_i) \Rightarrow \text{Var}(w_i) = \frac{2}{n}$$

- Nếu khởi tạo từ Uniform:

$$\frac{(2r)^2}{12} = \frac{2}{n} \Rightarrow r = \sqrt{\frac{6}{n}} \Rightarrow w_i \sim \mathcal{U}(-\sqrt{\frac{6}{n}}, \sqrt{\frac{6}{n}})$$

- Nếu khởi tạo từ Gaussian:  $w_i \sim \mathcal{N}(0, \frac{2}{n})$

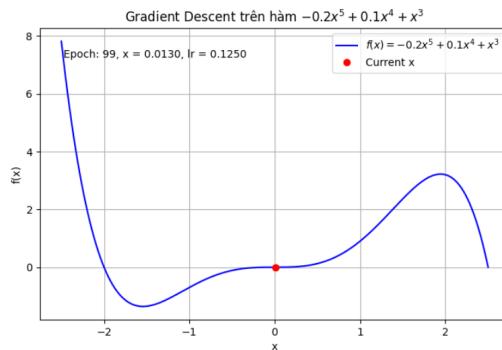
### 6.2 Optimization

Như đã trình bày trong các phần trước, ta biết rằng quá trình *tối ưu hóa* (optimization) giúp cập nhật các ma trận tham số sao cho mô hình có thể xây dựng được một siêu mặt phẳng (*hyperplane*) phù hợp nhất để phân tách dữ liệu huấn luyện. Tuy nhiên, khi áp dụng MLP (Multi-layer Perceptron) cho bài toán phân loại ảnh, nhiều vấn đề phát sinh mà ta cần khảo sát kỹ lưỡng.

### 6.2.1 Limitation of SGD

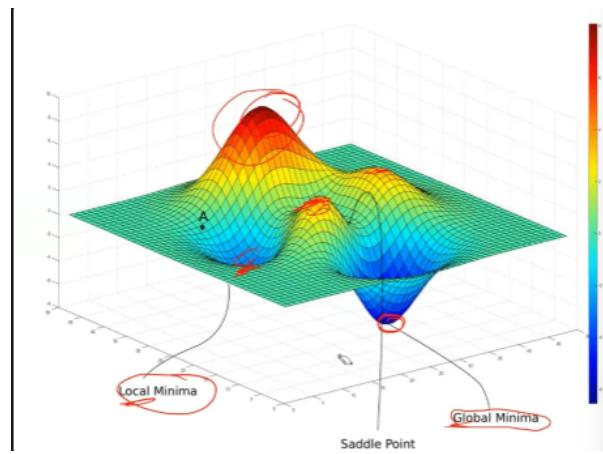
Trong không gian tham số nhiều chiều (high-dimensional search space), thuật toán tối ưu dễ gặp các điểm cực trị địa phương (*local minima*) hoặc các điểm yên ngựa (*saddle point*) khiến quá trình hội tụ gặp khó khăn.

Mặc dù ý tưởng của Stochastic Gradient Descent (SGD) là cập nhật ngược hướng đạo hàm, nhưng nếu rơi vào điểm yên ngựa thì gradient có thể gần như bằng 0, khiến các tham số không được cập nhật và mô hình không học được gì thêm.



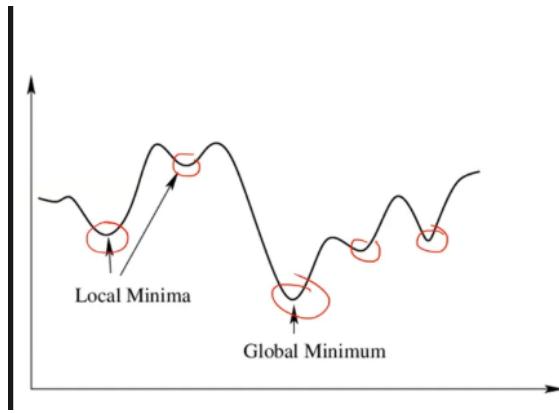
Hình 57: Mô phỏng điểm yên ngựa trong không gian hàm mất mát

Trong logistic regression, ta đã biết rằng các hàm kích hoạt (activation function) như sigmoid làm cho hàm mất mát không còn lồi (non-convex). Khi áp dụng MLP với nhiều lớp ẩn và hàm kích hoạt phi tuyến như ReLU, hàm mất mát trở nên phức tạp và chứa nhiều điểm cực trị. Vì không thể biết trước số lượng lớp ẩn hoặc hình dạng chính xác của không gian hàm mất mát, nên việc đảm bảo tính lồi của hàm mất mát là bất khả thi. Do đó, mục tiêu của ta là tìm các kỹ thuật tối ưu hóa sao cho vẫn có thể đạt được điểm cực tiểu toàn cục trong không gian chứa nhiều cực tiểu địa phương và điểm yên ngựa.



Hình 58: Không gian hàm mất mát phức tạp với nhiều cực trị

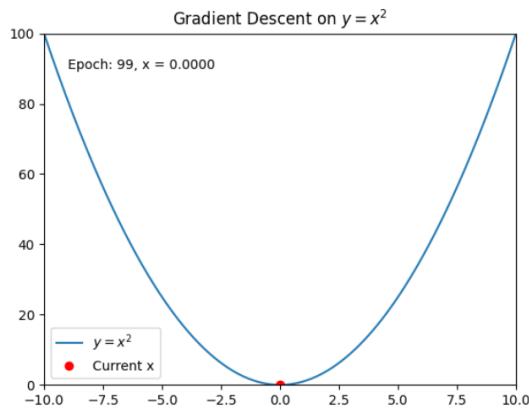
Để đơn giản hóa, ta có thể xem xét không gian hàm mất mát một chiều, sau đó tổng quát hóa việc cập nhật tham số theo đạo hàm theo từng hướng:



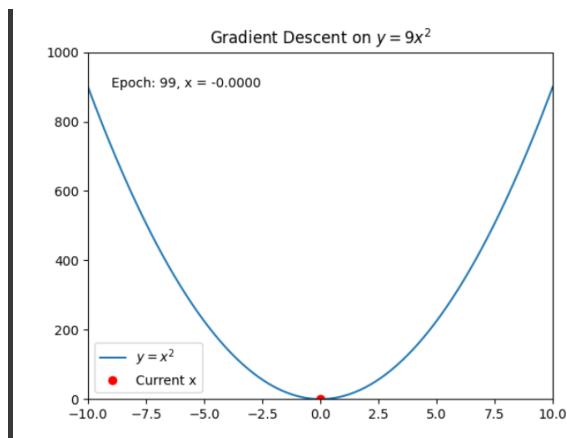
Hình 59: Không gian măt măt 1 chiều minh họa

Một vấn đề lớn của SGD là việc chọn *learning rate* (lr) phù hợp. Với một learning rate cố định, mô hình có thể:

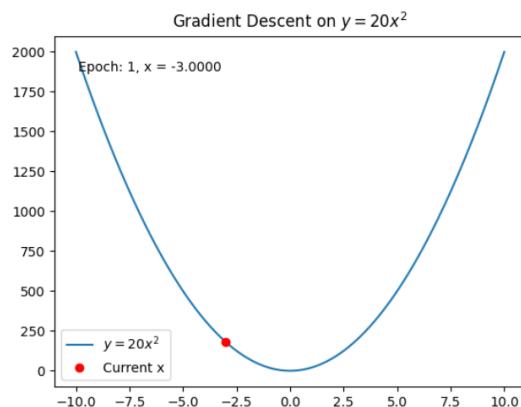
- Hội tụ chậm (nếu lr quá nhỏ) - Bị dao động (*zigzag*) hoặc phân kỳ (nếu lr quá lớn)
- Ví dụ với hàm số  $f(x) = ax^2$ :



Hình 60: Hàm số  $x^2$



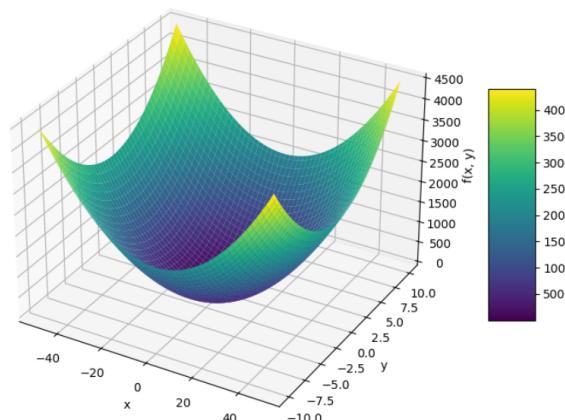
Hình 61: Hàm số  $9x^2$



Hình 62: Hàm số  $20x^2$

Khi lr = 0.01: -  $x^2$  hội tụ mượt mà -  $9x^2$  dao động nhẹ -  $20x^2$  phân kỳ  
Điều này cho thấy learning rate cần điều chỉnh tương ứng với quy mô dữ liệu.  
Xét một ví dụ khác với hàm  $f(x, y) = x^2 + 20y^2$ :

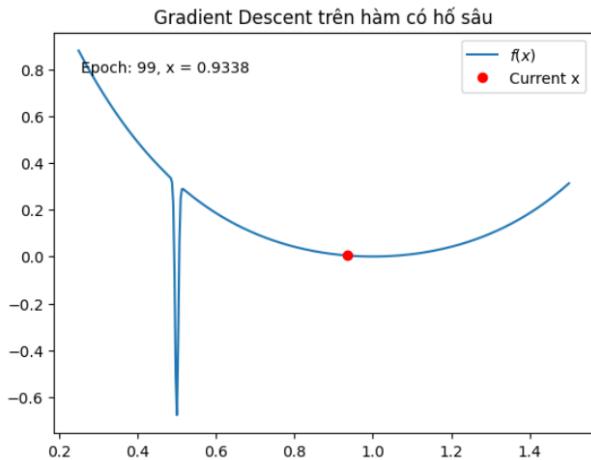
$$f(x, y) = x^2 + 20y^2$$



Hình 63: Hàm số  $x^2 + 20y^2$

Nếu sử dụng cùng một learning rate cho cả hai chiều, ta dễ gặp vấn đề: - Nếu lr lớn:  $20y^2$  phân kỳ - Nếu lr nhỏ:  $x^2$  hội tụ rất chậm

**Tiếp cận 2:** Với learning rate quá lớn, mô hình có thể bỏ qua cực trị tối ưu:

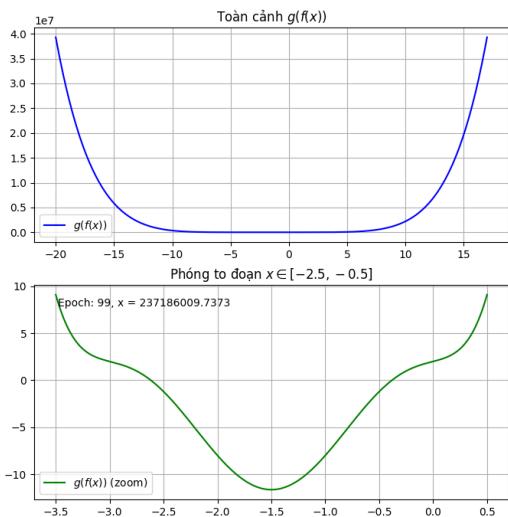


Hình 64: Minh họa mô hình bỏ qua điểm cực trị sâu

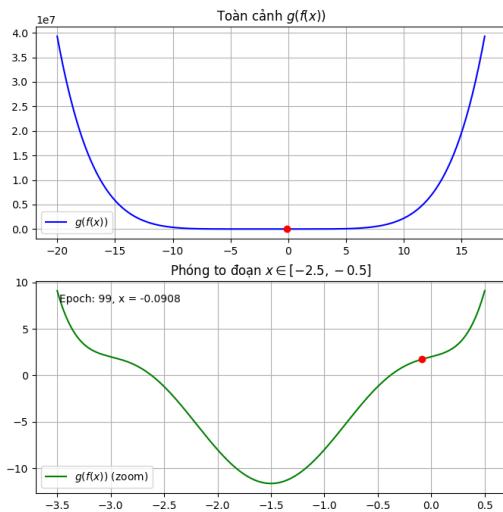
**Tiếp cận 3:** Xét hàm hợp  $g(f(x))$ , với:

$$f(x) = x^2 + 3x, \quad g(x) = x^3 + x + 2$$

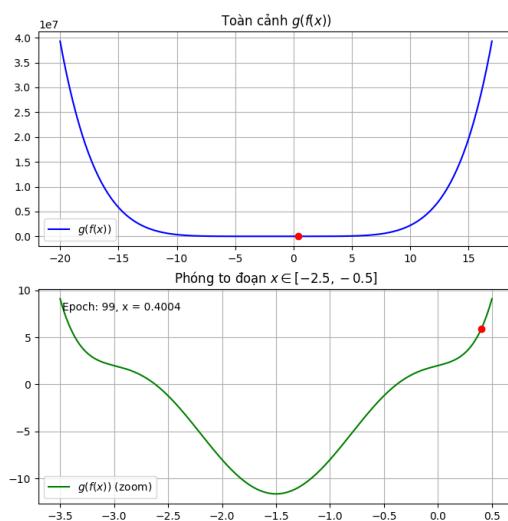
Ta khảo sát hiệu quả tối ưu hóa với các learning rate khác nhau:



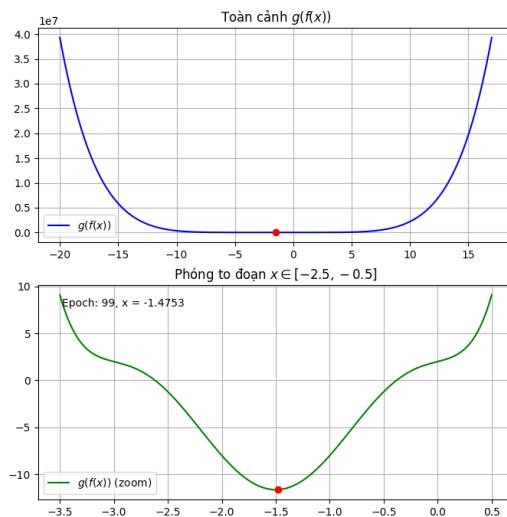
Hình 65: Phân kỳ với lr = 0.01



Hình 66: Zigzag với lr = 0.001

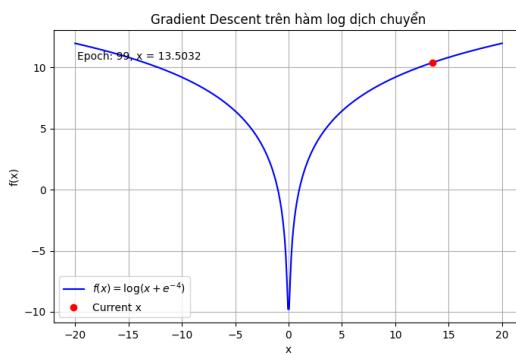


Hình 67: Hội tụ mượt với lr = 0.0001



Hình 68: Tiến nhanh về cực tiểu với lr = 0.0005

**Tiếp cận 4:** Xét hàm cực đoan:  $\log(x^4 + 10^{-5})$



Hình 69: Hàm mất mát dạng log

Ở các epoch đầu, gradient lớn giúp cập nhật nhanh. Tuy nhiên, khi đến vùng phẳng, nếu lr lớn, gradient nhỏ dễ khiến bước nhảy bật ra khỏi cực tiểu. Nếu dùng lr nhỏ từ đầu thì mô hình học quá chậm.

**Kết luận:** Từ các ví dụ trên, ta thấy rằng learning rate không nên giữ cố định mà cần điều chỉnh linh hoạt — thường là giảm dần theo số epoch. Điều này mở đường cho các kỹ thuật hiện đại như:

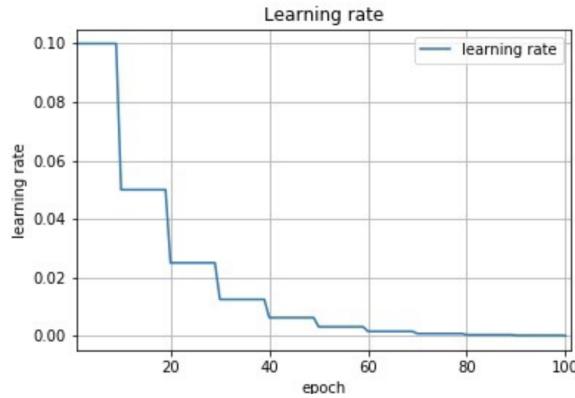
- Learning rate schedule (Step decay, Exponential decay, Cosine annealing, ...)
- Adaptive optimizers (Adam, RMSProp, AdaGrad, ...)

### 6.2.2 Learning Rate Decay

Trong quá trình huấn luyện mô hình học sâu, việc giữ nguyên một giá trị learning rate (LR) cố định trong suốt toàn bộ các epoch có thể gây ra hiện tượng dao động quanh

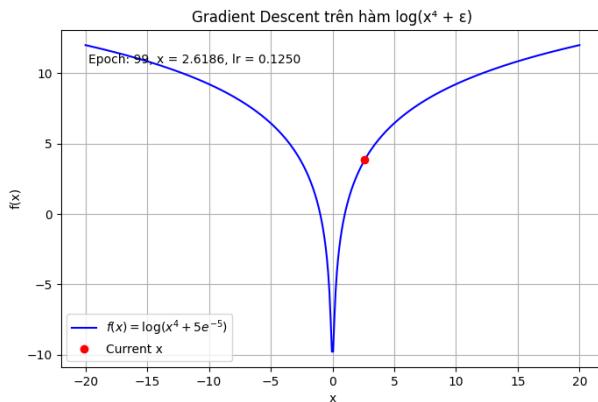
điểm hội tụ hoặc khiến mô hình hội tụ chậm. Do đó, một chiến lược phổ biến là giảm dần learning rate theo thời gian, gọi là *learning rate decay*.

Một cách tiếp cận đơn giản là sử dụng **step decay**, trong đó learning rate sẽ giảm sau mỗi khoảng thời gian cố định là  $p$  epoch. Cụ thể, sau mỗi  $p$  epoch, LR sẽ được nhân với một hệ số  $k$  ( $0 < k < 1$ ). Việc này giúp tốc độ học giảm dần theo thời gian, cho phép mô hình cập nhật trọng số cẩn trọng hơn ở giai đoạn cuối.

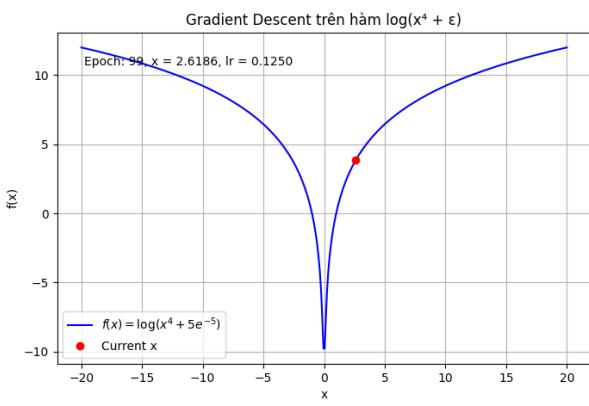


Hình 70: Biểu đồ mô phỏng step decay theo số lượng epoch

Sau khi áp dụng step decay, kết quả huấn luyện thu được cho thấy độ chính xác được cải thiện phần nào:



Hình 71: Độ chính xác khi sử dụng step decay



Hình 72: Hàm mất mát khi sử dụng step decay

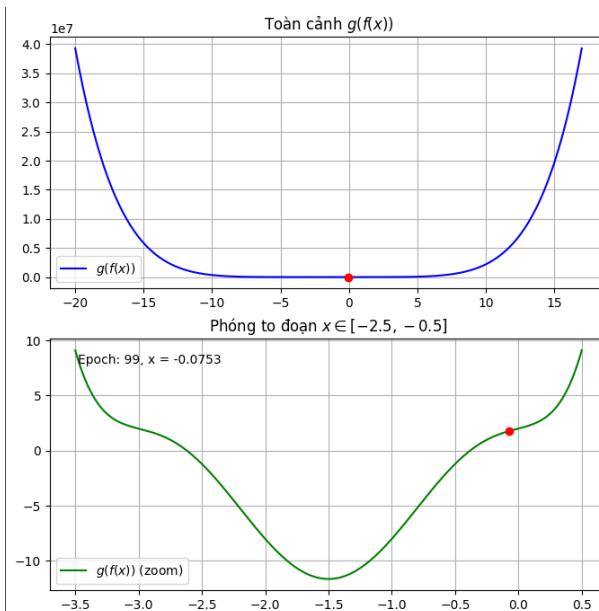
Tuy nhiên, nhược điểm của phương pháp này là việc lựa chọn tham số  $p$  và  $k$  không tuân theo một quy tắc cụ thể nào và mang tính thử nghiệm nhiều, dễ gây cảm giác thiếu hệ thống. Để giải quyết vấn đề đó, một phương pháp khác thường được sử dụng là **exponential decay**, dựa trên hàm mũ. Trong phương pháp này, LR giảm dần một cách liên tục theo thời gian:

$$lr = lr_0 \cdot \lambda^{\frac{\text{step}}{k}}$$

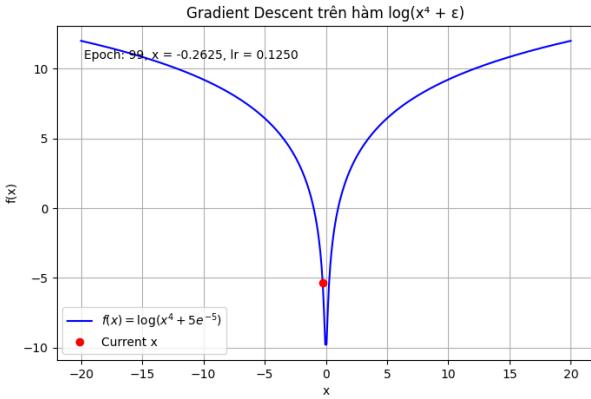
Trong đó:

- $lr_0$ : learning rate ban đầu,
- $\lambda$ : hệ số decay ( $0 < \lambda < 1$ ), điều chỉnh tốc độ giảm,
- $k$ : tham số kiểm soát tốc độ suy giảm.

Với lựa chọn tham số phù hợp, mô hình thu được kết quả khả quan hơn:



Hình 73: Độ chính xác khi áp dụng exponential decay



Hình 74: Hàm mất mát khi áp dụng exponential decay

### 6.2.3 Adagrad

Mặc dù các phương pháp giảm learning rate như *step decay* và *exponential decay* đều có hiệu quả nhất định, chúng yêu cầu người dùng lựa chọn các công thức điều chỉnh bên ngoài, hoặc gọi thêm các hàm điều chỉnh learning rate phụ thuộc thời gian. Tuy nhiên, trong thực tế, nhiều người dùng chuyên nghiệp mong muốn thay đổi trực tiếp công thức cập nhật của thuật toán tối ưu thay vì chỉ bổ sung một hàm biến đổi learning rate.

Ý tưởng là tìm một đại lượng  $k_t$  có thể tăng chập hoặc giảm chập theo thời gian huấn luyện (epoch). Nếu  $f(t)$  là một hàm giảm chập, ta có thể scale nó về khoảng  $[0, 1]$  rồi nhân với learning rate gốc để thực hiện điều chỉnh. Ngược lại, nếu  $f(t)$  tăng chập, ta chỉ cần chia learning rate cho  $f(t)$ .

Từ đó, ta có thể viết cập nhật đơn giản dưới dạng:

$$\text{SGD}(lr \cdot k_t)$$

Một ý tưởng ban đầu là sử dụng hàm  $f(t) = \frac{1}{\text{loss}(t)}$ , bởi vì hàm mất mát thường có xu hướng giảm theo thời gian huấn luyện. Tuy nhiên, cách tiếp cận này tồn tại hai vấn đề lớn:

- Loss không nhất thiết giảm chập; trong quá trình huấn luyện, loss có thể dao động hoặc thậm chí tăng trong một số epoch.
- Loss không có miền giá trị chuẩn hóa ổn định, dẫn đến hệ số learning rate thay đổi không kiểm soát.

Để khắc phục điều đó, một lựa chọn hợp lý hơn là sử dụng **tổng bình phương gradient** được tích luỹ theo thời gian. Đại lượng này chắc chắn tăng dần và có thể xem như một chỉ số phản ánh mức độ tích luỹ thông tin từ quá khứ. Đó cũng chính là ý tưởng cốt lõi của thuật toán **Adagrad**.

Thay vì sử dụng learning rate cố định, Adagrad điều chỉnh learning rate riêng cho từng tham số dựa trên lịch sử gradient. Cụ thể:

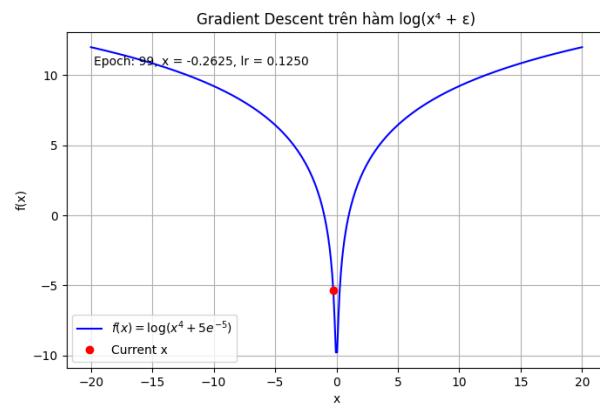
$$\begin{aligned} g_t &= \nabla f(x_{t-1}) \\ s_t &= s_{t-1} + g_t^2 \\ x_t &= x_{t-1} - \frac{lr}{\sqrt{s_t + \epsilon}} \cdot g_t \end{aligned}$$

Trong đó:

- $g_t$ : gradient tại thời điểm  $t$ ,
- $s_t$ : tổng tích luỹ bình phương gradient,
- $\epsilon$ : một hằng số nhỏ để tránh chia cho 0.

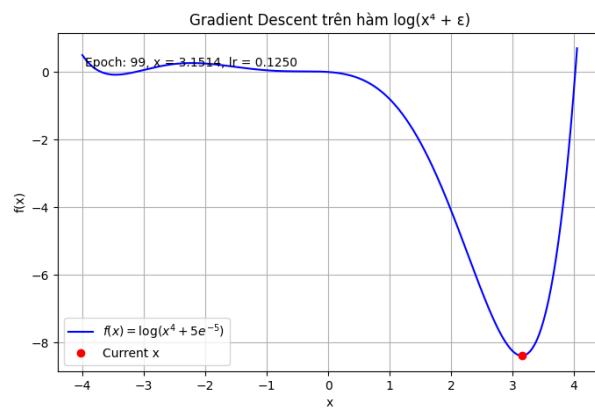
Nhờ đó, những tham số có gradient lớn thường xuyên sẽ được cập nhật ít hơn, còn những tham số hiếm khi thay đổi sẽ có learning rate tương đối cao hơn. Đây là một hướng tiếp cận tự điều chỉnh thông minh, đặc biệt phù hợp cho các bài toán dữ liệu thưa (*sparse data*).

Chúng ta sẽ thử áp dụng idea trên cho các tối ưu của mình



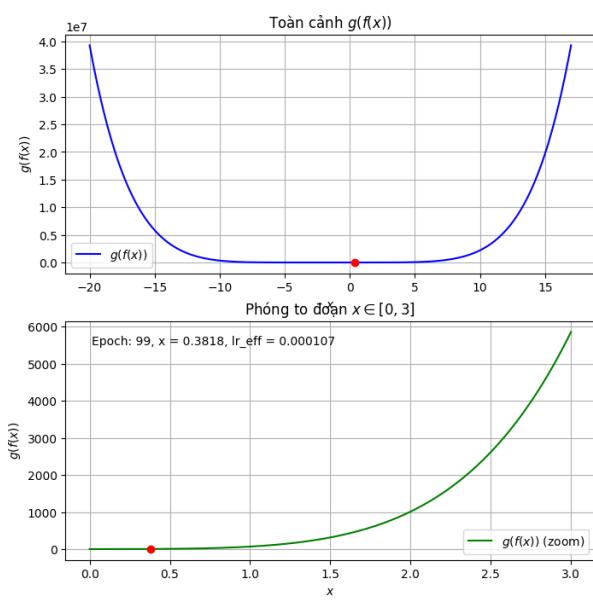
Hình 75: Enter Caption

Ta thấy adagrad đã giải quyết được vấn đề về hàm cực đoan này,



Hình 76: Enter Caption

Một cái thú vị nữa của Ada là khi chúng ta khởi tạo ban đầu cho ada là 1 vùng thoái thì các giá trị đạo hàm nhỏ  $< 1$ , kho đó ta thực hiện phép chia thì lr là tăng lên làm cho chúng ta đi nhanh qua vùng thoái đó



Hình 77:

#### 6.2.4 RMSProp

Một điểm hạn chế rõ rệt của thuật toán AdaGrad là việc tích lũy bình phương gradient theo thời gian khiến tổng đạo hàm tăng dần. Khi mô hình tiến vào các vùng có độ dốc lớn, việc tích lũy này khiến mẫu số trong biểu thức cập nhật tham số trở nên rất lớn, dẫn đến tốc độ học (learning rate) hiệu dụng bị giảm xuống quá nhỏ. Hệ quả là các bước cập nhật trở nên cực kỳ chậm, thậm chí mô hình không còn học được nữa. Do đó, cần một cơ chế để khắc phục vấn đề này.

Một giải pháp trực quan là sử dụng một **cửa sổ trượt** có độ dài  $k$  và tính trung bình hoặc tổng của  $k$  gradient gần nhất. Tuy nhiên, phương pháp này thiếu tính tổng quát và có thể gây sai lệch (bias) do chỉ tập trung vào một khoảng thời gian ngắn.

Vì vậy, người ta đề xuất áp dụng kỹ thuật trung bình trượt có trọng số theo thời gian, vốn là một khái niệm phổ biến trong xử lý chuỗi thời gian (time-series), để xây dựng cơ chế cập nhật mềm dẻo hơn. Cụ thể, tại mỗi thời điểm  $t$ , giá trị bình phương gradient tích lũy được cập nhật theo công thức:

$$V_t = \rho V_{t-1} + (1 - \rho) s_t$$

Công thức trên có dạng rất giống với biểu thức nội suy tuyến tính giữa hai điểm  $y$  và  $z$ :

$$x = \lambda y + (1 - \lambda) z$$

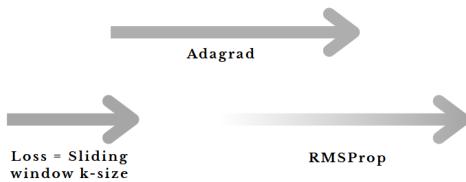
Ý tưởng đằng sau phương pháp này là không hoàn toàn tin tưởng vào giá trị mới tại thời điểm hiện tại, do nó có thể chứa nhiều hoặc dao động bất thường. Thay vào đó, việc kết hợp thông tin từ quá khứ giúp làm mượt quá trình học, từ đó dẫn đến các bước cập nhật ổn định hơn. Trong bối cảnh học sâu, đây là một kỹ thuật hữu hiệu để **làm suy giảm ảnh hưởng của gradient lớn đột biến**, tránh việc learning rate bị thu nhỏ quá mức.

Áp dụng ý tưởng trên, RMSProp cập nhật tốc độ học dựa trên giá trị trung bình trượt của bình phương gradient như sau:

$$\begin{aligned}
g_t &= \nabla f(x_{t-1}) \\
s_t &= \rho \cdot s_{t-1} + (1 - \rho) \cdot g_t^2 \\
x_t &= x_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} \cdot g_t
\end{aligned}$$

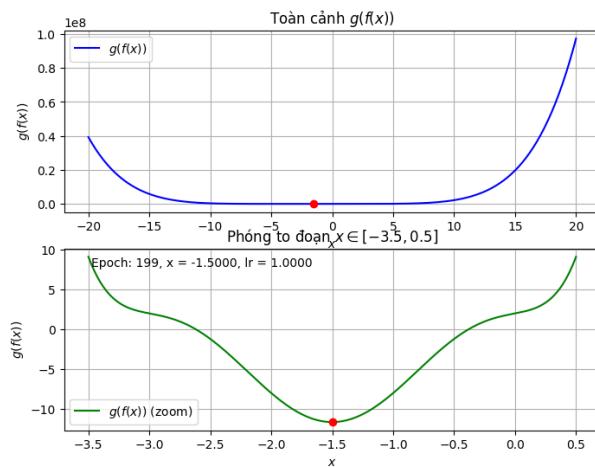
Trong đó:

- $g_t$ : gradient của hàm loss tại thời điểm  $t$ ,
- $s_t$ : trung bình trượt của bình phương gradient,
- $\rho$ : hệ số suy giảm (thường lấy giá trị khoảng 0.9),
- $\epsilon$ : hằng số nhỏ để tránh chia cho 0,
- $\eta$ : tốc độ học ban đầu (initial learning rate).



Hình 78: Minh họa cơ chế cập nhật của RMSProp

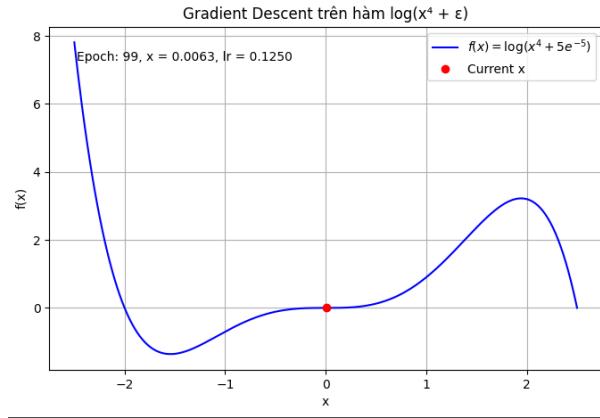
Sau khi triển khai, kết quả huấn luyện thể hiện sự ổn định tốt hơn so với AdaGrad, như thể hiện trong hình minh họa bên dưới:



Hình 79: Kết quả huấn luyện sử dụng RMSProp

Tuy nhiên, RMSProp vẫn gặp khó khăn khi mô hình bị mắc kẹt tại **saddle point** hoặc **cực tiểu địa phương**. Nguyên nhân là vì thuật toán chỉ dựa trên thông tin của đạo hàm tại từng thời điểm và cập nhật ngược lại theo hướng gradient, điều này có thể gây dao động hoặc dừng lại ở các điểm không tối ưu.

Sau khi đã tối ưu cơ chế điều chỉnh tốc độ học, bước tiếp theo là cải thiện quá trình cập nhật đạo hàm bằng cách kết hợp với **momentum**, một khái niệm khác giúp mô hình vượt qua các vùng "trap" nói trên.



Hình 80: Khả năng bị mắc kẹt của mô hình tại saddle point

### 6.2.5 Momentum

Xét một bài toán tối ưu hoá có đồ thị của hàm số  $f(x) = -0.2x^5 + 0.1x^4 + x^3 - 0.5x$ . Hàm số này là một mỏ rộng của một hàm có *saddle point*, nhưng có thêm thành phần  $-0.5x$  khiến hàm xuất hiện cực tiểu địa phương (*local minimum*). Với các thuật toán tối ưu học được trước đó, ta chỉ đơn thuần di chuyển theo hướng ngược với gradient. Tuy nhiên, điều này là chưa đủ để thoát khỏi các điểm yên ngựa hoặc vượt qua các dốc nhỏ trong hàm matsu matsu.

Để minh họa, hãy tưởng tượng đồ thị hàm số là một con dốc và điểm đang xét là một viên bi. Khi thả viên bi từ đỉnh, nếu có đủ động lượng, viên bi có thể vượt qua các dốc nhỏ mà không bị "kẹt". Động lượng này chính là cảm hứng để xây dựng thuật toán **Momentum**. Ý tưởng là tích luỹ các gradient trong quá khứ, giống như cách một quả bóng nặng khi lăn sẽ tích luỹ lực từ những lần va chạm và gia tốc trước đó.

Thay vì tin tưởng tuyệt đối vào gradient hiện tại, ta coi đó là một quan sát có nhiễu. Do đó, ta kết hợp cả các gradient trong quá khứ và hiện tại để quyết định hướng di chuyển. Việc kết hợp này được thực hiện bằng trung bình động có trọng số, tương tự như trong các bài toán chuỗi thời gian (*time series*) nhằm loại bỏ nhiễu.

Công thức cập nhật như sau:

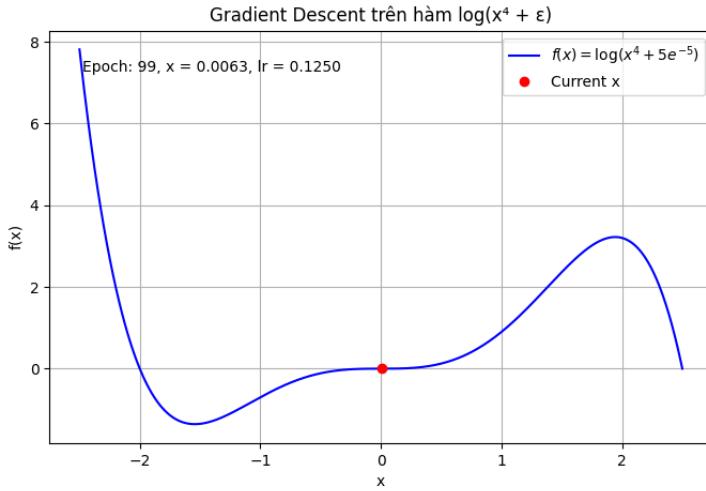
$$\begin{aligned} v_t &= \beta v_{t-1} + (1 - \beta)\alpha \nabla L(\theta_{t-1}) \\ &= mv_{t-1} + lr \cdot \nabla L(\theta_{t-1}) \\ \theta_t &= \theta_{t-1} - v_t \end{aligned}$$

Trong đó:

- $v_t$ : vận tốc tích luỹ tại bước  $t$ , đại diện cho hướng di chuyển tổng hợp từ quá khứ,
- $\beta$ : hệ số momentum (thường từ 0.9 đến 0.99), điều chỉnh mức độ tin tưởng vào quá khứ,
- $\alpha$ : tốc độ học (learning rate), kiểm soát bước nhảy ở mỗi lần cập nhật,

- $\nabla L(\theta_{t-1})$ : gradient của hàm mất mát tại bước  $t - 1$ .

**Vai trò của  $\alpha$ :** Tham số  $\alpha$  đóng vai trò như một bộ điều chỉnh kích thước bước cập nhật. Dù  $v_t$  biểu diễn hướng di chuyển đã được làm mượt qua thời gian, nhưng không có  $\alpha$ , tốc độ thay đổi của tham số có thể không được kiểm soát tốt. Khi kết hợp với  $\beta$ ,  $\alpha$  đảm bảo rằng tổng thể bước đi của thuật toán vẫn được điều chỉnh đúng tỉ lệ với hàm mất mát. Do đó,  $\alpha$  là yếu tố thiết yếu để đảm bảo sự ổn định và hiệu quả trong quá trình huấn luyện.



Hình 81: Minh họa về khả năng vượt dốc nhỏ nhờ Momentum

Và như thế khi chúng ta áp dụng vào đồ thị trên thì chúng ta đã có thể đưa được x về vùng cực tiểu

### 6.2.6 Adam

Sau khi chúng ta đã đi qua về cách chọn lr và cách đưa momentum vào bài toán, vậy sao chúng ta không áp dụng cả 2 concept ấy và đưa vào 1 thuật toán

TA viết lại công thức **Momentum**

$$\begin{aligned} v_t &= \beta v_{t-1} + (1 - \beta) \alpha \nabla L(\theta_{t-1}) \\ &= mv_{t-1} + lr \cdot \nabla L(\theta_{t-1}) \\ \theta_t &= \theta_{t-1} - v_t \end{aligned}$$

### RMSProp

$$\begin{aligned} g_t &= \nabla f(x_{t-1}) \\ s_t &= \rho \cdot s_{t-1} + (1 - \rho) \cdot g_t^2 \\ x_t &= x_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} \cdot g_t \end{aligned}$$

Chúng ta cần điều chỉnh để giảm bớt lượng hyperparameter chúng ta bỏ alpha là lr putify của cái mới, và thêm 1 lr cho giá trị cập nhật bên dưới, thế thì đc cái mt nó chỉ còn 1 biến số, đặt biến số đó là Beta 1

### 6.3 Generate advance MLP

Source code: [MLP with Adam](#)

## 7 Convolutional Neural Networks (CNNs)

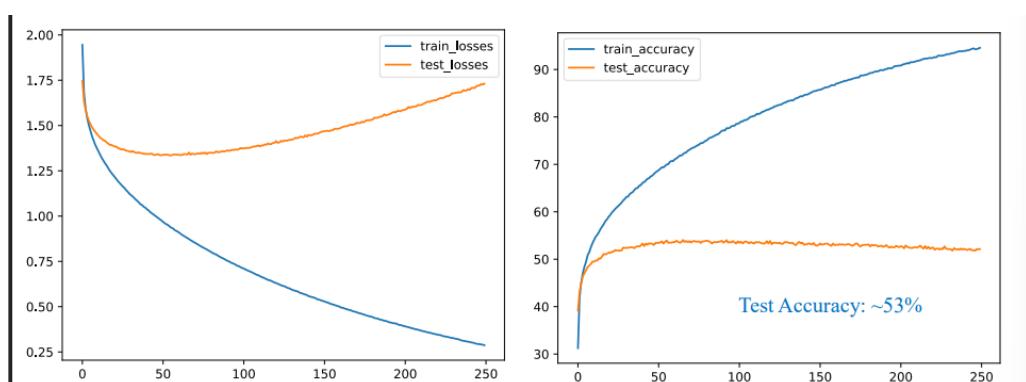
### 7.1 Motivation and Dataset

Sau khi huấn luyện thành công MLP với thuật toán tối ưu Adam trên bộ dữ liệu *Fashion MNIST*, mô hình đạt được độ chính xác cao và được đánh giá tích cực. Tuy nhiên, trong giai đoạn tiếp theo, bộ dữ liệu được thay đổi thành *CIFAR-10*, bao gồm 10 lớp phân loại hình ảnh màu kích thước  $32 \times 32$  với ba kênh RGB (ví dụ: *airplane*, *automobile*, *bird*, v.v.).



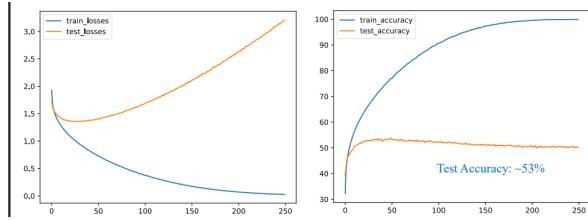
Hình 82: Hình ảnh minh họa từ bộ dữ liệu CIFAR-10

Một giả định ban đầu được đưa ra: vì hình ảnh màu cũng là ma trận  $H \times W$  với chiều sâu 3, nên hoàn toàn có thể áp dụng phương pháp MLP bằng cách `flatten` ảnh thành vector đầu vào với kích thước  $32 \times 32 \times 3 = 3072$  và kết nối trực tiếp tới lớp phân loại. Mô hình đầu tiên được thiết lập với kiến trúc `Linear(3072, 10)` cho kết quả sau:

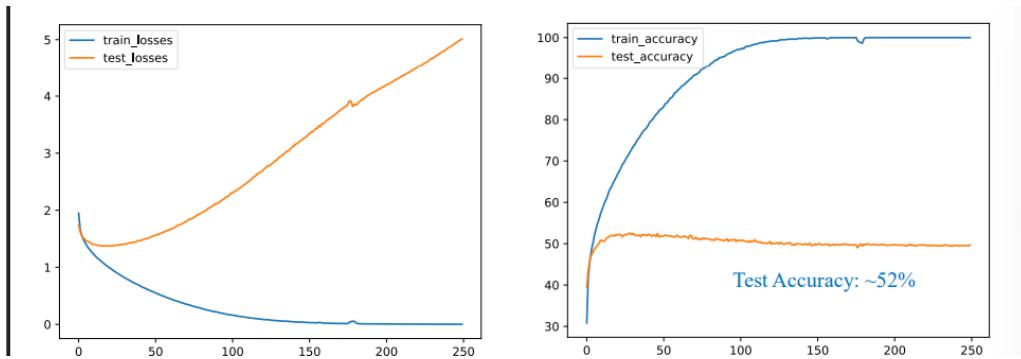


Hình 83: MLP với 1 hidden layer (256 nodes)

Mặc dù mô hình đạt độ chính xác gần 100% trên tập huấn luyện, độ chính xác trên tập kiểm tra chỉ đạt khoảng 53%, cho thấy hiện tượng *overfitting*. Việc tăng số lượng tầng ẩn được thực hiện nhằm cải thiện kết quả:



Hình 84: MLP với 2 hidden layers (256–256)



Hình 85: MLP với 3 hidden layers (256–256–256)

Như dự đoán, mô hình hội tụ nhanh hơn khi tăng số tầng ẩn. Tuy nhiên, hiện tượng *overfitting* vẫn xảy ra rõ rệt: *test loss* tăng dần và *test accuracy* gần như không cải thiện.

## 7.2 Hạn chế của MLP trong xử lý ảnh màu

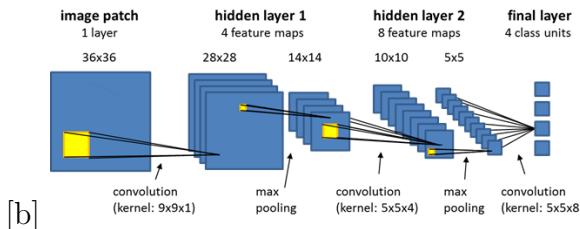
Sự khác biệt giữa hai bộ dữ liệu nằm ở đặc trưng hình ảnh: *Fashion MNIST* là ảnh xám với foreground rõ ràng, trong khi *CIFAR-10* là ảnh màu, khiến việc phân tách foreground và background trở nên khó khăn. Điều này làm tăng tính đa dạng và phức tạp của dữ liệu, khiến mô hình dễ bị học tủ, đặc biệt khi màu sắc của background có thể trùng với màu của vật thể ở lớp khác.

Bên cạnh đó, việc **flatten** ảnh thành vector đầu vào vô tình làm mất đi thông tin không gian (spatial information) giữa các điểm ảnh. Điều này khiến mô hình phải học thêm mối quan hệ giữa các điểm ảnh mà trước đó vốn đã tồn tại theo cách trực quan trong ảnh gốc.

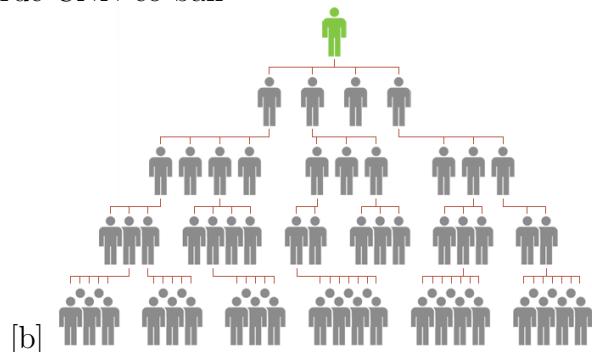
Một vấn đề triết lý khác được đặt ra: thị giác của con người không xử lý ảnh dưới dạng vector 1D mà tiếp nhận thông tin theo từng vùng cụ thể của ảnh. Điều này thúc đẩy ý tưởng rằng: thay vì kết nối toàn bộ đầu vào đến mỗi neuron, mô hình nên tập trung vào các vùng nhỏ hơn (local receptive fields). Đây chính là động lực cho sự ra đời của mạng nơ-ron tích chập (CNN).

### 7.3 Giới thiệu về CNN

CNN tận dụng các lớp tích chập để khai thác thông tin không gian cục bộ. Càng tiến sâu vào các tầng, phạm vi tiếp nhận (receptive field) của các neuron càng lớn, cho phép mô hình dần dần nắm bắt được cấu trúc tổng thể của hình ảnh.



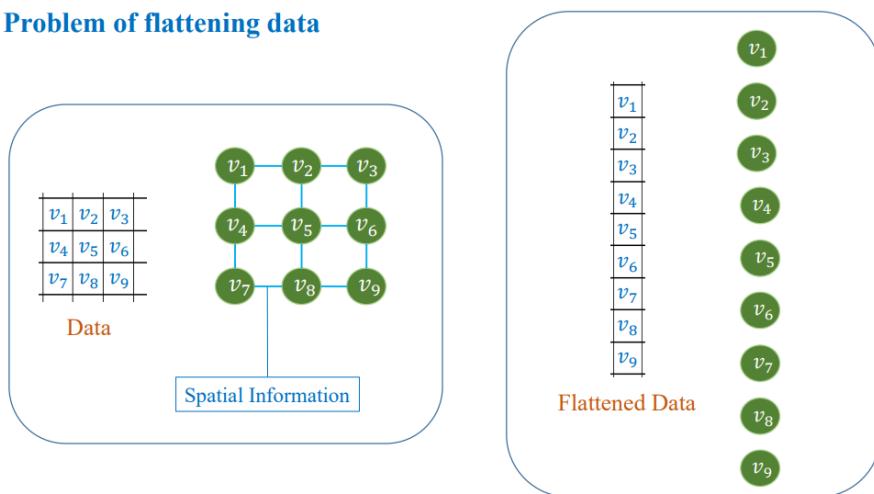
Hình 86: Kiến trúc CNN cơ bản



Hình 87: Kiến trúc CNN nhiều cấp độ

Hình 88: So sánh giữa kiến trúc CNN cơ bản và kiến trúc CNN đa cấp độ

#### Problem of flattening data



Hình 89: So sánh giữa MLP và CNN

Hình 89 minh họa rõ sự khác biệt: trong ảnh đầu vào  $3 \times 3$ , các điểm ảnh gần nhau có khả năng cao cùng màu và có liên hệ mật thiết. Việc **flatten** ảnh làm mất đi mối quan hệ cục bộ này, khiến mô hình phải học thêm thông tin vốn đã có. Điều này đặc biệt bất lợi trong các bài toán thị giác máy tính, nơi mà tính không gian đóng vai trò then chốt. CNN khắc phục vấn đề này một cách tự nhiên và hiệu quả hơn.

## 7.4 CIFAR-10 dưới góc nhìn Machine Learning truyền thống

---

Mặc dù các mô hình học sâu (Deep Learning) như Convolutional Neural Network (CNN) đã chứng minh hiệu quả vượt trội trong các bài toán phân loại ảnh, các phương pháp Machine Learning truyền thống vẫn có thể giải quyết tốt bài toán phân loại với bộ dữ liệu CIFAR-10 nếu được kết hợp với bước trích xuất đặc trưng (feature extraction) hiệu quả.

### 7.4.1 Trích xuất đặc trưng: Hàm và bộ lọc

Trong học máy truyền thống, quá trình trích xuất đặc trưng đóng vai trò then chốt. Một số phương pháp phổ biến để trích xuất đặc trưng trong ảnh bao gồm các bộ lọc biên như:

Cho ảnh gốc như hình bên dưới:



Hình 90: Ảnh gốc trước khi áp dụng bộ lọc

Chúng ta cùng phân tích một số bộ lọc phổ biến và xem ảnh của chúng ta trông như thế nào sau khi áp dụng các bộ lọc này:

- **Sobel Filter:** Dùng để phát hiện biên theo phương ngang hoặc dọc, nhờ vào tính toán đạo hàm riêng tại mỗi điểm ảnh. Có dạng:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

- **Roberts Filter:** Bộ lọc đơn giản phát hiện cạnh bằng cách tính hiệu độ sáng giữa các điểm ảnh chéo. Có dạng:

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

- **Laplacian of Gaussian (LoG):** Phát hiện biên bằng đạo hàm bậc hai của ảnh sau khi làm mượt. Có dạng:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

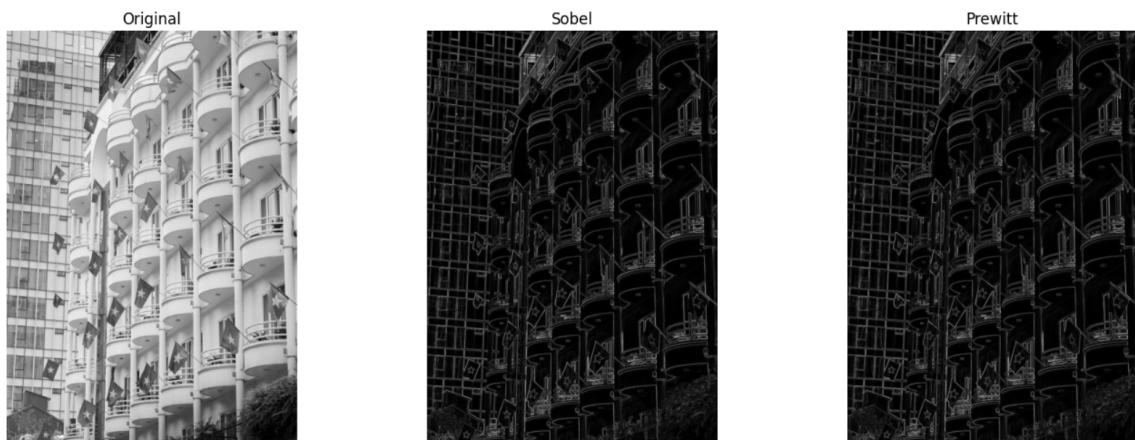
- **Scharr Filter:** Một biến thể cải tiến của Sobel, cho kết quả chính xác hơn khi tính gradient. Có dạng:

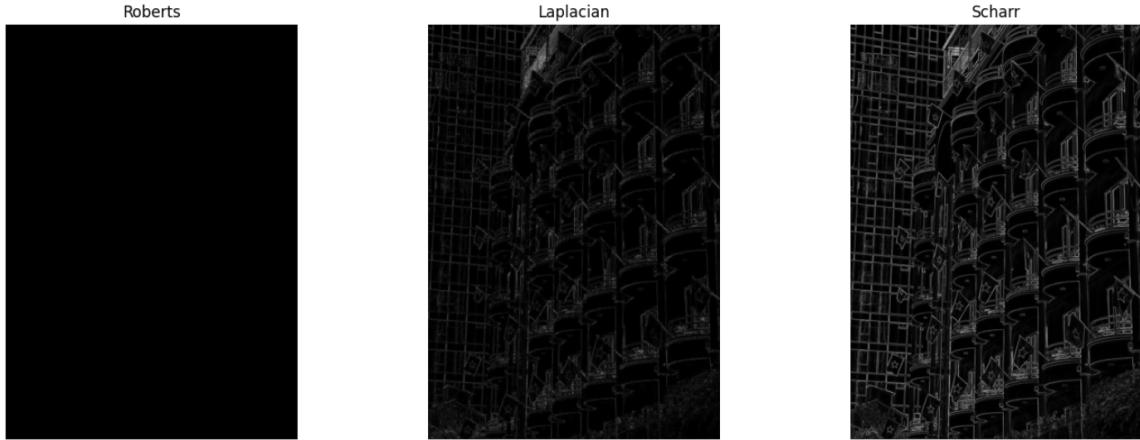
$$\begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$$

- **Prewitt Filter:** Phát hiện biên tương tự Sobel nhưng sử dụng bộ trọng số khác. Có dạng:

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

Sau khi áp dụng các bộ lọc lên ảnh gốc, chúng ta có được kết quả như sau:





Hình 91: So sánh đầu ra các bộ lọc khác nhau

Các bộ lọc trên sẽ giúp chuyển ảnh đầu vào về dạng chứa các đặc trưng rõ ràng hơn (edges, corners, texture...), từ đó hỗ trợ mô hình học máy học tập tốt hơn.

#### 7.4.2 Huấn luyện mô hình

Sau khi đã có tập đặc trưng đầu vào (feature vectors), ta có thể sử dụng các mô hình học máy truyền thống để huấn luyện, ví dụ:

- **Support Vector Machine (SVM):** Tìm siêu phẳng tối ưu phân tách các lớp ảnh dựa trên đặc trưng đã trích xuất. Trong bài toán nhiều lớp như CIFAR-10, ta có thể áp dụng chiến lược One-vs-All hoặc One-vs-One.
- **XGBoost:** Một thuật toán boosting mạnh mẽ, sử dụng cây quyết định để học các đặc trưng phi tuyến phức tạp. Khi kết hợp với đặc trưng trích xuất thủ công từ các bộ lọc, XGBoost có thể đạt hiệu suất cạnh tranh.

Trong một số nghiên cứu, việc kết hợp giữa trích xuất đặc trưng bằng các phương pháp truyền thống và mô hình học máy như SVM, Random Forest hoặc XGBoost có thể mang lại kết quả phân loại ảnh CIFAR-10 khá khả quan, đặc biệt là khi số lượng dữ liệu huấn luyện bị giới hạn hoặc không có GPU để huấn luyện mô hình học sâu.

Mã nguồn tham khảo: [<link>](#)

#### 7.5 Triển khai CNN: Một hướng tiếp cận tích hợp trong học sâu

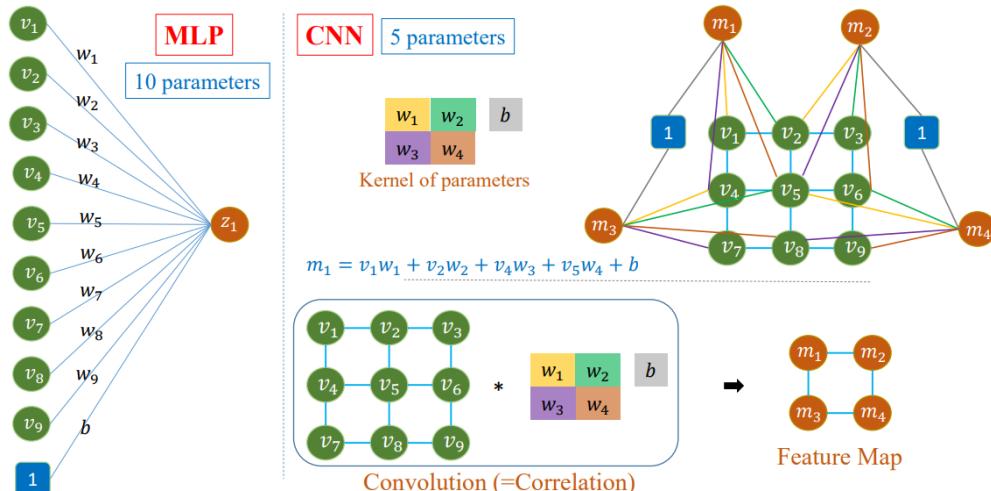
Từ các phương pháp học máy cổ điển đã trình bày, đặc biệt là việc sử dụng các bộ lọc (filter) như Sobel, Roberts, hoặc Laplacian để trích xuất đặc trưng trước khi huấn luyện mô hình SVM, có thể thấy rằng cách tiếp cận này tồn tại hai điểm hạn chế cơ bản:

- **Số lượng và chất lượng bộ lọc không được tối ưu hóa:** Khi chuyên gia lựa chọn thủ công  $k$  bộ lọc để trích xuất đặc trưng, câu hỏi đặt ra là liệu  $k$  đã đủ lớn để bao phủ không gian đặc trưng chưa? Đồng thời, các giá trị cụ thể trong ma trận bộ lọc liệu có thực sự phù hợp với bản chất dữ liệu?
- **Quá trình trích xuất và phân loại tách rời:** Việc trích xuất đặc trưng và huấn luyện mô hình được thực hiện qua hai giai đoạn riêng biệt, dẫn đến thiếu khả năng tối ưu toàn cục thông qua lan truyền ngược (backpropagation).

Các mạng nơ-ron tích chập (CNN) giải quyết triệt để cả hai hạn chế nêu trên thông qua cơ chế học end-to-end. Với số lượng bộ lọc không giới hạn (tùy theo chiều sâu mạng), các tham số trong bộ lọc được cập nhật tự động trong quá trình huấn luyện. Đồng thời, việc trích xuất đặc trưng và phân loại được tích hợp trong cùng một mô hình thống nhất.

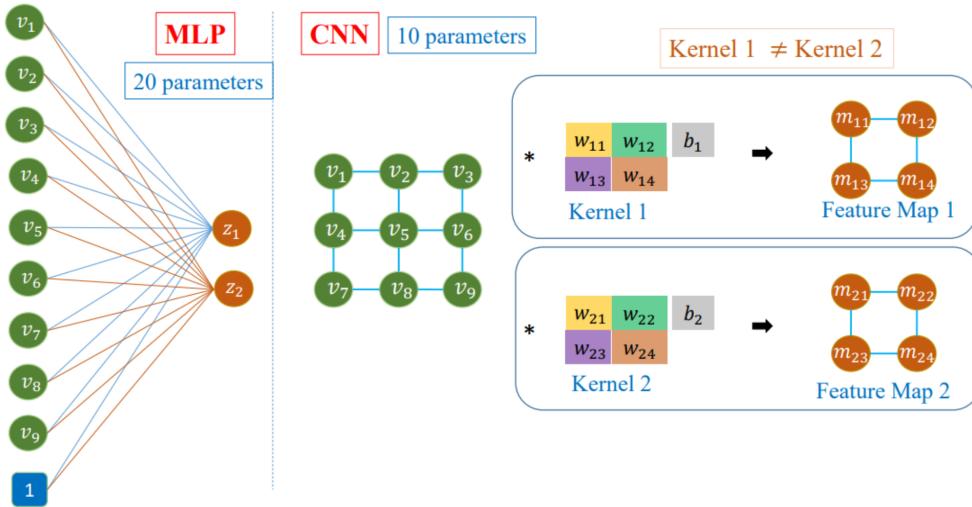
### 7.5.1 Đặc điểm cốt lõi của CNN

**Kernel và phép tích chập** Các phương pháp học máy truyền thống sử dụng các bộ lọc trượt trên ảnh đầu vào để trích xuất đặc trưng cục bộ mà không cần làm phẳng (flatten), do đó giữ lại được cấu trúc không gian của ảnh. Ý tưởng này có thể được mở rộng sang học sâu bằng cách xem các bộ lọc như ma trận tham số—tương tự trọng số ( $w$ ) và hệ số điều chỉnh ( $b$ ) trong MLP. Khi trượt kernel qua từng vùng của ảnh, ta thực hiện phép nhân Hadamard giữa kernel và vùng con tương ứng trong ảnh, sau đó cộng tổng để tạo ra một điểm ảnh trong *feature map*. Phép toán này được gọi là *phép tích chập* (convolution), và feature map được xem như một nút (node) trong mạng MLP.



Hình 92: So sánh cấu trúc MLP và CNN

**Huấn luyện và mở rộng số lượng đặc trưng** Trong quá trình huấn luyện, các ma trận tham số (kernel) được cập nhật thông qua lan truyền ngược. Mỗi feature map được sinh ra bởi một kernel cụ thể. Khi tăng số lượng feature map (để tăng biểu diễn đặc trưng), ta chỉ cần thêm nhiều kernel khác nhau vào cùng một lớp tích chập.



Hình 93: Liên hệ giữa logit trong MLP và feature map trong CNN

**So sánh hiệu quả tham số giữa CNN và MLP** Dù chưa xét đến hiệu năng, CNN có thể đạt được hiệu quả tương đương MLP nhưng với số lượng tham số ít hơn. Nguyên nhân là do CNN chỉ kết nối cục bộ giữa các vùng ảnh, trong khi MLP kết nối toàn cục (fully connected). Do đó, CNN sử dụng tham số một cách hiệu quả hơn và phù hợp hơn với cấu trúc dữ liệu ảnh. Việc xử lý cục bộ không làm mất đi tính biểu diễn; trái lại, theo nguyên lý *chia để trị*, đây là một hướng tiếp cận tối ưu cả trong AI lẫn thực tế. Hơn nữa, trong thị giác máy tính, mục tiêu là mô phỏng cách nhìn của con người—chúng ta cũng chỉ tập trung vào một vùng nhỏ thay vì toàn bộ ảnh ngay lập tức.



Hình 94: Tính cục bộ trong CNN: từ trực giác sinh học đến ứng dụng

**Ảnh màu và xử lý nhiều kênh** Đối với ảnh màu, mỗi điểm ảnh là một vector ba thành phần tương ứng với các kênh RGB. Khi sử dụng kernel để tích chập ảnh màu,

kernel cũng phải có cùng số kênh với ảnh đầu vào (ví dụ:  $3 \times 3 \times 3$  cho ảnh RGB), nhằm đảm bảo phép tích vô hướng được định nghĩa rõ ràng. Như vậy, nếu kernel có kích thước  $3 \times 3$  trên ảnh RGB, số tham số cần học là  $3 \times 3 \times 3 = 27$ , cộng thêm một hệ số bù  $b$ .

Mặc dù trực giác cho ta thấy ảnh màu là một vector 3 chiều, nhưng ta không xử lý từng kênh một cách tách biệt. Thay vào đó, ta coi đó là một thực thể thống nhất (một điểm ảnh màu) và xử lý kernel theo không gian ba chiều. Việc xử lý như vậy cho phép CNN học được các mối tương quan chéo giữa các kênh màu, thay vì chỉ giới hạn trong từng kênh riêng biệt.

### 7.5.2 Sải bước (Stride)

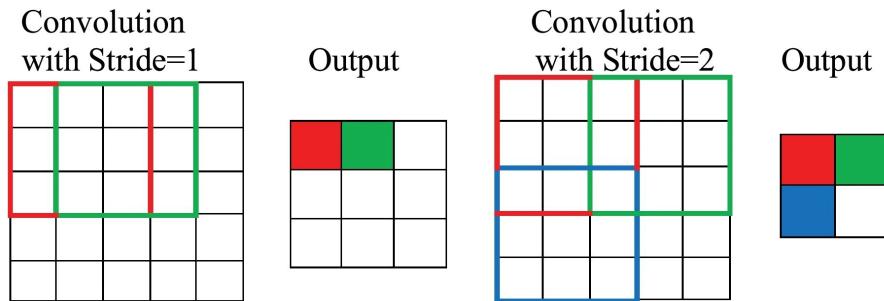
Trong quá trình thực hiện phép tương quan chéo (*cross-correlation*), cửa sổ tích chập (*convolution window*) thường được đặt tại góc trên bên trái của đầu vào, sau đó dịch chuyển sang phải và xuống dưới để bao phủ toàn bộ không gian ảnh. Trong các ví dụ cơ bản, bước dịch chuyển này thường là một điểm ảnh mỗi lần, tức là sải bước bằng 1 theo cả chiều dọc và chiều ngang.

Tuy nhiên, để cải thiện hiệu suất tính toán hoặc chủ đích giảm kích thước đầu ra, ta có thể sử dụng giá trị sải bước lớn hơn. Khi đó, cửa sổ tích chập sẽ bỏ qua một số vị trí ở giữa, dẫn đến ma trận đặc trưng (*feature map*) có kích thước nhỏ hơn so với khi dùng sải bước bằng 1.

Ta định nghĩa **sải bước** (*stride*) là số lượng điểm ảnh mà cửa sổ tích chập di chuyển qua mỗi lần theo một chiều nhất định. Nếu ký hiệu  $S_D$  là kích thước đầu vào,  $K$  là kích thước kernel (giả sử là hình vuông), và  $S$  là sải bước, thì kích thước đầu ra  $S_o$  được tính theo công thức sau:

$$S_o = \left\lfloor \frac{S_D - K}{S} \right\rfloor + 1$$

Trong đó, ký hiệu  $\lfloor \cdot \rfloor$  biểu thị hàm lấy phần nguyên xuống.



Hình 95: Minh họa cơ chế hoạt động của sải bước trong tích chập

Việc lựa chọn giá trị sải bước phù hợp ảnh hưởng trực tiếp đến độ phân giải không gian của đặc trưng đầu ra, từ đó ảnh hưởng đến khả năng học của mô hình. Sải bước lớn giúp giảm kích thước đầu ra và tốc độ tính toán nhanh hơn, nhưng có thể làm mất một phần thông tin không gian quan trọng.

### 7.5.3 Chiều kênh trong tích chập (Channel Dimension in Convolution)

Trong các phần trước, chúng ta đã nhắc đến khái niệm dữ liệu đa kênh (*multi-channel data*), song chưa trình bày chi tiết về cách xử lý chiều kênh trong phép tích chập. Phần

này sẽ làm rõ cơ chế hoạt động của chiều kên khi thực hiện tích chập trong mạng nơ-ron.

**Đầu vào đa kên.** Lấy ví dụ điển hình là bộ dữ liệu CIFAR-10, trong đó mỗi ảnh có kích thước  $3 \times 32 \times 32$ , tương ứng với 3 kênh màu (RGB), chiều cao và chiều rộng là  $32 \times 32$  pixel. Mặc dù ảnh có độ sâu là 3, nhưng về bản chất mỗi điểm ảnh vẫn chỉ biểu diễn một vị trí không gian cụ thể với 3 giá trị cường độ ánh sáng tương ứng ba kênh. Do đó, khi áp dụng tích chập, kernel phải "tương thích" với chiều sâu của ảnh.

Cụ thể, nếu ảnh đầu vào có  $c_i$  kên, thì mỗi bộ lọc tích chập cũng phải có đúng  $c_i$  kên tương ứng để thực hiện phép tương quan chéo (*cross-correlation*). Giả sử kích thước của kernel theo mỗi chiều không gian là  $k_h \times k_w$ , thì:

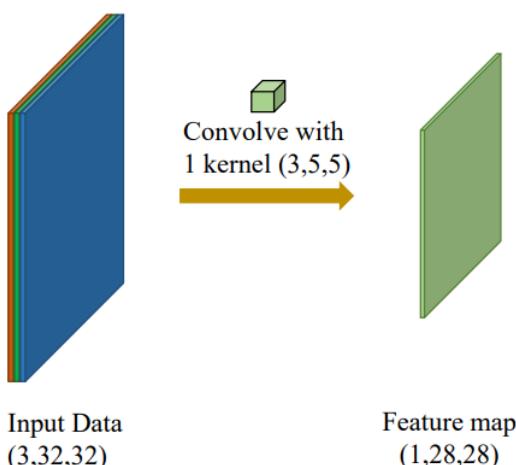
- Nếu  $c_i = 1$ , kernel là một mảng hai chiều  $k_h \times k_w$ .
- Nếu  $c_i > 1$ , kernel sẽ bao gồm  $c_i$  mảng con kích thước  $k_h \times k_w$ , tương ứng với từng kên của đầu vào.

Khi đó, mỗi cặp (kênh đầu vào, kên kernel) sẽ thực hiện phép tương quan chéo riêng biệt để tạo ra một mảng hai chiều. Sau đó, ta cộng tất cả các kết quả lại theo từng phần tử để thu được một mảng hai chiều duy nhất – chính là **feature map** tương ứng với kernel đó. Tóm lại, nếu đầu vào có  $c_i$  kên thì kernel cũng phải có kích thước  $c_i \times k_h \times k_w$  để thực hiện tích chập hợp lệ.

### Chiều kên trong các lớp tích chập

**Đầu vào đa kên:** Trong các phần trước, ta đã đề cập đến khái niệm dữ liệu đa kên, tuy nhiên vẫn chưa phân tích cụ thể cách hoạt động của tích chập trong trường hợp này. Lấy ví dụ tập dữ liệu CIFAR-10, mỗi ảnh màu có kích thước  $3 \times 32 \times 32$ , tức là có ba kênh màu (RGB). Khi áp dụng tích chập, mục tiêu của kernel là tìm mối tương quan cục bộ giữa các điểm ảnh. Do đó, để xử lý đầu vào có độ sâu (số kên) là  $c_i$ , các kernel phải có cùng chiều sâu để mỗi vị trí trên ảnh đầu vào được xử lý toàn diện.

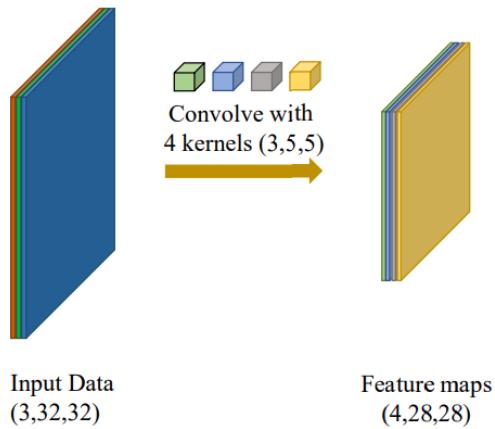
Cụ thể, nếu một bộ lọc tích chập có kích thước không gian là  $k_h \times k_w$  và đầu vào có  $c_i$  kên, thì bộ lọc phải có dạng  $c_i \times k_h \times k_w$ . Khi đó, phép tích chập được thực hiện trên từng cặp kênh tương ứng giữa bộ lọc và đầu vào, rồi cộng lại để tạo thành một mặt phẳng đầu ra (feature map) hai chiều.



Hình 96: Tích chập với đầu vào đa kên

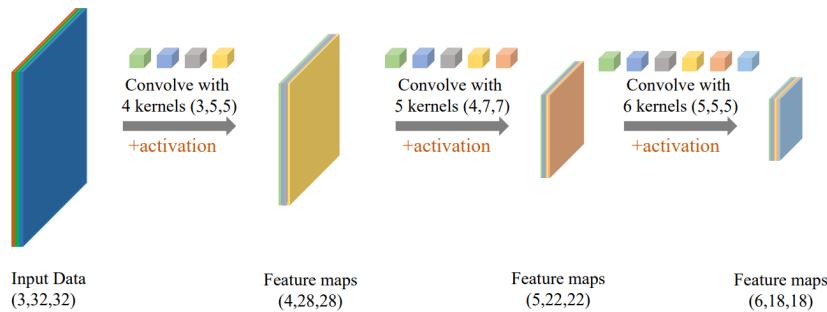
**Đầu ra đa kênh:** Trước đây, khi bàn về mối liên hệ giữa số lượng node và feature map trong mạng MLP, ta đã thấy rằng việc tăng số lượng node giúp tăng khả năng biểu diễn của mô hình. Tương tự, trong mạng tích chập, việc tăng số lượng feature map đầu ra ( $c_o$ ) tương đương với việc sử dụng nhiều bộ lọc tích chập khác nhau để trích xuất các đặc trưng khác nhau từ ảnh đầu vào.

Cụ thể, với  $c_i$  là số kênh đầu vào và  $c_o$  là số kênh đầu ra, ta cần xây dựng  $c_o$  bộ lọc, mỗi bộ lọc có kích thước  $c_i \times k_h \times k_w$ . Mỗi bộ lọc này sẽ tạo ra một feature map hai chiều thông qua phép tích chập với toàn bộ  $c_i$  kênh đầu vào. Sau cùng, ta thu được một tensor đầu ra có kích thước  $c_o \times H' \times W'$ , trong đó  $H'$  và  $W'$  là chiều cao và chiều rộng sau khi tích chập.



Hình 97: Tích chập với đầu ra đa kênh

Từ đó, ta có thể xây dựng một mô hình học sâu bao gồm nhiều lớp tích chập liên tiếp, mỗi lớp sử dụng hàm kích hoạt phi tuyến tính (chẳng hạn ReLU) để gia tăng độ phức tạp của mô hình và khả năng học các đặc trưng phức tạp.



Hình 98: Ví dụ kiến trúc mạng tích chập nhiều tầng

### Ảnh hưởng của kích thước kernel

Cũng giống như MLP cần hàm kích hoạt để mô hình hóa các quan hệ phi tuyến, trong mạng CNN, sau mỗi lớp tích chập ta cũng áp dụng một hàm kích hoạt lên toàn bộ các tensor đầu ra.

Một câu hỏi thường gặp là liệu kích thước kernel lớn có luôn tốt hơn không? Trực giác cho thấy nếu ta muốn bao phủ một vật thể lớn (chẳng hạn một con hươu trong ảnh), thì có vẻ như cần một kernel với kích thước lớn. Tuy nhiên, trong thực tế, thông qua

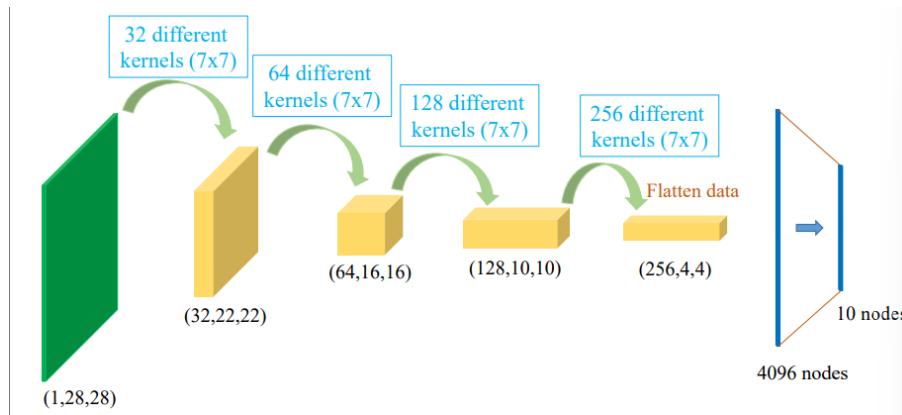
việc xếp chồng nhiều lớp tích chập với kernel nhỏ (ví dụ  $3 \times 3$ ), mỗi điểm ở tầng cao hơn (chẳng hạn tầng  $M_3$ ) sẽ có receptive field rộng hơn, tương ứng với vùng lớn hơn ở ảnh gốc ( $M_1$ ). Do đó, việc sử dụng kernel nhỏ nhưng nhiều lớp tích chập kế tiếp vẫn đảm bảo khả năng bao quát các đặc trưng lớn, đồng thời tiết kiệm tham số hơn.

## Số lượng kernel bao nhiêu là đủ?

Câu trả lời phụ thuộc vào độ phức tạp của bài toán. Tương tự như việc chọn số lượng node trong mạng MLP, nếu số lượng kernel (tức số kênh đầu ra  $c_o$ ) đủ lớn, mô hình sẽ có khả năng học tốt và hội tụ nhanh hơn. Việc dùng quá nhiều kernel có thể gây dư thừa, nhưng trong trường hợp chưa xét đến mô hình phức tạp như GANs, phần dư thừa này thường chỉ ảnh hưởng nhẹ tới hiệu suất, chủ yếu tạo ra nhiều giá trị gần bằng 0. Do đó, trong nhiều tình huống, người ta ưu tiên chọn số lượng kernel tương đối lớn để đảm bảo tính mạnh mẽ (robustness) của mô hình.

## 7.6 Chuyển đổi MLP thành mô hình CNN: Áp dụng trên Fashion-MNIST

Giả sử chúng ta muốn chuyển một mô hình mạng MLP cơ bản sang một kiến trúc CNN và áp dụng trên tập dữ liệu *Fashion-MNIST*. Với đầu vào là ảnh có kích thước  $28 \times 28$ , chúng ta sử dụng một bộ lọc (kernel) có kích thước  $7 \times 7$  để tiến hành tích chập. Mô hình CNN sau khi xây dựng có kiến trúc như hình dưới:



Hình 99: Kiến trúc CNN sử dụng kernel  $7 \times 7$  trên dữ liệu *Fashion-MNIST*

Mục tiêu của chúng ta là thực hiện **rút trích đặc trưng** (feature extraction) từ dữ liệu gốc nhằm tăng mức độ trừu tượng hóa và cô đọng thông tin. Sau khi đặc trưng đã được trích xuất đầy đủ (giả định), ta tiến hành *flatten* đầu ra thành một vector 1 chiều, và đưa vào tầng phân loại dưới dạng một mạng MLP.

## Phân tích chi tiết

- Vì sao kích thước không gian giảm dần, nhưng số lượng kênh lại tăng dần?**  
Dữ liệu ảnh là dữ liệu hai chiều, có tính phân bố rời rạc. Mỗi điểm ảnh riêng lẻ thường không mang quá nhiều thông tin; do đó, ta cần thực hiện tổng hợp không gian thông qua các tầng tích chập và/hoặc pooling. Việc giảm dần kích thước ảnh giúp cô đọng thông tin, làm cho mỗi phần tử đầu ra chứa nhiều thông tin hơn.

Tuy nhiên, việc giảm độ phân giải cũng dẫn đến mất mát thông tin. Để bù đắp điều này, ta cần tăng số lượng kênh (channels) nhằm đảm bảo không gian đặc trưng đủ khả năng biểu diễn các thông tin phức tạp hơn. Tóm lại, việc *giảm kích thước không gian* và *tăng số lượng kênh* là một chiến lược cân bằng giữa độ chi tiết và tính trừu tượng của dữ liệu.

**2. Nếu cuối cùng vẫn dùng MLP, tại sao không dùng MLP từ đầu?** Lý do không dùng MLP ngay từ đầu là vì MLP giả định đầu vào là vector phẳng (flattened vector) và không tận dụng được cấu trúc không gian hai chiều của dữ liệu ảnh. Điều này dẫn đến số lượng tham số rất lớn nếu ảnh đầu vào có kích thước lớn, và mất đi thông tin cục bộ (local patterns).

CNN cho phép khai thác các cấu trúc không gian cục bộ thông qua bộ lọc tích chập, từ đó học được các đặc trưng quan trọng như cạnh, góc, hoặc hình dạng, trước khi chuyển về vector và đưa vào MLP để phân loại. Việc kết hợp CNN và MLP như vậy mang lại hiệu quả học đặc trưng tốt hơn.

**3. Mô hình này có tương tự như việc ghép hai mô hình rời rạc như SVM và bộ lọc thủ công?** Thoạt nhìn, mô hình CNN + MLP có vẻ như là sự ghép nối giữa hai thành phần tách biệt: phần trích đặc trưng (CNN) và phần phân loại (MLP). Tuy nhiên, điểm khác biệt quan trọng là toàn bộ mô hình CNN + MLP được huấn luyện đầu-cuối (end-to-end learning) với cùng một hàm mất mát, thông qua lan truyền ngược (backpropagation). Điều này khác với cách tiếp cận truyền thống của SVM + bộ lọc, nơi các đặc trưng được trích xuất thủ công và cố định.

Hơn nữa, trong các bước tiếp theo, chúng ta có thể xây dựng các kiến trúc hoàn toàn sử dụng tích chập, ví dụ như tầng tích chập  $1 \times 1$ , để thay thế hoàn toàn vai trò của MLP. Khi đó, mô hình sẽ thuần CNN mà vẫn đảm bảo khả năng phân loại mạnh mẽ.

### 7.6.1 Pooling

Trong mạng nơ-ron tích chập (CNN), số lượng tham số thường ít hơn so với mô hình MLP (Multilayer Perceptron). Để khai thác hiệu quả đặc điểm này và nâng cao tốc độ xử lý mà không làm mất nhiều thông tin quan trọng, người ta thường sử dụng kỹ thuật *gộp đặc trưng* (pooling).

Pooling đóng vai trò như một bước trích chọn đặc trưng, nhằm mục đích làm giảm kích thước không gian của biểu diễn dữ liệu. Điều này vừa giúp tăng độ trừu tượng, vừa làm giảm chi phí tính toán. Ý tưởng cốt lõi là: *làm sao để giảm nhanh kích thước biểu diễn với tổn thất thông tin không đáng kể*.



Hình 100: Ảnh gốc qua các bước pooling 2 và 4

**Giảm độ phân giải và tổng hợp thông tin** Trong xử lý ảnh, mục tiêu cuối cùng thường là trả lời các câu hỏi toàn cục, ví dụ như: "Ảnh này có chứa mèo không?". Do đó, các tầng sâu trong mạng cần tích lũy thông tin từ toàn bộ vùng ảnh đầu vào. Bằng cách giảm dần độ phân giải không gian thông qua các tầng pooling, mạng học được biểu diễn toàn cục trong khi vẫn giữ nguyên các lợi thế từ các tầng tích chập.

Ngoài ra, các tầng pooling còn hỗ trợ tính bất biến với dịch chuyển nhỏ. Ví dụ, nếu ảnh đầu vào được dịch sang phải một vài điểm ảnh, các đặc trưng cấp thấp (như cạnh, đường viền) vẫn nên được giữ nguyên ở biểu diễn trừu tượng. Điều này là cần thiết vì trong thực tế, vị trí chính xác của vật thể trong ảnh thường thay đổi nhẹ do rung máy hoặc bối cảnh không ổn định.

**Nguyên lý hoạt động** Tương tự như các tầng tích chập, toán tử pooling sử dụng một cửa sổ kích thước cố định (kernel) trượt trên toàn bộ đầu vào với một bước nhảy (stride) xác định. Tại mỗi vị trí, nó tính một giá trị duy nhất đại diện cho toàn bộ vùng trong cửa sổ đó. Khác với tích chập, tầng pooling không có tham số học được; các phép toán thường được định sẵn, phổ biến nhất là:

- **Max pooling:** lấy giá trị lớn nhất trong vùng cửa sổ.
- **Average pooling:** lấy trung bình cộng của các phần tử trong vùng cửa sổ.

Đối với dữ liệu đầu vào có nhiều kênh (channels), pooling được áp dụng riêng biệt trên từng kênh, tức là số lượng kênh không thay đổi sau khi pooling.

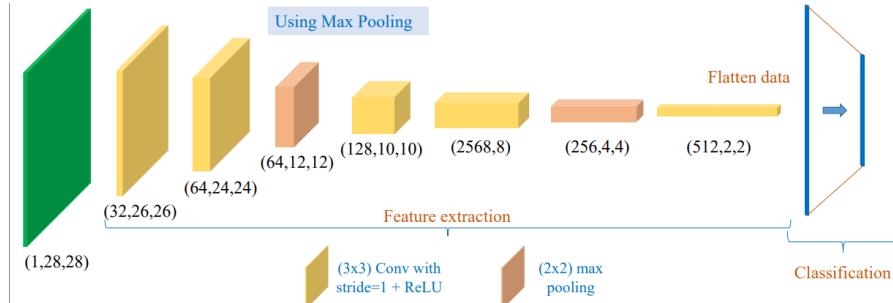
**So sánh với convolution có stride > 1** Một câu hỏi tự nhiên đặt ra: *Tại sao không dùng luôn tích chập với bước nhảy lớn thay cho pooling?* Câu trả lời nằm ở bản chất của hai phép toán:

- Convolution hoạt động trên toàn bộ tập kênh đầu vào và có tham số học được (filter), cho phép biểu diễn linh hoạt hơn nhưng tốn chi phí huấn luyện.
- Pooling hoạt động theo từng mặt phẳng kênh riêng biệt, không có tham số, nhanh hơn và không làm thay đổi số lượng kênh.

Do đó, tùy vào mục tiêu (giảm thông tin hay học thêm đặc trưng), người ta có thể lựa chọn giữa pooling và convolution.

**So sánh với các phương pháp nội suy** Ngoài pooling, một số kỹ thuật giảm kích thước ảnh như nội suy (interpolation), đặc biệt là nội suy song tuyến tính (bilinear interpolation), cũng có thể sử dụng. Tuy nhiên, các kỹ thuật này thường được dùng trong tác vụ tăng kích thước (super-resolution). Trong bài toán giảm ảnh như hiện tại, pooling và convolution tỏ ra hiệu quả và phù hợp hơn.

**Nâng cấp mô hình Fashion-MNIST** Từ mô hình ban đầu sử dụng CNN cơ bản, chúng ta tiến hành nâng cấp bằng cách thêm vào các tầng pooling để tăng hiệu quả trích xuất đặc trưng. Cấu trúc mô hình nâng cấp được thể hiện trong Hình 112.



Hình 101: Mô hình nâng cấp với các tầng pooling

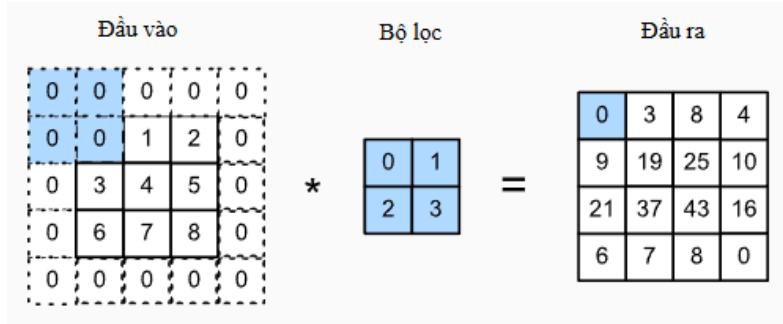
### 7.6.2 Padding

Như đã trình bày trong các mục trước, tầng tích chập (convolutional layer) được sử dụng để tính toán các giá trị đầu ra  $z$  bằng cách khai thác cấu trúc không gian của dữ liệu đầu vào. Sau đó, các kỹ thuật như pooling được áp dụng để giảm chiều dữ liệu, giúp tăng tốc độ hội tụ của mô hình.

Tuy nhiên, nếu xét theo động cơ xuất phát từ mạng MLP (Multilayer Perceptron), ta biết rằng việc tăng số lượng tầng ẩn có thể giúp mạng học được các đặc trưng phức tạp và trừu tượng hơn. Trên thực tế, như đã được kiểm chứng trong thực nghiệm với tập dữ liệu CIFAR-10, mô hình hiện tại chỉ đạt khoảng 70% độ chính xác. Do đó, có lý do chính đáng để tin rằng hiệu suất có thể được cải thiện nếu tăng thêm số tầng trong mạng.

Tuy nhiên, một vấn đề phát sinh là: với mỗi tầng tích chập, kích thước của đầu ra thường bị giảm xuống (trừ khi bước sải stride bằng 1 và có padding phù hợp). Nếu không kiểm soát được sự suy giảm này, mô hình sẽ nhanh chóng bị giới hạn về độ sâu khả thi. Do đó, người ta đề xuất sử dụng kỹ thuật *padding* — tức là chèn thêm các giá trị xung quanh dữ liệu đầu vào nhằm duy trì kích thước không gian.

**Động cơ kỹ thuật** Khi áp dụng tích chập với các kernel nhỏ (ví dụ  $3 \times 3$ ), nếu không sử dụng padding, các điểm ảnh ở rìa sẽ không được xử lý đầy đủ, dẫn đến mất mát thông tin. Mặc dù sự mất mát là nhỏ trong một tầng, nhưng khi áp dụng liên tiếp nhiều tầng, hiệu ứng tích luỹ sẽ khiến thông tin ở biên bị loại bỏ hoàn toàn. Để khắc phục, người ta chèn thêm các giá trị (thường là 0) xung quanh ảnh đầu vào để mở rộng không gian xử lý.



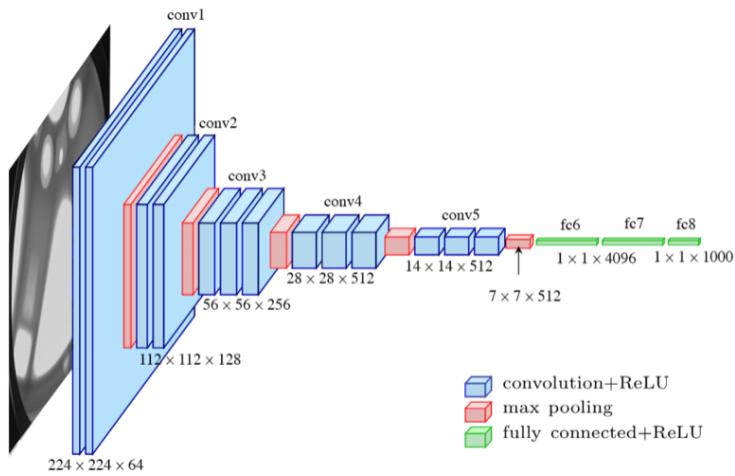
Hình 102: Minh họa ý tưởng padding nhằm duy trì kích thước dữ liệu

**Công thức tính toán** Giả sử đầu vào có kích thước không gian là  $S_D$ , kernel có kích thước  $K \times K$ , bước sải stride là  $S$ , và padding được chèn thêm là  $P$ , khi đó kích thước đầu ra  $S_o$  được tính theo công thức:

$$S_o = \left\lfloor \frac{S_D - K + 2P}{S} \right\rfloor + 1$$

Công thức này cho phép tính toán chính xác kích thước không gian sau mỗi tầng tích chập, giúp thiết kế mạng dễ dàng hơn.

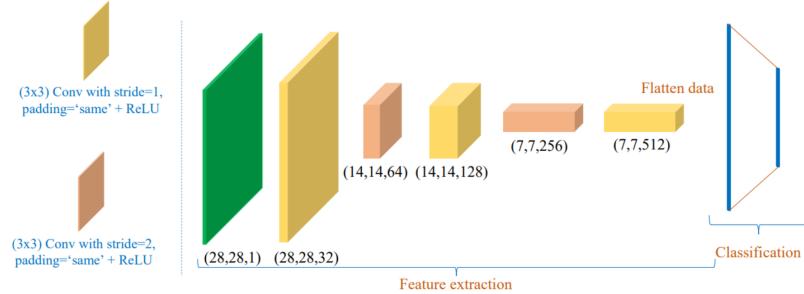
**Ứng dụng trong thực tế** Một ví dụ điển hình cho việc sử dụng padding là mô hình VGG16 — một trong những kiến trúc CNN phổ biến. Trong mô hình này, padding được áp dụng ở tất cả các tầng convolution nhằm giữ nguyên kích thước không gian của feature map, cho phép xây dựng mạng với độ sâu lớn mà không làm giảm quá nhanh kích thước đầu ra. Chỉ khi thông tin đã được trích xuất đủ, mô hình mới áp dụng pooling để giảm chiều không gian.



Hình 103: Kiến trúc mô hình VGG16 sử dụng padding trong các tầng tích chập

**Ghi chú** Khi xử lý dữ liệu bằng padding, các giá trị được thêm vào (thường là 0) không mang thông tin thực, nhưng chúng đóng vai trò quan trọng trong việc đảm bảo các đặc trưng ở biên vẫn được giữ lại sau nhiều tầng tích chập. Điều này giúp duy trì tính toàn vẹn không gian của dữ liệu và tăng hiệu quả học của mô hình.

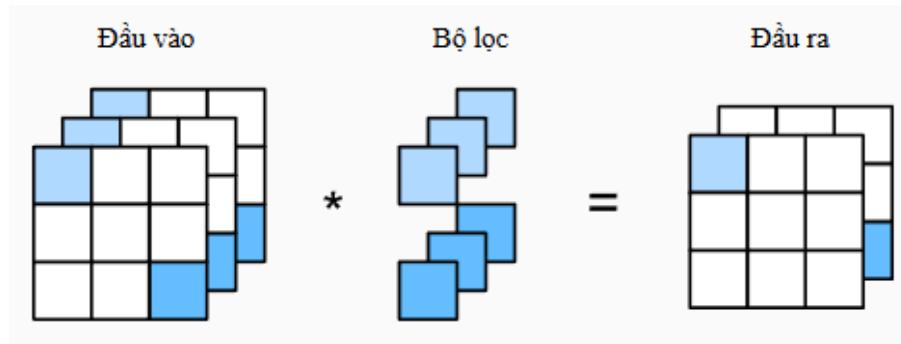
**Nâng cấp mô hình Fashion-MNIST** Từ mô hình ban đầu sử dụng CNN cơ bản, chúng ta tiến hành nâng cấp bằng cách thêm vào padding vào các tầng tích chập, khi đó chúng ta sẽ có model là:



Hình 104: Enter Caption

### 7.6.3 1x1 Convolution

Phép tích chập với kích thước nhân  $1 \times 1$  thoạt nhìn có vẻ không mang nhiều ý nghĩa, bởi lẽ mục đích của tích chập truyền thống là nhằm khai thác mối tương quan không gian giữa các điểm ảnh lân cận. Tuy nhiên,  $1 \times 1$  convolution lại là một công cụ hữu hiệu trong việc thiết kế các mô hình mạng nơ-ron sâu, đặc biệt là trong việc xử lý chiều kenh (channel dimension).

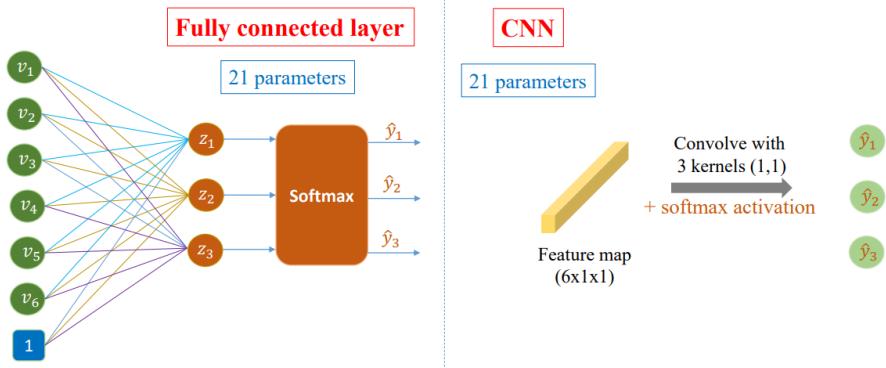


Hình 105: Minh họa phép tích chập  $1 \times 1$

Do kích thước cửa sổ lọc chỉ là  $1 \times 1$ , nên phép tích chập này không khai thác tương tác không gian (chiều cao và chiều rộng) giữa các điểm ảnh, mà chỉ thực hiện trên chiều sâu (channel). Cụ thể, tại mỗi vị trí không gian  $(i, j)$ , một phép tích chập  $1 \times 1$  thực hiện tổ hợp tuyến tính giữa tất cả các giá trị tại vị trí đó trên các kênh đầu vào. Điều này tương tự như một lớp perceptron áp dụng riêng biệt cho từng vị trí không gian trên tensor đầu vào.

Ví dụ, giả sử tensor đầu vào có kích thước  $(C_{in}, H, W)$ , khi áp dụng  $n$  bộ lọc  $1 \times 1$ , mỗi bộ lọc có kích thước  $(C_{in}, 1, 1)$ , ta thu được đầu ra có kích thước  $(n, H, W)$ . Do đó, phép tích chập  $1 \times 1$  còn được sử dụng như một lớp *fully connected* (FC) nhưng áp dụng song song tại tất cả các vị trí không gian.

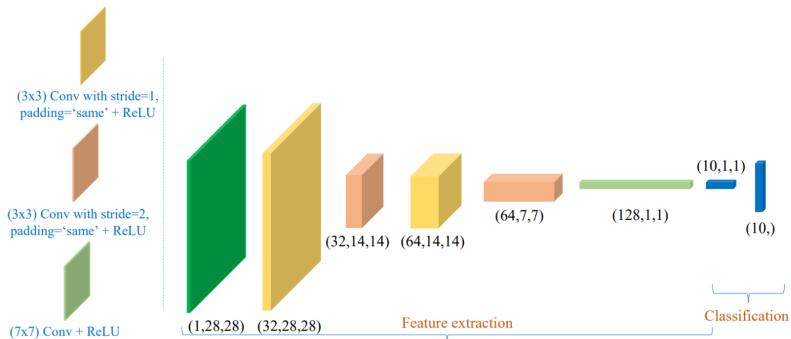
Ý tưởng này trở nên đặc biệt hữu ích trong giai đoạn cuối của mô hình CNN. Khi số lượng kênh đã được tăng lên đáng kể nhờ quá trình trích xuất đặc trưng, ta có thể sử dụng phép tích chập  $1 \times 1$  để thực hiện nhiệm vụ phân loại. Điều này cho phép xây dựng các mô hình thuần CNN, loại bỏ hoàn toàn tầng FC truyền thống.



Hình 106: Minh họa việc sử dụng  $1 \times 1$  convolution để sinh ra các logit

Trong hình trên, giả sử ở tầng cuối cùng, chúng ta có một tensor đặc trưng  $(6, 1, 1)$ , đại diện cho 6 đặc trưng trùm tương. Khi đó, để thu được đầu ra gồm 3 logit, ta có thể sử dụng 3 bộ lọc  $1 \times 1$  với số kênh đầu vào là 6. Kết quả là đầu ra có kích thước  $(3, 1, 1)$ , sau đó có thể reshape thành vector  $(3,)$  để đưa vào hàm softmax. Tổng quát, đầu ra từ mô hình có thể được reshape từ  $(N, C, 1, 1)$  thành  $(N, C)$  để phục vụ cho phân loại.

Áp dụng kỹ thuật này vào mô hình nhận dạng ảnh Fashion-MNIST, ta có thể thiết kế một mô hình thuần CNN như sau:



Hình 107: Mô hình CNN hoàn chỉnh cho tập dữ liệu Fashion-MNIST

#### 7.6.4 Implementation from Scratch

Dựa trên các khái niệm đã trình bày, chúng tôi xây dựng mô hình mạng nơ-ron sâu từ đầu và thu được kết quả như sau.

Mặc dù độ chính xác (accuracy) đã được cải thiện so với mô hình MLP cơ bản, tuy nhiên, độ chính xác trên tập huấn luyện (training accuracy) chỉ hội tụ xấp xỉ 55%, thay vì tiến gần 100% như các khái niệm trước đó. Điều này phản ánh hạn chế về khả năng học biểu diễn của mô hình. Để kiểm chứng, chúng tôi tiến hành mở rộng độ sâu bằng cách tăng số lượng tầng ẩn.

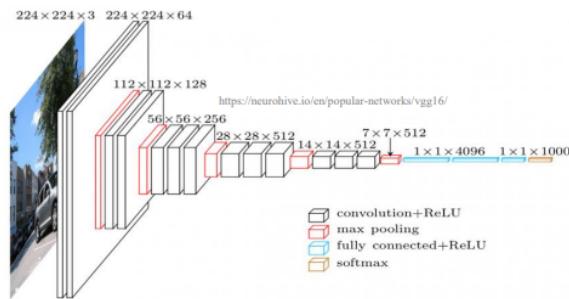
Sau khi tăng độ sâu, kết quả thu được đã có dấu hiệu cải thiện rõ rệt, cho thấy tiềm năng học biểu diễn được mở rộng. Tiếp theo, chúng tôi áp dụng kiến trúc khối (block) theo phong cách VGG để tiếp tục nâng cao hiệu suất mô hình. Kiến trúc này sẽ được trình bày và phân tích chi tiết dưới đây.

**Khái niệm về khối (block):** Khối cơ bản trong mạng nơ-ron tích chập cỗ điển bao gồm chuỗi các tầng sau:

1. Tầng tích chập (convolutional layer), thường sử dụng phần đệm (padding) để giữ nguyên độ phân giải không gian.
2. Hàm kích hoạt phi tuyến, phổ biến nhất là ReLU.
3. Tầng gộp (pooling), thường là gộp cực đại (max pooling), nhằm giảm chiều không gian và tăng tính bất biến không gian.

Một khối VGG điển hình bao gồm một hoặc nhiều tầng tích chập liên tiếp, sau đó là một tầng gộp cực đại. Việc tổ chức này giúp mô hình học được các đặc trưng cục bộ hiệu quả hơn và giảm số lượng tham số.

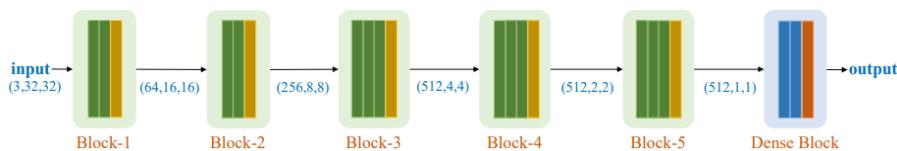
**Ý nghĩa thực tiễn:** Khái niệm khối giúp trừu tượng hóa kiến trúc mạng, cho phép thiết kế mạng sâu một cách có hệ thống và dễ kiểm soát độ phức tạp tính toán. Mỗi khối có thể coi như một "bộ lọc đặc trưng" có chức năng nhận diện một mức trừu tượng cụ thể của dữ liệu (ví dụ: cạnh, hình dạng, đối tượng).



Hình 108: Mô hình VGG-16

Mạng VGG-16 bao gồm 5 khối tích chập. Hai khối đầu tiên mỗi khối chỉ chứa một tầng tích chập, trong khi ba khối còn lại chứa hai tầng tích chập. Khối đầu tiên có 64 kênh đầu ra, và số kênh được nhân đôi sau mỗi khối tiếp theo, đạt tối đa 512 ở các tầng sâu nhất. Với 13 tầng tích chập và 3 tầng kết nối đầy đủ (fully connected), tổng cộng 16 tầng học tham số, mô hình này được gọi là VGG-16.

**Áp dụng vào dữ liệu CIFAR-10:** Dựa trên kiến trúc của VGG, chúng tôi xây dựng một phiên bản thu gọn phù hợp với bộ dữ liệu CIFAR-10 như trong Hình 109.



Hình 109: Kiến trúc VGG16-like áp dụng cho CIFAR-10

Vì kích thước ảnh trong CIFAR-10 là  $32 \times 32$ , sau 5 lần áp dụng các tầng gộp (mỗi lần giảm còn một nửa) thì kích thước vẫn vừa vặn với đầu vào của tầng kết nối đầy đủ. Tuy nhiên, nếu áp dụng kiến trúc này cho MNIST (kích thước ảnh  $28 \times 28$ ), ta sẽ gặp vấn đề khi đến tầng thứ 5:

$$28 \rightarrow 14 \rightarrow 7 \rightarrow 3 \rightarrow 1 \rightarrow 0$$

Rõ ràng, ảnh sẽ bị triệt tiêu về kích thước trước khi đến tầng cuối cùng. Do đó, để áp dụng kiến trúc VGG16-like một cách hợp lý, kích thước ảnh đầu vào tối thiểu nên là  $32 \times 32$ .

**Kết quả huấn luyện:** Khi triển khai mô hình theo kiến trúc trên, kết quả thu được cho thấy mạng đã đạt đến mức bão hòa trong khả năng học, thể hiện qua việc độ chính xác không còn tăng đáng kể. Điều này đặt ra câu hỏi về giới hạn hiện tại của mô hình và hướng đến việc sử dụng thêm các kỹ thuật tối ưu hóa hiện đại hơn.

Source code: [CNN from scratch](#)

**Hướng tối ưu hóa tiếp theo:** Trong các phần trước, chúng ta đã có dịp tiếp xúc với một số kỹ thuật tối ưu như thuật toán Adam hay, He Initialize nhưng ở concept trước thì chúng ta đã không chú trọng vào normalize ban đầu vì chúng ta chỉ mong muốn mean = 0 để dễ dàng He thôi, vậy nếu đã có  $[-1,1]$  rồi thì tại sao người ta vẫn phải phát triển thêm Guassian làm gì

## 8 Pytorch

---

### 8.1 Autograd

Trong các bài toán học máy, đặc biệt là học sâu (deep learning), việc tính đạo hàm của hàm mất mát đối với các tham số của mô hình là thiết yếu để tối ưu hóa qua thuật toán lan truyền ngược (backpropagation). Cơ chế **autograd** của PyTorch cho phép tự động hóa quá trình tính đạo hàm này bằng cách xây dựng và duy trì một đồ thị tính toán (computation graph) động trong quá trình thực thi chương trình. Để hiểu rõ cơ chế này, người học cần nắm vững quy tắc chuỗi trong vi phân, bởi toàn bộ quá trình đạo hàm ngược đều dựa trên quy tắc này.

Trong PyTorch, mọi đại lượng liên quan đến mô hình học máy như dữ liệu đầu vào, tham số mô hình, và đầu ra đều được biểu diễn dưới dạng **tensor**. Để PyTorch có thể tính toán đạo hàm của một tensor, ta cần gán thuộc tính `requires_grad=True` cho tensor đó. Tuy nhiên, điều kiện tiên quyết là tensor phải có kiểu dữ liệu số thực (`float`). Mặc định, các tensor được khởi tạo sẽ có `requires_grad=False`. Khi một tensor có `requires_grad=True`, mọi phép toán được thực hiện từ tensor này đều tạo ra các tensor mới có cùng thuộc tính `requires_grad=True`. Đặc biệt, nếu một phép toán liên quan đến một tensor có `requires_grad=True` và một tensor không có thuộc tính này, thì kết quả vẫn là một tensor với `requires_grad=True`.

Khi thực hiện các phép toán giữa các tensor, PyTorch sẽ xây dựng một đồ thị tính toán biểu diễn các mối quan hệ toán học giữa các đại lượng. Trong đồ thị này, các node là các phép toán (như cộng, nhân, mũ), còn các cạnh là các tensor truyền qua các phép toán đó. Các tensor được tạo ra trực tiếp từ người dùng, không phải từ phép toán, và có `requires_grad=True`, được gọi là **leaf tensor**. Trong một mô hình học sâu, các tham số như trọng số và độ lệch (weights và bias) thường chính là các leaf tensor này.

Để tính đạo hàm của hàm mất mát  $L$  đối với các tham số mô hình, ta gọi phương thức `backward()` lên  $L$ . Khi đó, PyTorch sẽ tính toán đạo hàm của  $L$  đối với từng leaf tensor có `requires_grad=True` và lưu kết quả vào thuộc tính `.grad` của các tensor đó. Việc tính toán được thực hiện bằng cách áp dụng quy tắc chuỗi ngược qua từng phép toán trong đồ thị tính toán. Ví dụ, với mô hình tuyến tính đơn giản  $y = ax + b$ , hàm mất mát  $L = (ax + b - y)^2$  sẽ có đạo hàm theo  $a$  và  $b$  lần lượt là  $\frac{\partial L}{\partial a} = 2(ax + b - y) \cdot x$  và  $\frac{\partial L}{\partial b} = 2(ax + b - y)$ .

Mỗi tensor không phải leaf tensor trong đồ thị, nếu có `requires_grad=True`, sẽ mang một thuộc tính `grad_fn` lưu thông tin về phép toán đã tạo ra tensor đó. Thuộc tính này tham chiếu tới các lớp kế thừa từ `torch.autograd.Function`. Mỗi lớp như vậy định nghĩa hai phương thức: `forward` để thực hiện phép toán và lưu trữ thông tin cần thiết cho quá trình lan truyền ngược, và `backward` để tính đạo hàm dựa trên gradient đầu ra. Điều này cho phép người dùng có thể tự định nghĩa các phép toán mới bằng cách viết các lớp kế thừa Function, từ đó mở rộng khả năng tính toán của PyTorch.

Trong trường hợp đầu ra của mô hình không phải là một số thực mà là một vector, thì đạo hàm được tính sẽ là một ma trận Jacobian. Cụ thể, nếu đầu vào là một vector  $\mathbf{x} \in \mathbb{R}^n$  và đầu ra là một vector  $\mathbf{y} \in \mathbb{R}^m$ , thì ma trận Jacobian là  $J_{ij} = \frac{\partial y_i}{\partial x_j}$ . Trong thực tế, PyTorch cho phép người dùng truyền một vector gradient đầu ra vào hàm `backward()`

thông qua đối số `gradient`, từ đó cho phép tính đạo hàm trong trường hợp tổng quát.

Một điểm quan trọng của autograd trong PyTorch là nó sử dụng **đồ thị tính toán động** (dynamic computation graph). Nghĩa là, đồ thị chỉ được xây dựng tại thời điểm chương trình chạy qua các phép toán liên quan đến tensor có `requires_grad=True`. Sau khi thực hiện lan truyền ngược (gọi `backward`), toàn bộ đồ thị và các biến tạm trong quá trình tính toán sẽ được giải phóng khỏi bộ nhớ để tiết kiệm tài nguyên. Do đó, nếu người dùng cần thực hiện lan truyền ngược nhiều lần trên cùng một đồ thị, cần chỉ định `retain_graph=True` khi gọi `backward()`. Đồng thời, do các giá trị gradient sẽ được cộng dồn, cần gọi `.zero_grad()` để xóa gradient trước mỗi lần lan truyền ngược mới trong vòng lặp huấn luyện.

Trong quá trình inference, khi không cần tính đạo hàm, người dùng nên sử dụng context manager `torch.no_grad()` để tắt cơ chế autograd, từ đó giảm thiểu chi phí bộ nhớ và tính toán. Ngoài ra, thuộc tính `requires_grad` cũng có thể được gán bằng `False` trong trường hợp người dùng muốn “đóng băng” (freeze) một số tầng trong mô hình nhằm không cập nhật các tham số đó trong quá trình huấn luyện.

Cuối cùng, một điểm khác biệt giữa PyTorch và TensorFlow 1.x là cơ chế xây dựng đồ thị: PyTorch sử dụng đồ thị động (define-by-run), trong khi TensorFlow 1.x sử dụng đồ thị tĩnh (define-and-run). Điều này giúp việc debug trong PyTorch trở nên trực quan và linh hoạt hơn. TensorFlow 2.x đã thay đổi cách tiếp cận và cũng hỗ trợ đồ thị động tương tự PyTorch.

## 9 Advance CNN optimize

### 9.1 Noise addiction

Kỹ thuật thêm nhiễu (noise injection) bao gồm việc đưa các thành phần ngẫu nhiên, chẳng hạn như nhiễu Gaussian, vào dữ liệu đầu vào, trọng số, hoặc các tầng kích hoạt trong quá trình huấn luyện mạng nơ-ron. Mục tiêu của kỹ thuật này là tăng tính khái quát (generalization) và giảm hiện tượng quá khớp (overfitting) bằng cách làm cho mô hình trở nên bền vững hơn trước các biến thiên nhỏ của dữ liệu. Về mặt lý thuyết, khi thêm nhiễu  $\epsilon$  vào đầu vào  $x$  (ví dụ  $\tilde{x} = x + \epsilon$ , với  $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$ ), kỳ vọng của hàm mất mát trên phân phối nhiễu có thể xấp xỉ bởi:

$$\mathbb{E}_\epsilon[\mathcal{L}(f(\tilde{x}), y)] \approx \mathcal{L}(f(x), y) + \frac{\sigma^2}{2} \|\nabla_x f(x)\|^2 + o(\sigma^2).$$

Điều này tương đương với việc thêm một hạng tử điều chuẩn (regularization) phụ thuộc vào đạo hàm của mô hình, nhằm làm giảm độ nhạy của hàm mục tiêu đối với nhiễu. Phân tích này cho thấy rằng thêm nhiễu vào đầu vào có tác dụng tương đương với điều chuẩn Tikhonov (L2 regularization). Ngoài ra, việc thêm nhiễu cũng có thể áp dụng tại các tầng ẩn hoặc trên trọng số mô hình để tăng tính ngẫu nhiên và cải thiện độ ổn định trong quá trình huấn luyện.

### 9.2 Z1 Score \_ Guassian Distribution

Một trong những kỹ thuật quan trọng giúp tăng hiệu suất và độ ổn định trong huấn luyện mô hình học sâu là chuẩn hóa dữ liệu đầu vào. Trong phần trước, chúng ta đã sử dụng chuẩn hóa tuyến tính về khoảng  $[-1, 1]$ . Tuy nhiên, một lựa chọn mạnh mẽ và phổ biến hơn là chuẩn hóa theo phân phối Gaussian, còn gọi là Z-score normalization.

Chuẩn hóa Z-score đưa dữ liệu về phân phối chuẩn với kỳ vọng gần 0 và độ lệch chuẩn gần 1, theo công thức:

$$z = \frac{x - \mu}{\sigma}$$

trong đó  $x$  là một điểm dữ liệu,  $\mu$  là giá trị trung bình của tập dữ liệu, và  $\sigma$  là độ lệch chuẩn.

**Ưu điểm so với chuẩn hóa  $[-1, 1]$ :** Dù cả hai phương pháp đều đưa dữ liệu về trung tâm (zero-mean), chuẩn hóa Z-score thể hiện các đặc điểm vượt trội như sau:

- **Tính bất biến tuyến tính (Linear Invariance):** Z-score không bị ảnh hưởng bởi các phép biến đổi tuyến tính như thay đổi độ sáng hoặc độ tương phản trong ảnh. Điều này đặc biệt quan trọng trong bài toán thị giác máy tính, khi đối tượng có thể xuất hiện dưới nhiều điều kiện ánh sáng khác nhau.

$$Y = aX + b$$

$$\begin{aligned}\bar{Y} &= \frac{Y - \mu_Y}{\sigma_Y} = \frac{(aX + b) - \frac{1}{n} \sum_i (aX_i + b)}{\sqrt{\frac{1}{n} \sum_i ((aX_i + b) - \frac{1}{n} \sum_i (aX_i + b))^2}} \\ &= \frac{aX - \frac{1}{n} \sum_i aX_i}{\sqrt{\frac{1}{n} \sum_i (aX_i - \frac{1}{n} \sum_j aX_j)^2}} \\ &= \frac{X - \frac{1}{n} \sum_i X_i}{\sqrt{\frac{1}{n} \sum_i (X_i - \frac{1}{n} \sum_j X_j)^2}} = \frac{X - \mu_X}{\sqrt{\frac{1}{n} \sum_i (X_i - \mu_X)^2}} = \bar{X}\end{aligned}$$

*Ví dụ:* Một hình ảnh của chú chó có thể bị chụp dưới ánh sáng mạnh hoặc yếu hơn, nhưng chuẩn hóa Z-score giúp mạng nơ-ron nhận biết đó vẫn là cùng một đối tượng, nhờ đó không cần học lại từ đầu cho mỗi phiên bản ánh sáng khác nhau.

- **Tăng khả năng tổng quát hóa (Generalization):** Khi loại bỏ các biến thể tuyến tính không cần thiết, mô hình có thể tập trung học các đặc trưng có tính phân biệt cao hơn, từ đó tổng quát tốt hơn trên dữ liệu chưa từng thấy.

**Vấn đề chuẩn hóa theo từng kênh màu (Channel-wise Normalization):** Trong ảnh màu RGB, mỗi kênh (Red, Green, Blue) thường có phân bố giá trị khác nhau do bản chất vật lý và đặc trưng thị giác:

- Kênh đỏ (R) thường có độ sáng cao hơn.
- Kênh xanh lá (G) thường chiếm ưu thế trong ảnh tự nhiên.
- Kênh xanh dương (B) thường có giá trị thấp hơn và nhạy cảm với nhiễu.

Do đó, nếu chuẩn hóa toàn bộ ảnh bằng một giá trị trung bình và độ lệch chuẩn chung (global normalization), ta sẽ vô tình làm mất đi sự khác biệt thống kê giữa các kênh màu, từ đó ảnh hưởng đến khả năng phân biệt màu sắc của mô hình. Ngược lại, **chuẩn hóa riêng biệt cho từng kênh màu** (channel-wise normalization) giúp bảo toàn cấu trúc thống kê tự nhiên, làm cho việc học sâu hiệu quả và ổn định hơn.

**Thử nghiệm thực tế:** Sau khi áp dụng chuẩn hóa Z-score theo từng kênh màu vào mô hình VGG16-like (Hình 112), chúng tôi thu được kết quả như sau:

- **Độ chính xác kiểm thử (Test Accuracy):** đạt khoảng **80%**, cao hơn đáng kể so với khi sử dụng chuẩn hóa tuyến tính đơn giản.

**Kết luận:** Chuẩn hóa theo phân phối Gaussian không chỉ giúp mô hình hội tụ nhanh hơn mà còn nâng cao hiệu suất tổng thể nhờ tăng cường tính ổn định và khả năng học của mạng. Việc kết hợp chuẩn hóa Z-score với chuẩn hóa từng kênh màu càng làm nổi bật vai trò của tiền xử lý dữ liệu trong học sâu. Đây cũng là lý do vì sao các kiến trúc hiện đại thường áp dụng chuẩn hóa kiểu này, cùng với các kỹ thuật như Batch Normalization, để tối ưu hóa quá trình huấn luyện.

### 9.3 Batch Normalization

Một trong những công cụ hiệu quả giúp tăng độ chính xác là chuẩn hóa dữ liệu đầu vào. Tuy nhiên, nếu chỉ áp dụng chuẩn hóa tại đầu vào, thì qua nhiều lớp tuyến tính trong mô hình học sâu, phân phối dữ liệu vẫn sẽ bị biến đổi theo những ma trận tham số khác nhau, các giá trị kích hoạt ở các tầng trung gian có thể nhận các giá trị với mức độ biến thiên lớn- dọc theo các tầng từ đầu vào đến đầu ra, qua các nút ở cùng một tầng, và theo thời gian do việc cập nhật giá trị tham số. Những nhà phát minh kỹ thuật chuẩn hóa theo batch cho rằng sự thay đổi trong phân phối của những giá trị kích hoạt có thể cản trở sự hội tụ của mạng. Để thấy rằng nếu một tầng có các giá trị kích hoạt lớn gấp 100 lần so với các tầng khác, thì cần phải có các điều chỉnh bổ trợ trong tốc độ học.

**Vấn đề của việc chuẩn hóa đầu vào** Giả sử dữ liệu đầu vào được chuẩn hóa theo Z-score, thì sau khi đi qua một lớp tuyến tính  $z = Wx + b$ , phân phối của  $z$  lại có thể bị lệch đi đáng kể do tham số trọng số  $W$  và  $b$ . Khi điều này lặp lại qua nhiều lớp, phân phối của đầu ra bị thay đổi liên tục, khiến các lớp sau phải liên tục thích nghi với phân phối mới. Đây chính là nguyên nhân chính gây khó khăn cho việc huấn luyện mạng sâu.

**Ý tưởng chính của BatchNorm** Thay vì chỉ chuẩn hóa dữ liệu đầu vào, BatchNorm tiến hành chuẩn hóa đầu ra của từng lớp trung gian, cụ thể là:

- Áp dụng chuẩn hóa cho từng mini-batch thay vì toàn bộ dữ liệu.
- Tính toán giá trị trung bình và phương sai theo công thức:

$$\mu_B = \frac{1}{m} \sum_{i=1}^m z_i, \quad \sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (z_i - \mu_B)^2$$

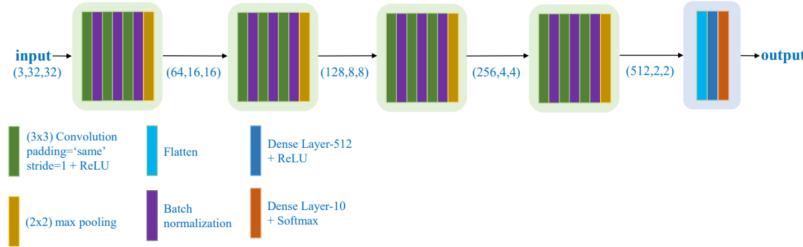
- Chuẩn hóa từng  $z_i$  trong batch:

$$\hat{z}_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

- Áp dụng biến đổi tuyến tính có thể học được:

$$y_i = \gamma \hat{z}_i + \beta$$

với  $\gamma$  và  $\beta$  là hai tham số được học cùng với các trọng số của mạng.



Hình 110: Enter Caption

**Ý nghĩa của tham số  $\gamma$  và  $\beta$ :** Hai tham số này được khởi tạo ban đầu lần lượt là  $\gamma = 1$  và  $\beta = 0$ , nhằm giữ nguyên phân phối chuẩn hóa ban đầu. Tuy nhiên, trong quá trình huấn luyện, mạng có thể điều chỉnh các giá trị này để undo hiệu ứng chuẩn hóa nếu điều đó có lợi cho việc học ở lớp hiện tại.

Điều này rất quan trọng vì trong một số trường hợp, việc duy trì phân phối đầu ra khác với phân phối chuẩn ( $\mu = 0, \sigma = 1$ ) có thể mang lại hiệu quả tối ưu hơn cho một tầng nhất định trong mạng nơ-ron.

**Ảnh hưởng của kích thước batch** BatchNorm phụ thuộc vào việc ước lượng mean và variance từ từng mini-batch. Khi kích thước batch nhỏ, việc ước lượng có thể kém ổn định. Trong các trường hợp như vậy, các biến thể như **LayerNorm**, **InstanceNorm** hoặc **GroupNorm** có thể được áp dụng.

**Tổng kết** Batch Normalization là một bước trung gian hiệu quả giúp ổn định phân phối dữ liệu qua các lớp trong mạng, giảm hiện tượng Internal Covariate Shift, tăng tốc độ hội tụ, và cải thiện hiệu suất của mô hình.

Ta thử áp dụng vào thuật toán của mình xem như thế nào?

#### 9.4 Dropout

Dropout là một kỹ thuật điều chỉnh chuẩn được áp dụng phổ biến trong các mạng nơ-ron sâu nhằm ngăn ngừa hiện tượng quá khớp. Tại mỗi bước huấn luyện, một tập con các đơn vị (neuron) trong mạng sẽ bị loại bỏ ngẫu nhiên với xác suất  $p$ , tức là các đầu ra tương ứng được đặt bằng 0. Cụ thể, đầu ra  $\mathbf{h}$  của một tầng được nhân với một mặt nạ nhiễu Bernoulli  $\mathbf{m} \sim \text{Bernoulli}(1 - p)$  để tạo thành đầu ra mới  $\tilde{\mathbf{h}} = \mathbf{m} \odot \mathbf{h}$ . Cách tiếp cận này khiến cho mạng không thể phụ thuộc vào bất kỳ một neuron cụ thể nào, từ đó buộc mô hình học được các đặc trưng có tính khái quát cao hơn.

Về bản chất, dropout có thể được hiểu như một kỹ thuật xấp xỉ hội trung mô hình (model averaging). Trong quá trình huấn luyện, mỗi bước tương ứng với việc huấn luyện một mạng con (subnetwork) khác nhau, được hình thành bằng cách loại bỏ ngẫu nhiên các neuron. Khi kiểm tra, toàn bộ mạng được sử dụng nhưng các trọng số được thu nhỏ lại (scaled) để xấp xỉ trung bình của các mạng con đã được huấn luyện. Điều này tương đương với việc lấy kỳ vọng đầu ra theo phân phối dropout. Vì lý do đó, dropout không chỉ làm giảm overfitting mà còn cải thiện độ bền mô hình thông qua việc lan truyền tính ngẫu nhiên trong mạng lưới.

## 9.5 Weight Decay và Kernel Regularization

Weight decay, hay còn gọi là điều chuẩn L2, là một kỹ thuật cỗ điển nhằm kiểm soát độ phức tạp của mô hình bằng cách thêm vào hàm mất mát một hạng tử phạt tỉ lệ với bình phương chuẩn  $L^2$  của vector trọng số:

$$J(w) = \mathcal{L}(w) + \frac{\lambda}{2} \|w\|_2^2.$$

Hệ số  $\lambda > 0$  kiểm soát mức độ điều chuẩn. Trong quá trình tối ưu, gradient của hàm mất mát sẽ được điều chỉnh bằng cách cộng thêm  $\lambda w$ , dẫn đến việc trọng số bị kéo gần về 0 sau mỗi bước cập nhật. Điều này làm giảm nguy cơ mô hình học thuộc nhiều trong dữ liệu huấn luyện và cải thiện độ khái quát.

Mở rộng hơn, weight decay có thể được xem như một trường hợp đặc biệt của điều chuẩn trong không gian hàm học máy (function space regularization), cụ thể là điều chuẩn hạt nhân (kernel regularization) trong không gian Hilbert tái sinh (RKHS). Với một kernel  $K$ , ta có thể định nghĩa chuẩn trong không gian hàm tương ứng:

$$\min_{f \in \mathcal{H}} \sum_{i=1}^n \mathcal{L}(f(x_i), y_i) + \lambda \|f\|_{\mathcal{H}}^2,$$

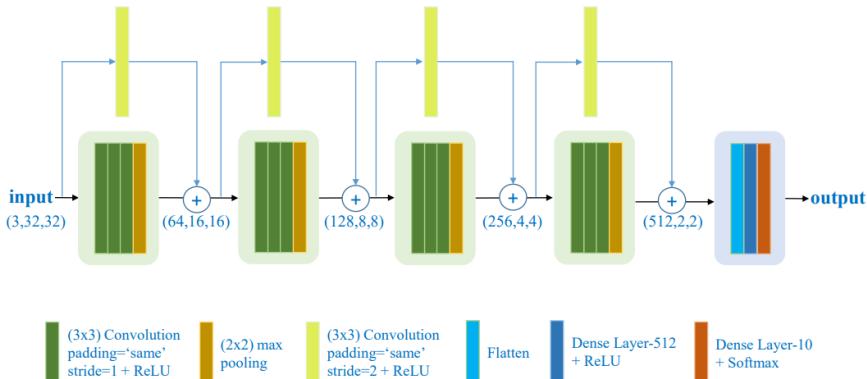
trong đó  $\|f\|_{\mathcal{H}}$  đo độ trơn (smoothness) của hàm  $f$  trong không gian RKHS xác định bởi kernel  $K$ . Trong trường hợp kernel tuyến tính  $K(x, x') = x^\top x'$ , điều chuẩn hạt nhân trùng với điều chuẩn L2 truyền thống. Từ góc độ lý thuyết học thống kê, điều chuẩn có vai trò giới hạn không gian hàm tìm kiếm, từ đó kiểm soát sự đánh đổi giữa sai số huấn luyện và độ khái quát (bias-variance trade-off). Điều này phù hợp với nguyên lý tối thiểu hóa rủi ro cấu trúc (Structural Risk Minimization), nền tảng của nhiều lý thuyết học máy hiện đại.

## 9.6 Advanced Activation Function: Swish

Tiếp theo, chúng ta thử nghiệm với một hàm kích hoạt nâng cao hơn là **Swish**. Về mặt hình dáng khi biểu diễn đồ thị, Swish có vẻ ngoài khá giống với ReLU, nhưng điểm khác biệt quan trọng là **Swish là một hàm lồi (convex)** trên một số khoảng và **trơn (smooth)** cũng như **khả vi tại mọi điểm**. Những đặc tính này giúp quá trình lan truyền ngược (backpropagation) trở nên mượt mà hơn và ổn định hơn trong quá trình tối ưu. Một số ưu điểm nổi bật của Swish bao gồm:

- **Tính trơn và khả vi toàn miền:** Hạn chế hiện tượng gradient bị gián đoạn như ở ReLU, hỗ trợ việc huấn luyện các mô hình sâu.
- **Kích hoạt mềm các giá trị âm nhỏ:** Không “loại bỏ” hoàn toàn thông tin từ các đầu vào âm nhỏ như ReLU, giúp duy trì tín hiệu trong mạng.
- **Tính không đơn điệu (non-monotonicity):** Cung cấp khả năng biểu diễn linh hoạt và phức tạp hơn, cho phép mạng học được các quan hệ phi tuyến tính vi.

Khi áp dụng Swish vào mô hình của chúng tôi, kết quả thu được thể hiện cải thiện đáng kể so với việc sử dụng ReLU, cả về tốc độ hội tụ lẫn độ chính xác cuối cùng. Chi tiết sẽ được trình bày trong phần kết quả bên dưới.



Hình 111: Enter Caption

## 9.7 Skip connection

Skip Connection là một kỹ thuật thiết yếu trong các kiến trúc mạng nơ-ron sâu hiện đại, đặc biệt nổi bật từ kiến trúc ResNet và các mô hình kế thừa. Ý tưởng cốt lõi là cho phép dòng thông tin đi tắt (shortcut) qua một số tầng của mạng, nhằm giảm hiện tượng mất thông tin và giúp gradient lan truyền ngược hiệu quả hơn.

Một cách triển khai đơn giản của skip connection là sử dụng phép cộng. Giả sử đầu vào có kích thước  $(C, H, W) = (64, 32, 32)$ , sau khi đi qua các lớp convolution có kernel size  $3 \times 3$  với padding là "same", ta vẫn giữ được kích thước đầu ra là  $(64, 32, 32)$ . Khi đó, ta có thể cộng trực tiếp các tensor này lại với nhau. Tuy nhiên, phép cộng trong ngữ cảnh này không đơn thuần là toán học đơn giản, mà phải được hiểu như một node trong đồ thị tính toán, nơi các đạo hàm có thể lan truyền ngược một cách đầy đủ trong quá trình học.

Ngoài phép cộng, một phương pháp khác là phép nối (concatenation). Trong trường hợp này, thay vì cộng hai tensor có cùng kích thước, ta ghép chúng lại theo chiều kênh (channel), dẫn đến output có kích thước  $(128, 32, 32)$ . Phép nối giúp giữ lại nhiều thông tin hơn, và thường được dùng trong các kiến trúc phức tạp hơn như DenseNet hoặc UNet.

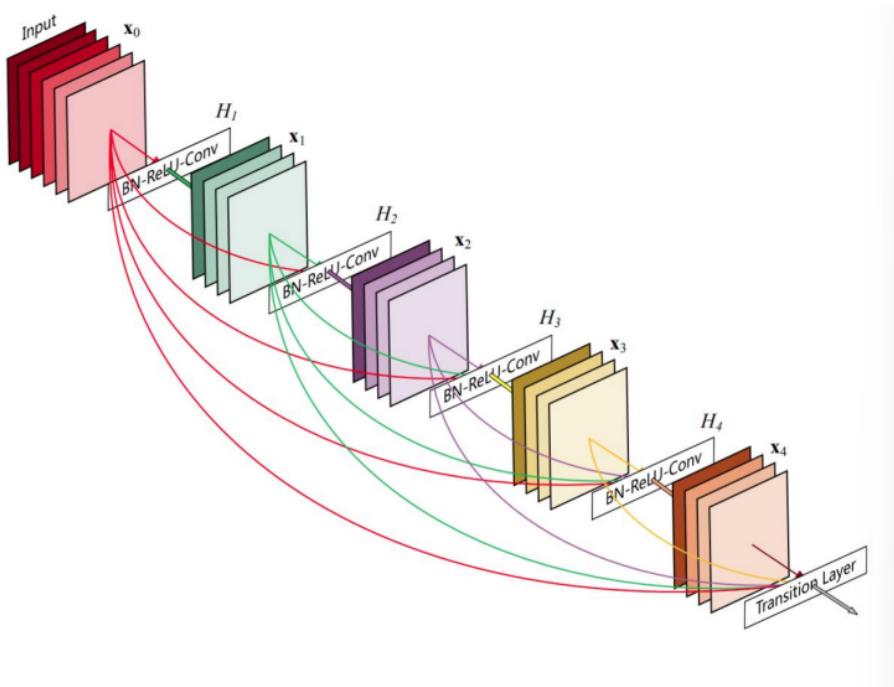
Một thách thức khác là vấn đề sai khác về kích thước. Ví dụ, nếu một nhánh của mạng sử dụng pooling hoặc stride làm giảm kích thước từ  $(32, 32)$  xuống  $(16, 16)$ , thì ta không thể cộng trực tiếp với tensor có kích thước lớn hơn. Trong trường hợp này, cần sử dụng các kỹ thuật như pooling hoặc convolution với stride để đưa hai tensor về cùng kích thước trước khi áp dụng skip connection.

Về mặt lan truyền ngược, PyTorch tự động quản lý biểu đồ đạo hàm (computational graph) để đảm bảo việc cập nhật tham số là chính xác. Khi thực hiện các phép cộng hay nối, hệ thống vẫn giữ thông tin đạo hàm để tính toán hiệu quả trong quá trình backpropagation.

Skip connection là một phần không thể thiếu trong các mô hình hiện đại do khả năng giữ lại thông tin, tăng tốc độ hội tụ, và giảm hiện tượng biến mất gradient. Các mô hình nổi tiếng như ResNet, UNet, DenseNet, và Transformer đều khai thác kỹ thuật này theo nhiều cách sáng tạo và hiệu quả.

Triển khai code ta đucợ kết quả sau

Concept mà chúng ta đã tìm hiểu thì được lấy idea từ kiến trúc resnet, cho phép tín hiệu truyền qua các lớp sâu mà không bị suy giảm, từ đó giúp khắc phục hiện tượng biến mất gradient và cải thiện khả năng hội tụ của mạng nơ-ron sâu.



Hình 112: Enter Caption

Ý tưởng ban đầu xuất phát từ ResNet, và sau này được mở rộng thành DenseNet — một kiến trúc cũng dựa trên skip connection nhưng có mức kết nối dày đặc hơn: mỗi khối (block) được kết nối trực tiếp với tất cả các khối phía sau. Mặc dù DenseNet có thể mang lại hiệu quả huấn luyện cao hơn, song chi phí tính toán tăng đáng kể do số lượng kết nối lớn. Vì lý do đó, trong phạm vi báo cáo này, chúng tôi chỉ tập trung trình bày kiến trúc ResNet.

## 10 Tạm kết

---

Trải qua các chương của tài liệu, chúng ta đã cùng nhau thực hiện một hành trình toàn diện, đi từ những khái niệm cơ bản nhất của hồi quy tuyến tính đến việc xây dựng và tối ưu hóa các kiến trúc mạng nơ-ron tích chập phức tạp. Mục tiêu xuyên suốt của tài liệu là không chỉ trình bày “cái gì” mà còn làm rõ “tại sao” và “như thế nào”, nhằm xây dựng một nền tảng kiến thức vững chắc và một trực giác sâu sắc về cơ chế hoạt động của các mô hình học sâu.

Chúng ta đã thấy rằng, dù các mô hình có đa dạng và phức tạp đến đâu, chúng đều được xây dựng dựa trên một vài nguyên lý cốt lõi:

1. **Định nghĩa kiến trúc mô hình (Computational Graph):** Thiết lập mối quan hệ toán học từ đầu vào đến đầu ra.
2. **Thiết kế hàm mất mát (Loss Function):** Định lượng sự khác biệt giữa dự đoán của mô hình và giá trị thực tế.
3. **Sử dụng thuật toán tối ưu hóa (Optimizer):** Tự động điều chỉnh các tham số của mô hình (trọng số) để giảm thiểu hàm mất mát, với thuật toán lan truyền ngược (backpropagation) đóng vai trò trung tâm.

Từ Linear Regression, Logistic Regression, MLP cho đến CNN, hành trình này cho thấy sự tiến hóa của các kiến trúc để giải quyết những bài toán ngày càng phức tạp, đặc biệt là trong lĩnh vực thị giác máy tính. Các kỹ thuật nâng cao như phương pháp khởi tạo, thuật toán tối ưu hóa Adam, hay các kỹ thuật chính quy hóa như Dropout và Batch Normalization không còn là những “hộp đen” mà đã trở thành những công cụ hữu hiệu giúp chúng ta kiểm soát quá trình huấn luyện, chống lại hiện tượng quá khớp (overfitting) và cải thiện hiệu năng tổng quát của mô hình. Việc làm quen với một framework hiện đại như PyTorch đã giúp kết nối lý thuyết với thực hành, cho phép hiện thực hóa các ý tưởng một cách nhanh chóng và hiệu quả.

Lĩnh vực học sâu vẫn đang phát triển với tốc độ vũ bão. Những kiến thức được trình bày trong tài liệu này là nền tảng cơ bản nhưng vô cùng quan trọng. Đây là điểm khởi đầu để bạn có thể tiếp tục khám phá các lĩnh vực hấp dẫn khác như Mạng Nơ-ron Hồi quy (RNN) cho dữ liệu chuỗi, kiến trúc Transformer đang thống trị xử lý ngôn ngữ tự nhiên, hay các mô hình sinh (Generative Models) như GANs.

Hy vọng rằng tài liệu này đã cung cấp cho bạn một nền tảng vững chắc, khơi dậy niềm đam mê và trang bị đủ sự tự tin để bạn có thể tự mình xây dựng, huấn luyện và cải tiến các mô hình học sâu. Chúc bạn thành công trên con đường chinh phục lĩnh vực đầy thách thức và tiềm năng này.