Module 6 - Week 3

Multi Layer Perceptron

TimeSeries Team

Ngày 26 tháng 11 năm 2025

I. Từ tản mạn về Softmax Regression đến động lực của mạng neuron nhiều lớp

1.1 Giới thiệu về Multi Layer Perceptron

Trong bài trước, chúng ta đã thấy rằng Softmax Regression là một công cụ mạnh mẽ cho bài toán phân loại đa lớp, đặc biệt bởi khả năng giải quyết vấn đề "phần bù" trong mô hình logistic truyền thống. Thay vì chỉ dự đoán một giá trị điểm số cho từng lớp, Softmax đưa tất cả các lớp vào cùng một không gian quyết định: mỗi nhãn không còn tồn tại độc lập mà ảnh hưởng trực tiếp đến xác suất của các nhãn còn lại.

Nhờ cơ chế chuẩn hoá bằng hàm mũ, mô hình không chỉ dự đoán "lớp nào phù hợp nhất", mà còn ước lượng được xác suất phân bố trên toàn bộ các lớp, phản ánh đúng bản chất của một hệ phân loại đa lớp: để hiểu một nhãn, ta phải nhìn vào toàn bộ nhãn khác.

Tuy nhiên, Softmax Regression vẫn có một giới hạn quan trọng: mô hình chỉ học được ranh giới tuyến tính giữa các lớp. Trong nhiều bài toán thực tế, từ nhận dạng chữ số, phân biệt vật thể đến phân loại hình ảnh y khoa, quan hệ giữa dữ liệu và nhãn phức tạp hơn rất nhiều so với một mặt phẳng phân tách đơn giản.

Chính từ nhu cầu vượt qua giới hạn này, Multilayer Perceptron (MLP) ra đời. Ta có thể xem MLP như một cách tổng quát hoá Logistic Regression hay Softmax Regression: thay vì chỉ có một tầng tính toán tuyến tính, ta chồng nhiều tầng perceptron lên nhau, mỗi tầng lại được nối với các hàm kích hoạt phi tuyến. Việc đưa phi tuyến vào mô hình mở rộng đáng kể khả năng biểu diễn, cho phép mạng học được các cấu trúc ẩn, mẫu hình phức tạp, và quan hệ không tuyến tính trong dữ liệu.

Nói cách khác, nếu Softmax Regression chỉ "vẽ đường thẳng" để phân lớp, thì MLP có thể "uốn cong" không gian đặc trưng, tạo ra những ranh giới linh hoạt và tinh vi hơn, phù hợp với bản chất đa chiều của nhiều bài toán hiện đại

Bước sang một góc nhìn cấu trúc hơn, một mô hình MLP điển hình gồm ba nhóm tầng:

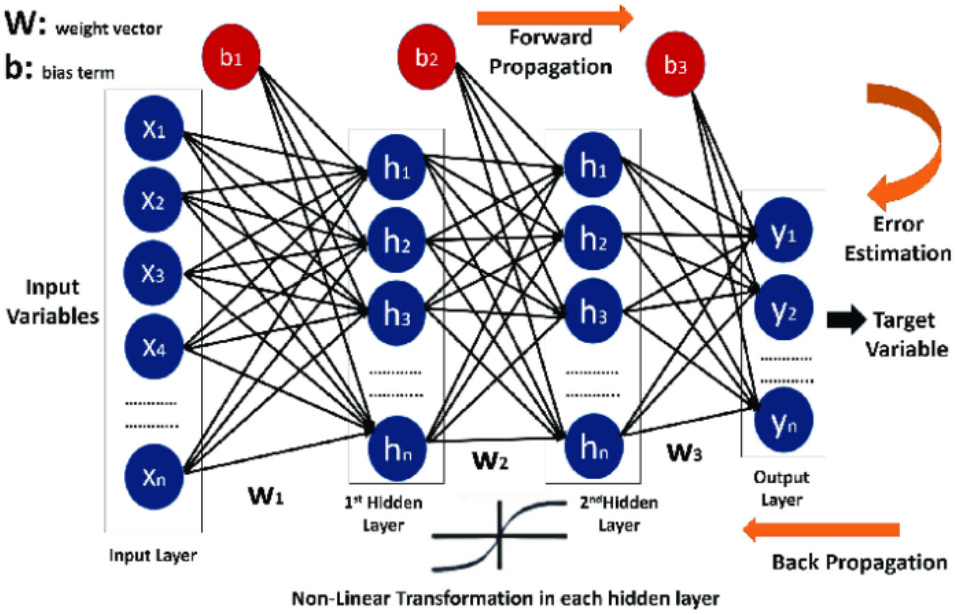
1. (1) tầng vào nhận dữ liệu gốc
2. (2) các tầng ẩn thực hiện biến đổi không tuyến tính
3. (3) tầng ra đưa ra dự đoán cuối cùng

Mỗi tầng ẩn có thể được xem như một phép ánh xạ học được, chuyển dữ liệu từ không gian biểu diễn này sang không gian biểu diễn khác. Khi nhiều tầng như vậy được xếp chồng lên nhau, mạng có khả năng từng bước trích xuất những đặc trưng trừu tượng hơn, từ các mẫu đơn giản ở tầng đầu đến các cấu trúc phức tạp ở tầng cuối. Đây chính là điểm khiến MLP vượt trội so với các mô hình tuyến tính đơn tầng.

Quá trình dự đoán trong mạng xảy ra theo một dòng chảy gọi là forward propagation: dữ liệu được truyền từ tầng vào qua từng tầng ẩn, lần lượt đi qua phép biến đổi tuyến tính và hàm kích hoạt phi tuyến. Tại tầng cuối, mạng xuất ra một vector điểm số, thường được chuẩn hoá bằng Softmax để thu được phân bố xác suất.

Ngược lại, việc học của mạng diễn ra thông qua backward propagation. Ở hướng này, sai số giữa dự đoán và giá trị thật được lan ngược từ đầu ra về các tầng trước đó. Nhờ quy tắc đạo hàm chuỗi, mạng

biết được mỗi tham số đóng góp thế nào vào sai số tổng thể, từ đó cập nhật trọng số để cải thiện chất lượng dự đoán. Chính sự kết hợp giữa forward nhằm tính toán đầu ra và backward nhằm tối ưu hoá tham số đã tạo nên cơ chế học hiệu quả của MLP — một nền tảng kéo theo sự bùng nổ của nhiều mô hình học sâu ngày nay.



Hình trên minh họa trực quan dòng chảy thông tin trong một mạng MLP. Các ma trận trọng số W_1, W_2, W_3 cùng với các véc-tơ bias b_1, b_2, b_3 quyết định cách tín hiệu được biến đổi qua từng tầng. Ở mỗi tầng ẩn, dữ liệu lần lượt đi qua phép biến đổi tuyến tính và hàm kích hoạt phi tuyến (được biểu diễn bằng đường cong phía dưới hình). Mũi tên phía trên cho thấy quá trình *forward propagation*, trong khi mũi tên cong bên phải biểu diễn *backward propagation* – nơi sai số được lan ngược để cập nhật các tham số. Tổng thể sơ đồ nhấn mạnh vai trò của các tầng ẩn như những phép ánh xạ có thể học được, giúp mạng xây dựng dần các biểu diễn trừu tượng hơn của dữ liệu qua từng lớp.

1.2 Thiết lập Multi Layer Perceptron trong PyTorch

Dưới đây là cách thiết lập điển hình cho một mô hình MLP trong `PyTorch`. Ví dụ sau minh hoạ một mạng đơn giản gồm hai tầng tuyến tính và một hàm kích hoạt phi tuyến:

```
model = nn.Sequential(
    nn.Linear(2, 2),
    nn.Linear(2, 2),
    nn.Sigmoid()
)
summary(model, (10000000, 2))
```

```python
model = nn.Sequential(
    nn.Linear(2, 2),
    nn.Linear(2, 2),
    nn.Sigmoid()
)
```

```python
summary(model, (10000000, 2))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1          [-1, 10000000, 2]               6
            Linear-2          [-1, 10000000, 2]               6
           Sigmoid-3          [-1, 10000000, 2]               0
================================================================
Total params: 12
Trainable params: 12
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 76.29
Forward/backward pass size (MB): 457.76
Params size (MB): 0.00
Estimated Total Size (MB): 534.06
----------------------------------------------------------------
```

Kết quả từ hàm `summary` cho thấy mỗi tầng `Linear(2, 2)` chứa tổng cộng $2 \times 2 + 2 = 6$ tham số (gồm 4 trọng số và 2 hệ số bias), và toàn bộ mạng có 12 tham số có thể học được.

Một điểm đáng lưu ý là kích thước đầu vào được đặt là $10{,}000{,}000 \times 2$. Khi batch có số lượng mẫu rất lớn như vậy, bộ nhớ dành cho dữ liệu và các tensor trung gian trong quá trình lan truyền thuận và lan truyền ngược tăng lên đáng kể. Điều này thể hiện rõ qua báo cáo bộ nhớ trong phần `Estimated Total Size`, nơi chi phí cho quá trình forward/backward chiếm phần lớn dung lượng, mặc dù bản thân mô hình chỉ gồm 12 tham số.

Quan sát này nhấn mạnh rằng chi phí tính toán của MLP không chỉ phụ thuộc vào số lượng tham số của mô hình, mà còn phụ thuộc mạnh vào kích thước batch và kích thước các *activation* được tạo ra trong quá trình huấn luyện.

1.3 Giới thiệu về các hàm kích hoạt Activation Function

Một khi đã xây dựng được kiến trúc của mạng MLP, câu hỏi tự nhiên tiếp theo là: *nên dùng hàm kích hoạt nào cho các tầng ẩn và tầng đầu ra?* Hàm kích hoạt (activation function) chính là yếu tố tạo ra tính phi tuyến, giúp MLP vượt qua giới hạn của các mô hình tuyến tính như Logistic Regression hay Softmax Regression. Dưới đây là các hàm kích hoạt thường gặp nhất trong bối cảnh học về MLP.

Sigmoid

Hàm sigmoid là lựa chọn kinh điển trong các mạng nơ-ron thời kỳ đầu. Sigmoid "nén" giá trị đầu vào về khoảng $(0, 1)$ theo công thức:

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

Sigmoid có ưu điểm dễ hiểu, mượt, và cho cảm giác rất tự nhiên khi muốn mô hình hoá một "xác suất". Vì vậy, nó vẫn thường được dùng ở **tầng đầu ra của bài toán phân loại nhị phân**.

Tuy nhiên, khi dùng ở các tầng ẩn, sigmoid gặp nhiều hạn chế:

- Gradient rất nhỏ khi đầu vào lớn hoặc quá âm (gọi là *vanishing gradient*). Đạo hàm của hàm sigmoid có dạng

$$\sigma'(x) = \sigma(x)\big(1 - \sigma(x)\big),$$

và luôn nhỏ hơn hoặc bằng 0.25. Khi x rất lớn hoặc rất âm, $\sigma(x)$ gần như bằng 1 hoặc 0, nên $\sigma'(x) \approx 0$.

Trong quá trình *backpropagation*, gradient tại một tầng được tính bằng cách *nhân* gradient của các tầng phía sau với đạo hàm của hàm kích hoạt tại tầng đó. Nếu mỗi tầng đều cho ra một đạo hàm rất nhỏ (ví dụ cỡ 0.1 hay 0.01), thì sau khi đi qua nhiều tầng, gradient sẽ bị "co" lại gần như bằng 0.

Bạn có thể hình dung như việc truyền tin bằng cách thì thầm: mỗi người chỉ nghe được khoảng 10% âm lượng của người trước. Sau vài chục người, âm thanh ban đầu gần như biến mất hoàn toàn. Với sigmoid trong mạng sâu cũng vậy: thông tin về sai số chưa kịp lan ngược hết các tầng thì đã bị "tiêu biến" trên đường, khiến các tầng đầu rất khó học được điều gì.

- Đầu ra không có trung bình bằng 0, khiến việc tối ưu chậm hơn. Hàm sigmoid luôn cho giá trị trong khoảng $(0, 1)$. Điều này có nghĩa là đa số các đầu ra đều nằm ở phía dương, khiến các kích hoạt không được phân bố quanh điểm 0. Khi đó, trong quá trình lan truyền ngược, các đạo hàm liên quan đến trọng số sẽ có xu hướng đẩy các cập nhật theo cùng một hướng (vì mọi giá trị đều dương).

Bạn có thể hình dung như việc lái xe mà vô-lăng luôn hơi lệch sang một bên: dù bạn cố gắng chỉnh, xe lúc nào cũng bị kéo về hướng đó, khiến việc điều khiển khó khăn hơn.

Trong tối ưu hoá mạng nơ-ron cũng vậy: khi đầu ra của một tầng không tập trung quanh 0, các bản cập nhật trọng số trở nên không cân bằng, dẫn đến việc hội tụ chậm hơn và đôi khi dao động mạnh. Đó là lý do các hàm kích hoạt có đầu ra đối xứng quanh 0 như $\tanh(x)$ thường cho tốc độ học tốt hơn sigmoid trong các tầng ẩn.

Chính những điểm này đã khiến sigmoid dần nhường chỗ cho các hàm kích hoạt hiệu quả hơn trong các tầng ẩn.

Hyperbolic Tangent (tanh)

Hàm *tanh* là một phiên bản cải tiến hơn của sigmoid:

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

Không giống sigmoid, đầu ra của tanh nằm trong khoảng $(-1, 1)$ và có trung bình bằng 0. Sự đối xứng này giúp việc tối ưu dễ dàng hơn vì các gradient lan truyền ngược không bị *kéo lệch* về một phía như sigmoid.

Nhược điểm của tanh là nó **không giải quyết triệt để** vấn đề vanishing gradient — khi đầu vào quá lớn, hàm vẫn "bị bão hoà" và gradient gần như biến mất. Điều này có thể được lí giải như sau:

Đạo hàm của hàm tanh là:

$$\tanh'(x) = 1 - \tanh^2(x).$$

Khi đầu vào x có giá trị rất lớn (dương hoặc âm), ta có:

$$\tanh(x) \approx 1 \quad \text{hoặc} \quad \tanh(x) \approx -1.$$

Lúc đó:

$$\tanh'(x) = 1 - (\pm 1)^2 = 0.$$

Điều này có nghĩa là khi tín hiệu đi vào quá mạnh, hàm tanh đưa đầu ra về một giá trị gần như cố định (sát 1 hoặc sát -1). Vì đầu ra gần như không thay đổi khi x thay đổi, đạo hàm — tức tốc độ thay đổi của hàm — gần như bằng 0. Đây chính là hiện tượng *bão hoà gradient* (saturation).

Một cách hình dung: bạn tưởng tượng tanh như một nút điều chỉnh âm lượng của radio. Ở mức vừa phải, xoay nút một chút thì âm lượng thay đổi thấy rõ. Nhưng nếu bạn đã vặn hết cỡ lên mức 100%, xoay thêm nữa cũng chẳng tăng được nữa.

Mặc dù vậy, tanh vẫn được xem là lựa chọn tốt hơn sigmoid nếu muốn dùng ở tầng ẩn, đặc biệt trong các mô hình cổ điển hoặc bài toán có dữ liệu được chuẩn hoá quanh 0.

ReLU và các biến thể đơn giản (LeakyReLU, PReLU)

Sự xuất hiện của ReLU đã thay đổi toàn bộ cách xây dựng mạng sâu:

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

ReLU rất đơn giản nhưng lại cực kỳ hiệu quả:

- Không có vùng bão hoà ở phía dương $\Rightarrow$ giảm đáng kể vanishing gradient.
- Tính toán nhanh (không cần hàm mũ).
- Thường giúp mạng hội tụ nhanh hơn và đạt hiệu năng cao hơn.

Sự đơn giản này lại tạo ra một lợi thế rất lớn: phần dương của ReLU có đạo hàm bằng 1, không hề suy giảm. Điều này giúp gradient truyền ngược mạnh mẽ hơn nhiều so với sigmoid hay tanh.

Còn phần âm, ReLU đặt toàn bộ giá trị về 0. Điều này có thể hiểu như việc loại bỏ những tín hiệu không hữu ích cho quá trình học. Ví dụ, khi phân loại hình ảnh có phải là "con mèo" hay không, ta không cần quan tâm tới những đặc tính sai lệch như “mèo có đẻ trứng không”, “mèo có lông vũ không” hay “mèo có cánh không”. Những thông tin này hoàn toàn vô nghĩa đối với nhiệm vụ, nên việc gán chúng bằng 0 trong ReLU tương đương với việc bỏ qua các đặc điểm không liên quan.

Nếu coi việc truyền gradient giống như truyền ánh sáng qua lớp kính, thì sigmoid và tanh giống như lớp kính làm mờ — ánh sáng càng truyền xa càng yếu. ReLU thì giống như mặt kính trong suốt: miễn là tín hiệu nằm ở vùng dương, gradient đi qua gần như nguyên vẹn.

Nhờ đặc tính này, ReLU trở thành lựa chọn mặc định cho hầu hết các tầng ẩn trong MLP và các mô hình sâu hơn.

Nhược điểm lớn nhất của ReLU là vấn đề **dead neurons**: nếu giá trị vào của một neuron luôn âm, gradient sẽ bằng 0 và neuron đó không học được nữa.

Để khắc phục điều này, các biến thể nhẹ như **LeakyReLU** được dùng:

$$\text{LeakyReLU}(x) = \begin{cases} x, & x \geq 0, \\ \alpha x, & x < 0, \end{cases}$$

với α thường rất nhỏ (ví dụ 0.01). Ý tưởng đơn giản: luôn giữ một lượng gradient khác 0 ở phía âm để neuron không “chết”.

PReLU (Parametric ReLU) đi xa hơn khi cho phép mạng *tự học* độ dốc ở phần âm:

$$\text{PReLU}(x) = \begin{cases} x, & x > 0, \\ ax, & x \leq 0, \end{cases}$$

trong đó a là tham số learnable được tối ưu bằng backpropagation.

Cách hiểu trực quan: thay vì cố định độ dốc ở phần âm, mạng sẽ tự tìm ra mức độ “mở cửa” phù hợp nhất để cho thông tin đi qua.

PReLU giúp mạng thích ứng tốt hơn cho từng tầng và từng đặc trưng, đặc biệt trong các mô hình có độ sâu vừa phải.

Softmax

Trong bối cảnh MLP, Softmax được dùng **duy nhất ở tầng đầu ra của bài toán phân loại đa lớp**. Hàm Softmax biến vector điểm số thành một phân bố xác suất:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}.$$

Hiểu đơn giản: Softmax giống như việc biến mọi điểm số thành "phiếu bầu" — lớp có điểm số cao nhất sẽ chiếm phần lớn xác suất, nhưng các lớp khác vẫn có trọng số nhỏ phản ánh mức độ cạnh tranh giữa chúng.

Softsign

Hàm Softsign có dạng:

$$f(x) = \frac{x}{1 + |x|}.$$

Nó hoạt động tương tự tanh nhưng "dịu" hơn, hội tụ theo dạng đa thức thay vì dạng mũ. Dù không phổ biến bằng ReLU hay tanh, Softsign đôi khi được dùng trong các mô hình nhỏ hoặc bài toán hồi quy.

Swish

Swish là một hàm kích hoạt hiện đại, được đề xuất bởi Google:

$$\text{Swish}(x) = x \cdot \sigma(x).$$

Swish mượt, phi tuyến mạnh và không đơn điệu. Trực giác dễ hiểu: thay vì cắt bỏ phần âm như ReLU, Swish cho phép "một chút thông tin" đi qua ngay cả khi x âm. Nhờ đó, Swish thường mang lại khả năng tối ưu tốt hơn ReLU trong các mạng nhiều tầng, nhưng tính toán chậm hơn do phụ thuộc vào sigmoid.

Tóm tắt Tùy theo vị trí trong mạng MLP, ta có thể lựa chọn hàm kích hoạt như sau:

- **Tầng ẩn:** ReLU, LeakyReLU, PReLU, hoặc Swish.
- **Phân loại nhị phân:** Sigmoid ở tầng đầu ra.
- **Phân loại đa lớp:** Softmax ở tầng đầu ra.
- **Khi dữ liệu chuẩn hoá quanh 0:** tanh có thể là lựa chọn phù hợp.

Hàm kích hoạt chính là điểm làm nên sự "thông minh" của MLP — nếu thiếu chúng, dù mạng có nhiều tầng đến đâu, mô hình vẫn chỉ là một phép biến đổi tuyến tính đơn giản.

1.4 Hàm Loss

a. Binary Cross Entropy Loss

Hàm Binary Cross Entropy (BCE) Loss được sử dụng trong các bài toán **phân loại nhị phân**, nơi mỗi mẫu chỉ thuộc một trong hai lớp (ví dụ: đúng/sai, có/không, mèo/không phải mèo).

Công thức của Binary Cross Entropy Loss được cho bởi:

$$\mathcal{L}_{\text{BCE}}(y, \hat{y}) = -\left[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right].$$

Trong đó:

- $y \in \{0, 1\}$ là nhãn *ground-truth*,
- $\hat{y} \in (0, 1)$ là xác suất dự đoán của mô hình, thường là đầu ra của hàm sigmoid.

Trực giác của BCE Loss khá dễ hiểu:

- Nếu nhãn thật $y = 1$, mô hình sẽ bị phạt nặng khi $\hat{y}$ nhỏ.
- Nếu nhãn thật $y = 0$, mô hình sẽ bị phạt nặng khi $\hat{y}$ lớn.

Nói cách khác, hàm loss buộc mô hình phải dự đoán xác suất càng gần với nhãn thật càng tốt, và hình phạt tăng rất nhanh khi mô hình **tự tin nhưng sai**.

b. Cross Entropy Loss

Cross Entropy Loss là hàm loss tiêu chuẩn cho bài toán **phân loại đa lớp**. Khác với bài toán nhị phân, ở đây mỗi mẫu có thể thuộc một trong C lớp khác nhau.

Công thức Cross Entropy Loss được viết như sau:

$$\mathcal{L}_{\text{CE}}(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i).$$

Trong đó:

- y là vector nhãn *ground-truth* dạng one-hot encoding,
- $\hat{y}$ là vector xác suất dự đoán của mô hình sau khi đi qua hàm Softmax,
- C là số lượng lớp.

Vì hàm Softmax cho ra một phân bố xác suất trên tất cả các lớp, ta cần chuyển nhãn ground-truth về cùng dạng vector để có thể so sánh trực tiếp. Ý tưởng này được thực hiện thông qua **one-hot encoding**: lớp đúng được gán giá trị 1, các lớp còn lại được gán 0.

Nhờ vậy, biểu thức loss thực chất chỉ giữ lại thành phần $\log(\hat{y}_k)$ ứng với lớp đúng k, còn các lớp sai không đóng góp vào giá trị loss. Điều này phản ánh trực giác quan trọng: *mô hình chỉ bị phạt dựa trên mức độ tự tin của nó đối với nhãn đúng.*

1.5 Optimizer - Tản mạn về SGD và Adam

Sau khi đã xây dựng được kiến trúc mạng và xác định hàm loss, câu hỏi tiếp theo là: *làm thế nào để điều chỉnh các trọng số của mạng sao cho loss ngày càng nhỏ?* Câu trả lời nằm ở **optimizer** — thành phần chịu trách nhiệm dẫn đường cho quá trình học.

Có thể hình dung việc huấn luyện mạng nơ-ron như một hành trình đi xuống đáy của một thung lũng (hàm loss). Mỗi bước cập nhật trọng số là một bước di chuyển, và optimizer quyết định *đi hướng nào* và *đi nhanh hay chậm*.

Trong số rất nhiều optimizer đã được đề xuất, hai lựa chọn cơ bản và phổ biến nhất là **Stochastic Gradient Descent (SGD)** và **Adam**.

Stochastic Gradient Descent (SGD)

SGD là optimizer đơn giản và lâu đời nhất. Thay vì sử dụng toàn bộ dữ liệu để cập nhật trọng số, SGD chỉ dùng một mẫu hoặc một mini-batch nhỏ ở mỗi bước. Nhờ đó, quá trình học trở nên nhanh và có thể mở rộng cho các tập dữ liệu lớn.

Cách hoạt động của SGD có thể hiểu như sau: mỗi lần mô hình nhìn vào một phần nhỏ dữ liệu, tính gradient của loss, sau đó điều chỉnh trọng số một chút theo hướng làm loss giảm xuống.

Vì chỉ dựa trên một phần dữ liệu, đường đi của SGD thường không mượt mà, mà có xu hướng *zig-zag*. Tuy nhiên, chính sự nhiễu này đôi khi lại giúp mô hình thoát khỏi các điểm tối ưu cục bộ và đạt khả năng tổng quát tốt hơn.

Các biến thể như **SGD với momentum** giúp quá trình cập nhật trở nên ổn định hơn, bằng cách tích luỹ "đà" từ các bước trước đó thay vì chỉ nhìn vào gradient hiện tại.

Adam (Adaptive Moment Estimation)

Adam có thể xem là một phiên bản cải tiến của SGD. Thay vì dùng chung một learning rate cho tất cả trọng số, Adam tự động điều chỉnh learning rate cho từng tham số dựa trên lịch sử gradient.

Nói một cách trực quan, Adam vừa:

- *nhớ hướng đi cũ* (momentum),
- vừa *tự điều chỉnh độ dài bước đi* tuỳ theo mức độ ổn định của gradient.

Nhờ đó, Adam thường hội tụ nhanh hơn và dễ sử dụng hơn trong thực tế, đặc biệt với các mô hình nhiều tầng hoặc dữ liệu nhiễu. Người dùng thường không cần tinh chỉnh learning rate quá kỹ mà vẫn đạt kết quả tốt.

So sánh trực quan

Có thể hình dung sự khác biệt giữa SGD và Adam như sau: SGD giống người đi bộ với bước chân cố định — đơn giản, bền bỉ, và hiệu quả khi quãng đường dài và địa hình tương đối ổn định. Adam thì giống xe tự động điều chỉnh tốc độ — tăng nhanh khi đường thoáng, giảm chậm khi sắp đến đích.

Trong thực hành:

- **SGD** thường cho khả năng tổng quát tốt, đặc biệt trong các bài toán thị giác.
- **Adam** thường được ưu tiên khi cần hội tụ nhanh hoặc khi huấn luyện mạng sâu.

Không có optimizer nào là "tốt nhất cho mọi bài toán". Việc lựa chọn SGD hay Adam phụ thuộc vào dữ liệu, kiến trúc mạng và mục tiêu của bài toán đang xét.

II. Multi Layer Perceptron trong bài toán dữ liệu ảnh

2.1 Dữ liệu ảnh

Trong học máy, ảnh không còn được nhìn như một đối tượng trực quan, mà được biểu diễn dưới dạng các **ma trận số** chứa giá trị cường độ điểm ảnh (pixel). Mỗi pixel tương ứng với một giá trị số thực, phản ánh mức độ sáng hoặc thông tin màu sắc tại vị trí đó trong ảnh.

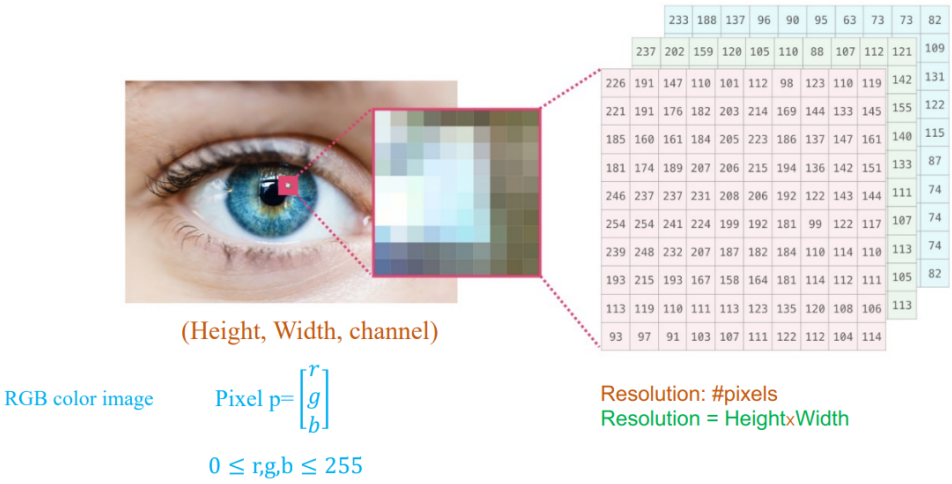
Tuỳ theo loại dữ liệu, ảnh thường được biểu diễn theo hai dạng chính:

- **Ảnh xám (Grayscale image):** Ảnh xám được biểu diễn dưới dạng một ma trận hai chiều kích thước (chiều cao × chiều rộng). Mỗi phần tử trong ma trận là một giá trị cường độ sáng, thường nằm trong khoảng từ 0 đến 255, với 0 biểu thị màu đen hoàn toàn và 255 biểu thị màu trắng hoàn toàn.

Hình 1: Ví dụ ảnh xám được biểu diễn dưới dạng ma trận 2 chiều.

- **Ảnh màu (Color image):** Ảnh màu thường được biểu diễn bởi ba ma trận hai chiều xếp chồng lên nhau, tương ứng với ba kênh màu **Red**, **Green** và **Blue** (RGB). Khi đó, ảnh có thể được xem như một tensor ba chiều với kích thước (chiều cao × chiều rộng × 3).

  Mỗi kênh màu lưu trữ cường độ màu tương ứng tại mỗi pixel, và sự kết hợp của ba kênh này tạo nên màu sắc cuối cùng của ảnh.



Hình 2: Ví dụ ảnh màu với ba kênh RGB được xếp chồng thành tensor 3 chiều.

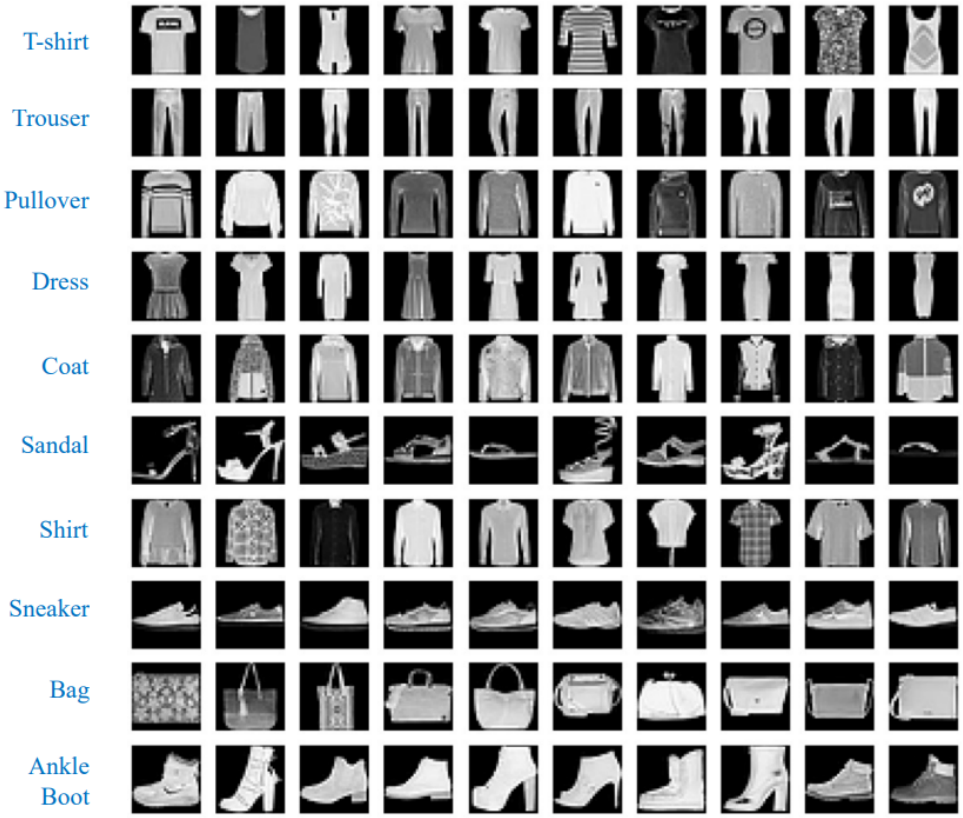
2.2 Bài toán phân loại ảnh trên bộ dữ liệu FASHION-MNIST

Fashion-MNIST là một bộ dữ liệu chuẩn thường được sử dụng trong học máy và học sâu như một bước khởi đầu cho bài toán phân loại ảnh. Bộ dữ liệu bao gồm các hình ảnh quần áo và phụ kiện thời

trang, được thiết kế để thay thế cho MNIST chữ số truyền thống nhưng vẫn giữ mức độ đơn giản tương tự.

Cụ thể, Fashion-MNIST có các đặc điểm sau:

- **Định dạng:** Ảnh xám (grayscale).
- **Độ phân giải:** 28×28 pixels.
- **Số lượng:** 60,000 ảnh huấn luyện và 10,000 ảnh kiểm thử.
- **Số nhãn:** 10 loại sản phẩm thời trang (áo, quần, giày, túi xách, ...).



Hình 3: Bộ dữ liệu FASHION-MNIST với 10 nhãn phân loại.

Trong bài toán này, ta sẽ sử dụng **PyTorch** để xây dựng và so sánh mô hình **Softmax Regression** và **Multi Layer Perceptron (MLP)** cho nhiệm vụ phân loại ảnh.

a. Chuẩn bị dữ liệu

Trước hết, dữ liệu Fashion-MNIST được tải về bằng thư viện `torchvision`. Các ảnh ban đầu tồn tại dưới dạng `PIL.Image`, không thuận tiện cho việc tính toán trong PyTorch. Vì vậy, ta cần chuyển ảnh sang dạng tensor.

```
from torchvision import datasets, transforms

transform = transforms.Compose([
    transforms.ToTensor()
```

```
])

train_dataset = datasets.FashionMNIST(
    root='./data', train=True, download=True, transform=transform)

test_dataset = datasets.FashionMNIST(
    root='./data', train=False, download=True, transform=transform)
```

Hàm `ToTensor()` thực hiện hai việc quan trọng:

- Chuyển ảnh từ dạng `PIL.Image` sang tensor.
- Chuẩn hoá giá trị pixel từ khoảng $[0, 255]$ về $[0, 1]$.

Ngoài ra, ảnh cũng được chuyển về định dạng (C, H, W) theo chuẩn xử lý của PyTorch, trong đó $C = 1$ đối với ảnh xám.

b. Tải dữ liệu bằng DataLoader

Trong thực tế, việc đưa toàn bộ dữ liệu ảnh vào bộ nhớ và huấn luyện cùng lúc là không khả thi, đặc biệt khi kích thước tập dữ liệu lớn. Điều này không những tốn bộ nhớ mà còn khiến quá trình huấn luyện trở nên chậm và kém ổn định.

Vì vậy, PyTorch cung cấp `DataLoader` để chia dữ liệu thành các **batch nhỏ** và nạp dữ liệu theo từng phần trong quá trình huấn luyện.

```
from torchvision.datasets import FashionMNIST
from torch.utils.data import DataLoader
from torchvision import transforms

transform = transforms.Compose([transforms.ToTensor()])

trainset = FashionMNIST(
    root='data',
    train=True,
    download=True,
    transform=transform
)

trainloader = DataLoader(
    trainset,
    batch_size=1024,
    num_workers=2,
    shuffle=True
)

print(len(trainloader))
```

Việc chia dữ liệu thành các batch mang lại nhiều lợi ích:

- Giảm chi phí bộ nhớ khi huấn luyện.
- Tăng tốc độ tính toán thông qua vector hoá.
- Giúp mô hình học ổn định hơn nhờ cập nhật gradient từng phần.

DataLoader và khái niệm Generator

Có thể xem `DataLoader` như một dạng **generator** trong Python: thay vì nạp toàn bộ dữ liệu một lần, nó chỉ sinh ra từng batch ảnh khi mô hình thực sự cần đến. Cách làm này giúp tiết kiệm bộ nhớ và cho phép huấn luyện trên các tập dữ liệu lớn mà không vượt quá giới hạn phần cứng.

Batch cuối và vấn đề nhiễu

Trong một số trường hợp, batch cuối cùng có thể chứa số lượng mẫu rất ít (do tổng số mẫu không chia hết cho batch size). Batch nhỏ này đôi khi có thể gây nhiễu trong quá trình huấn luyện và làm giá trị loss dao động mạnh.

Để xử lý vấn đề này, ta có thể:

- Loại bỏ batch cuối bằng tuỳ chọn `drop_last=True`, hoặc
- Chấp nhận sự nhiễu nhỏ này, vì trong nhiều trường hợp ảnh hưởng của nó là không đáng kể.

d. Thiết kế mô hình

Mô hình Softmax Regression

Softmax Regression có thể xem là một trường hợp đặc biệt đơn giản của mô hình MLP, trong đó mạng **không có tầng ẩn**, mà chỉ gồm tầng đầu vào và tầng đầu ra, sau đó áp dụng hàm softmax để dự đoán xác suất cho từng lớp.

Do dữ liệu đầu vào là ảnh có kích thước 28×28, trong khi lớp `Linear` trong PyTorch chỉ nhận đầu vào dạng vector 1 chiều, ta cần **flatten** ảnh trước khi đưa vào mô hình. Việc flatten này chuyển ảnh từ tensor 2D thành một vector có chiều $28 \times 28 = 784$.

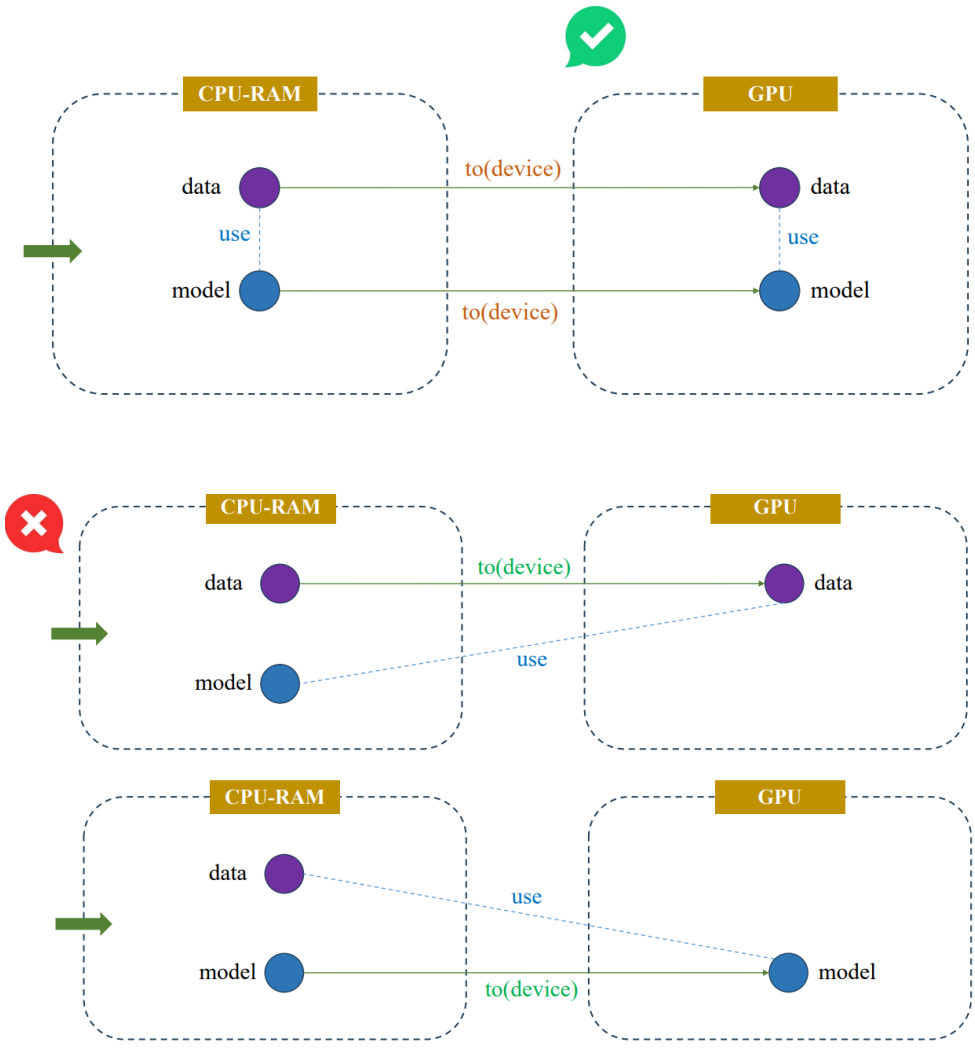
```
class SoftmaxRegression(nn.Module):
    def __init__(self, input_size=28*28, num_classes=10):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, x):
        x = self.flatten(x)
        x = self.linear(x)
        return x
```

Sau khi dữ liệu đầu vào được chuyển về đúng kích thước, vector đặc trưng sẽ được truyền qua lớp `Linear` để tính toán các *logit* tương ứng với từng lớp.

Lưu ý rằng trong PyTorch, hàm `CrossEntropyLoss` đã tự động kết hợp `Softmax` và `LogLoss` trong quá trình tính toán loss. Vì vậy, ta **không cần định nghĩa riêng một lớp softmax** trong hàm `forward` của mô hình. Việc giữ đầu ra dưới dạng logit giúp quá trình huấn luyện ổn định hơn và tránh các vấn đề sai số số học.

Lưu ý về việc sử dụng `to(device)`

CPU và GPU có hai vùng nhớ độc lập để lưu trữ dữ liệu và mô hình, được minh hoạ trong hình sau:



Một phép tính trong PyTorch chỉ có thể thực hiện khi **toàn bộ tensor và mô hình cùng nằm trên một vùng nhớ**. Nếu dữ liệu nằm trên CPU trong khi mô hình nằm trên GPU (hoặc ngược lại), PyTorch sẽ không thể thực hiện phép toán và sinh ra lỗi.

Trực giác có thể hiểu đơn giản rằng: các tensor chỉ có thể "tương tác" với nhau khi chúng ở cùng một nơi. Nếu dữ liệu ở CPU nhưng mô hình ở GPU, chúng không thể "nhìn thấy nhau" để thực hiện phép nhân ma trận hay tính loss.

Để đảm bảo sự đồng bộ này, PyTorch cung cấp lệnh `to(device)`, giúp chuyển dữ liệu và mô hình sang cùng một thiết bị tính toán. Việc sử dụng đúng `to(device)` giúp tránh lỗi trong quá trình huấn luyện và đảm bảo toàn bộ pipeline hoạt động nhất quán.

Thiết bị tính toán được xác định thông qua câu lệnh:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Câu lệnh trên cho phép PyTorch tự động kiểm tra:

- Nếu GPU khả dụng, toàn bộ mô hình và dữ liệu sẽ được đưa lên GPU để tăng tốc tính toán.
- Nếu không có GPU, PyTorch sẽ mặc định sử dụng CPU.

Trong thực hành, cách sử dụng chuẩn là:

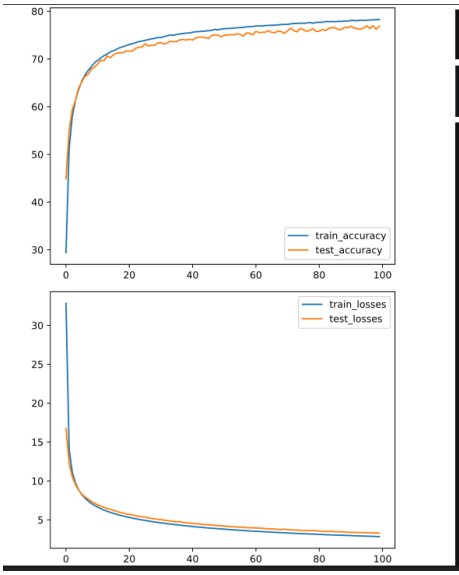
- Chuyển mô hình sang `device` bằng `model.to(device)`.

- Chuyển từng batch dữ liệu sang `device` bằng `data.to(device)`.

Cách làm này đảm bảo dữ liệu và mô hình luôn nằm chung một vùng nhớ, từ đó quá trình lan truyền thuận (forward propagation) và lan truyền ngược (backpropagation) được thực hiện chính xác và hiệu quả. Chính vì vậy, phải luôn nhớ *"Model ở đâu, dữ liệu phải ở đó."*

Lưu ý về Normalization

Trong quá trình huấn luyện mô hình, việc chuẩn hoá dữ liệu đầu vào (*normalization*) đóng vai trò rất quan trọng. Nếu dữ liệu không được normalization, gradient có thể dao động mạnh khiến quá trình tối ưu trở nên khó khăn. Khi đó, ta thường buộc phải chọn learning rate rất nhỏ để mô hình học ổn định, dẫn đến tốc độ hội tụ chậm như trong hình sau khi độ chính xác cho cả tập train và test đều chưa đến 0.80




```python
model = nn.Sequential(nn.Flatten(), nn.Linear(784, 10))
model = model.to(device)
```

```python
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.00001)
```

```python
# train
for epoch in range(max_epoch):
    for i, (inputs, labels) in enumerate(trainloader, 0):
        # Move inputs and labels to the device
        inputs, labels = inputs.to(device), labels.to(device)

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        # Determine class predictions and track accuracy
        _, predicted = torch.max(outputs.data, 1)
        correct += (predicted == labels).sum().item()

        # Backward pass and optimization
        loss.backward()
        optimizer.step()
```

Trong PyTorch, normalization thường được thực hiện thông qua `transforms.Normalize(mean, std)`, với công thức tổng quát:

$$\text{Image}_{\text{norm}} = \frac{\text{Image} - \text{mean}}{\text{std}}.$$

Các phương thức normalization trong PyTorch

- **Chuẩn hoá về khoảng** $[0, 1]$ Đây là kết quả mặc định sau khi áp dụng `ToTensor()`, khi đó ảnh ban đầu trong khoảng $[0, 255]$ được chia cho 255.

```
transform = transforms.Compose([
    transforms.ToTensor()
])
```

- **Chuẩn hoá về khoảng** $[-1, 1]$ Cách này giúp dữ liệu có trung bình xấp xỉ 0 và độ lệch chuẩn xấp xỉ 1, thường giúp quá trình lan truyền gradient ổn định hơn.

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
```

```
])
```

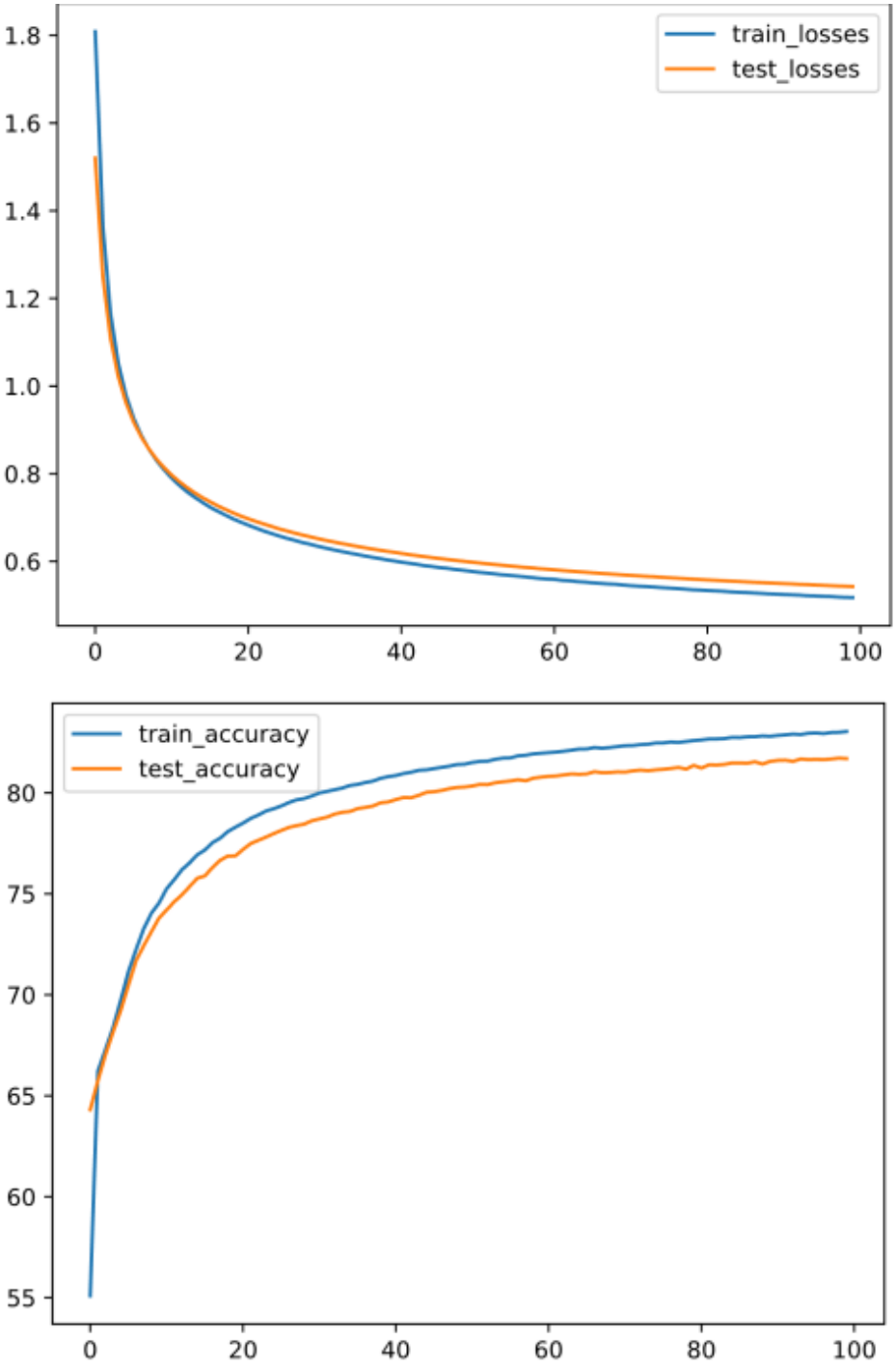
- **Z-score Normalization** Ở cách này, dữ liệu được chuẩn hoá dựa trên mean và std được tính trực tiếp từ tập huấn luyện.

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])
```

So sánh hiệu năng giữa các phương thức normalization

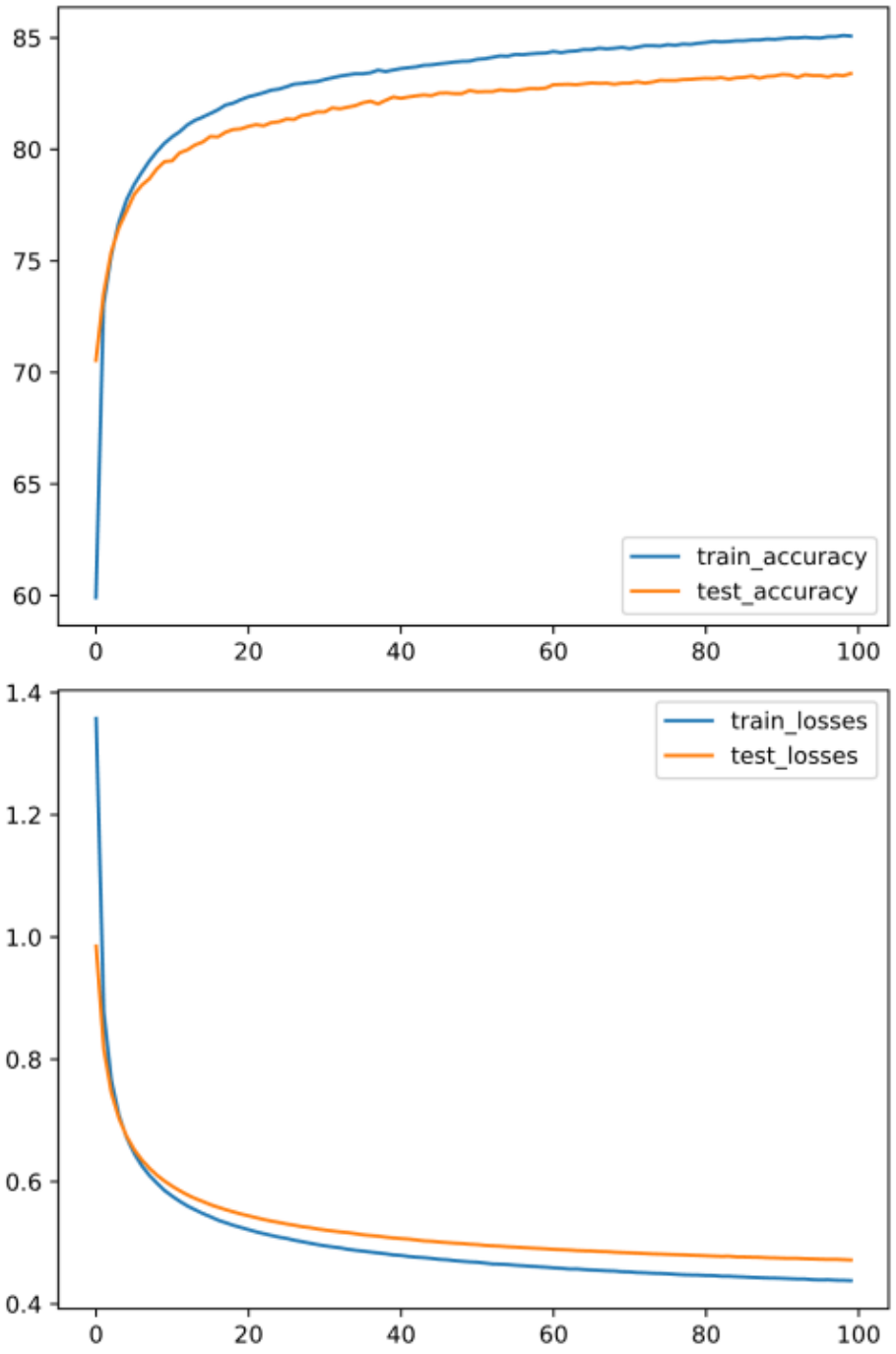
Dựa trên kết quả thực nghiệm, ta có thể đưa ra một số nhận xét như sau:

- Khi chỉ chuyển ảnh về khoảng $[0, 1]$, mô hình đạt độ chính xác cao nhất trên tập train và test vào khoảng 0.82–0.83.



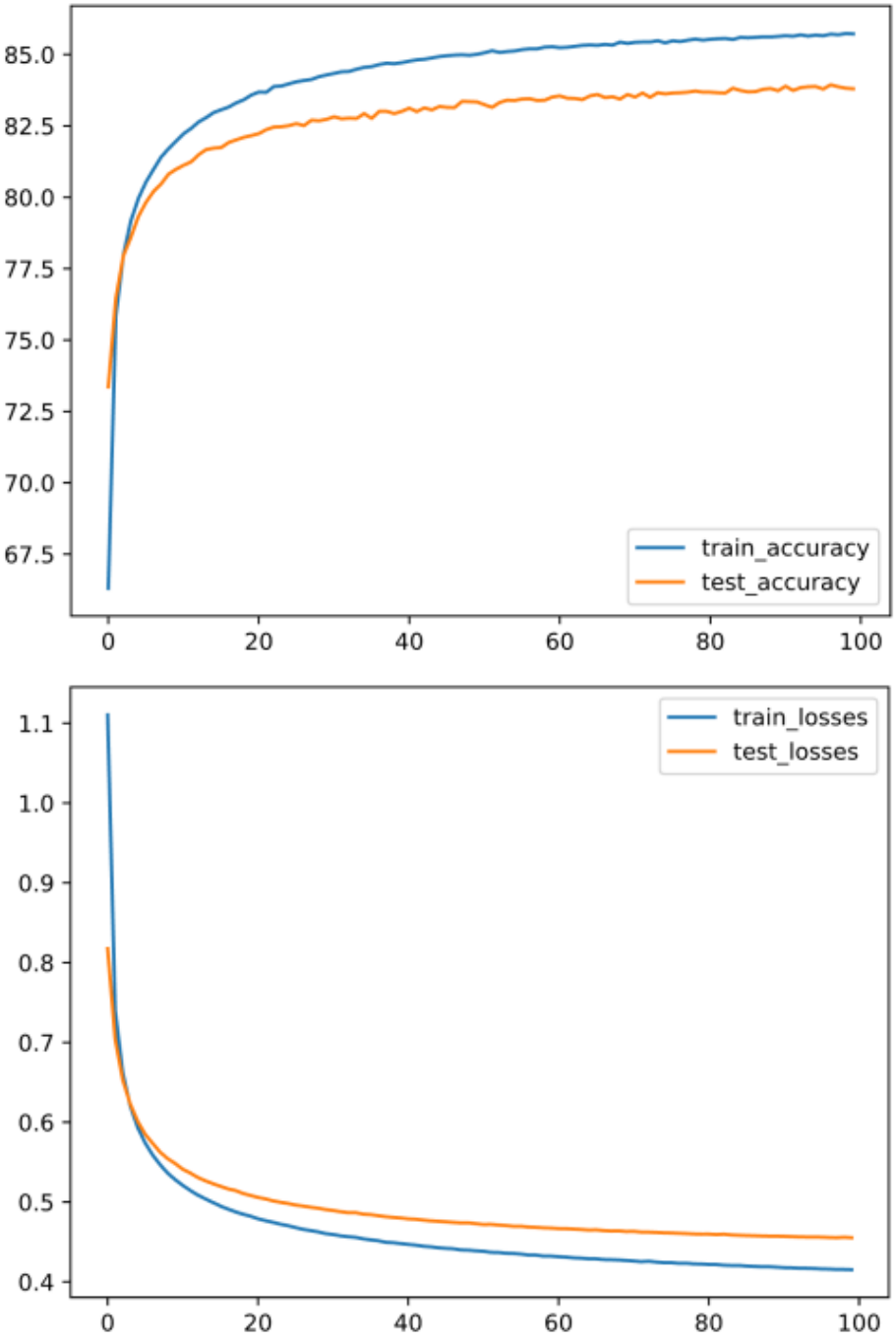
Hình 4: Hiệu suất mô hình khi chuẩn hóa về khoảng $[0, 1]$

- Khi chuẩn hoá ảnh về khoảng $[-1, 1]$, hiệu năng được cải thiện rõ rệt, với độ chính xác cao nhất trên cả tập train và test đạt gần 0.85.



Hình 5: Hiệu suất mô hình khi chuẩn hóa về khoảng $[-1, 1]$

- Với Z-score normalization, mô hình đạt độ chính xác trên tập train khoảng 0.85, tuy nhiên độ chính xác trên tập test giảm nhẹ, vào khoảng 0.825, cho thấy dấu hiệu overfitting nhẹ.



Hình 6: Hiệu suất mô hình khi chuẩn hóa theo z-score

Kết quả này cho thấy normalization không chỉ giúp mô hình hội tụ nhanh hơn, mà còn ảnh hưởng trực tiếp đến khả năng tổng quát hoá. Trong bài toán Fashion-MNIST với mô hình Softmax Regression, việc chuẩn hoá dữ liệu về khoảng $[-1, 1]$ cho kết quả cân bằng nhất giữa độ chính xác và tính ổn định trong quá trình huấn luyện.

Mô hình Multi Layer Perceptron

Khác với Softmax Regression chỉ gồm một lớp tuyến tính, Multi Layer Perceptron (MLP) mở rộng mô hình bằng cách **thêm các tầng ẩn cùng hàm kích hoạt phi tuyến**. Điều này cho phép mô hình học được các quan hệ phức tạp hơn giữa các pixel trong ảnh.

Mô hình MLP được sử dụng trong thí nghiệm có kiến trúc như sau:

```
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 256),
    nn.ReLU(),
    nn.Linear(256, 10)
)
model = model.to(device)
```

Ở đây:

- `nn.Flatten()` chuyển ảnh 28×28 thành vector 784 chiều.
- `Linear(784, 256)` là tầng ẩn đầu tiên, trích xuất đặc trưng từ ảnh.
- `ReLU()` đưa tính phi tuyến vào mô hình, giúp mạng học các mẫu phức tạp.
- `Linear(256, 10)` là tầng đầu ra, sinh ra logit cho 10 lớp.

So với Softmax Regression, MLP không chỉ nhìn ảnh như một vector phẳng, mà còn học cách kết hợp các pixel để tạo ra đặc trưng trừu tượng hơn (thí dụ như cạnh, hình dạng hoặc cấu trúc vùng ảnh).

Dưới đây là đoạn code huấn luyện một mô hình multi layer perceptron đơn giản

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

max_epoch = 5
for epoch in range(max_epoch):
    for i, (inputs, labels) in enumerate(trainloader, 0):

        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        loss.backward()
        optimizer.step()

    print(f"Epoch [{epoch + 1}/{max_epoch}]")
```

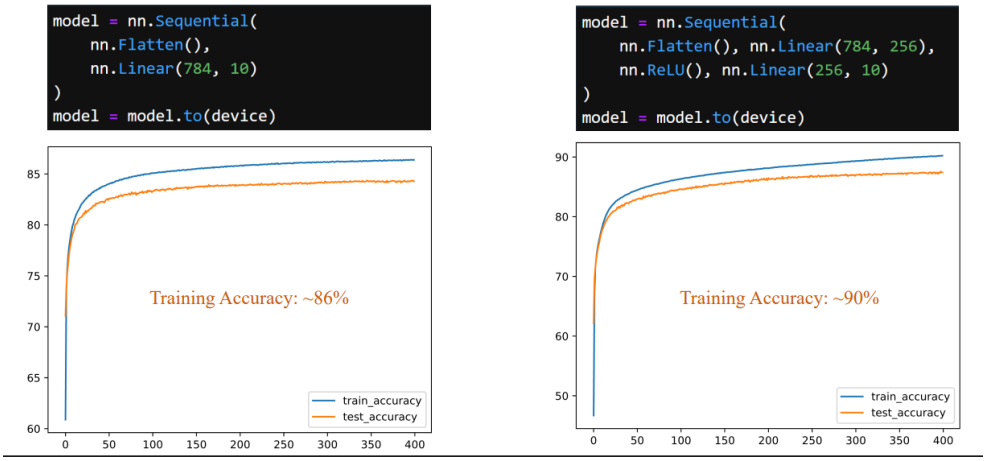
Sau đó, ta thực hiện đánh giá trên tập test

```
correct = 0
total = 0
with torch.no_grad():
    for images, labels in testloader:
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
```

```
        _, predicted = torch.max(outputs.data, 1)

        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total
print(f"accuracy: {accuracy}")
```

So sánh Softmax Regression và MLP

Dựa trên kết quả thực nghiệm như hình, ta có thể rút ra các nhận xét sau:



Hình 7: So sánh hiệu suất của mô hình Softmax và MLP

- **Softmax Regression** đạt độ chính xác huấn luyện khoảng 86%. Mô hình đơn giản, học nhanh, nhưng bị giới hạn bởi tính tuyến tính.
- **MLP với một tầng ẩn** đạt độ chính xác huấn luyện gần 90%. Việc bổ sung tầng ẩn và hàm ReLU giúp mô hình học được các quan hệ phi tuyến trong dữ liệu ảnh.

Như vậy Sự cải thiện hiệu năng cho thấy rằng dữ liệu ảnh, dù có độ phân giải thấp như Fashion-MNIST, vẫn chứa những cấu trúc mà mô hình tuyến tính không thể biểu diễn đầy đủ.

MLP, dù chỉ thêm một tầng ẩn đơn giản, đã có khả năng trích xuất đặc trưng tốt hơn, từ đó cải thiện rõ rệt độ chính xác so với Softmax Regression.

III: Một số vấn đề về Multi Layer Perceptron và các câu hỏi mở rộng

Phụ lục: Mở rộng hiểu biết về Multi Layer Perceptron

Sau khi xây dựng và huấn luyện thành công một mô hình MLP cơ bản, câu hỏi quan trọng không còn là *"làm sao để chạy được mô hình"*, mà là *"vì sao mô hình lại được thiết kế như vậy"*. Phần mở rộng này nhằm làm rõ những quyết định thiết kế cốt lõi trong MLP, từ đó giúp người đọc hình thành tư duy phản biện khi xây dựng mạng nơ-ron.

1. Vai trò cốt lõi của hàm kích hoạt

Một tầng `Linear` trong MLP thực chất chỉ là một phép biến đổi tuyến tính:

$$h = Wx + b.$$

Nếu ta xếp chồng nhiều tầng tuyến tính liên tiếp mà *không có hàm kích hoạt*, toàn bộ mạng có thể được rút gọn thành *một* phép biến đổi tuyến tính duy nhất. Nói cách khác, mạng dù sâu đến đâu cũng không thể biểu diễn được các quan hệ phi tuyến trong dữ liệu.

Hàm kích hoạt xuất hiện như một "điểm bẻ cong" không gian đặc trưng. Mỗi lần đi qua một hàm phi tuyến (như ReLU), dữ liệu được ánh xạ sang một không gian biểu diễn mới, giúp mạng dần học được các cấu trúc phức tạp hơn. Do đó, kiến trúc chuẩn của MLP thường có dạng:

$$\text{Linear} \to \text{Activation} \to \text{Linear} \to \text{Activation} \to \cdots$$

Từ lập luận này, ta có thể rút ra một nguyên tắc quan trọng: **mỗi tầng tuyến tính muốn "tạo thêm năng lực biểu diễn" đều cần đi kèm một hàm kích hoạt phía sau.**

Gợi mở tư duy:

- Nếu đặt activation *sau tất cả các tầng Linear* thay vì sau mỗi tầng, liệu mô hình có khác gì Softmax Regression?
- Điều này gợi ý gì về vai trò của tầng ẩn so với tầng đầu ra?

2. Vì sao tầng đầu ra thường không dùng ReLU?

Trong bài toán phân loại nhiều lớp, tầng đầu ra của MLP thường không trả về xác suất trực tiếp, mà trả về các *logit* — tức là các giá trị thực chưa bị ràng buộc về khoảng. Những logit này sau đó được đưa vào hàm `CrossEntropyLoss`, nơi phép chuẩn hoá softmax được thực hiện một cách ngầm định để chuyển chúng thành phân bố xác suất.

Việc giữ các logit ở dạng giá trị thực không bị giới hạn là rất quan trọng cho quá trình tối ưu. Softmax không chỉ quan tâm đến giá trị tuyệt đối của từng logit, mà còn dựa vào *sự khác biệt tương đối* giữa chúng để xác định xác suất của mỗi lớp. Chính sự chênh lệch này cung cấp tín hiệu gradient rõ ràng giúp mô hình biết nên tăng hay giảm logit của lớp nào.

Nếu ta đặt ReLU ở tầng cuối, mọi logit âm sẽ bị cắt về 0. Điều này dẫn đến hai hệ quả nghiêm trọng.

Thứ nhất, **thông tin bị mất**. Khi các giá trị âm đều trở thành 0, mô hình không còn phân biệt được một lớp "rất không phù hợp" hay chỉ "hơi không phù hợp". Sự phân cấp giữa các lớp bị nén lại, khiến softmax nhận đầu vào đã bị méo mó.

Thứ hai, **gradient trở nên nghèo nàn**. Trong quá trình lan truyền ngược, các neuron bị ReLU chặn (đầu vào âm) sẽ có gradient bằng 0. Điều này khiến một phần thông tin sai số không thể truyền ngược về các tầng trước, làm giảm hiệu quả học. Nói cách khác, tối ưu hoá đòi hỏi dòng thông tin sai số phải được bảo toàn xuyên suốt mạng, trong khi ReLU ở tầng cuối lại chủ động làm mất thông tin đó.

Hệ quả cuối cùng là ý nghĩa xác suất sau softmax bị sai lệch: một số lớp có thể được gán xác suất bằng nhau chỉ vì logit của chúng đều bị cắt về 0, chứ không phải vì mô hình thực sự cho rằng chúng tương đương.

Mặt khác, khi ReLu nằm ở tầng ẩn, nó đóng vai trò như một bộ lọc, biến đổi và trích xuất đặc trưng từ dữ liệu đầu vào. Đây không phải là mất thông tin ngẫu nhiên, mà là chủ động loại bỏ những tín hiệu không hữu ích, và một đặc trưng bị ReLU chặn ở tầng này vẫn có thể được biểu diễn lại bởi neuron khác hoặc tầng sau. Nói cách khác, ở tầng ẩn này, ta cho phép mất đi thông tin cục bộ, miễn là không phá vỡ cấu trúc tổng thể của không gian đặc trưng

Vì những lý do trên, một nguyên tắc thiết kế phổ biến được rút ra là:

> **Hàm kích hoạt được sử dụng ở các tầng ẩn để tăng khả năng biểu diễn phi tuyến, nhưng không áp dụng cho tầng đầu ra trong bài toán phân loại sử dụng Cross Entropy Loss.**

Nguyên tắc này đảm bảo rằng tầng cuối cùng giữ trọn vẹn thông tin cần thiết cho softmax, đồng thời cho phép gradient lan truyền ngược một cách đầy đủ để tối ưu hoá toàn bộ mạng.

3. Khai báo activation trong `__init__` hay dùng `functional`?

Trong PyTorch, cùng một hàm kích hoạt có thể được sử dụng theo hai cách: khai báo như một module trong `__init__`, hoặc gọi trực tiếp trong `forward` thông qua `torch.nn.functional`.

Về mặt toán học, hai cách này cho kết quả tương đương đối với các activation không có tham số học được như ReLU hay Tanh. Tuy nhiên, về mặt thiết kế phần mềm, chúng mang ý nghĩa khác nhau.

Khi khai báo activation trong `__init__`, toàn bộ kiến trúc mạng được thể hiện rõ ràng, giúp ta dễ kiểm tra, in mô hình, và quản lý cấu trúc mạng. Cách này đặc biệt quan trọng khi activation có tham số học được (ví dụ như PReLU), vì các tham số đó cần được PyTorch theo dõi trong `model.parameters()`.

Ngược lại, việc sử dụng `nn.functional` mang tính linh hoạt và gọn nhẹ, thường phù hợp khi thử nghiệm nhanh hoặc khi activation chỉ xuất hiện một lần trong luồng tính toán.

Nguyên tắc tổng quát:

> Nếu một thành phần có trạng thái hoặc tham số cần học, hãy khai báo nó trong `__init__`.
> Nếu chỉ là phép toán thuần túy, `functional` là đủ.

4. MLP và giới hạn khi xử lý dữ liệu ảnh

Trong ví dụ Fashion-MNIST, ta dùng `Flatten` để biến ảnh 28×28 thành một vector 784 chiều. Cách làm này đơn giản và hiệu quả cho bài toán nhỏ, nhưng nó xoá bỏ hoàn toàn cấu trúc không gian của ảnh.

MLP, vì thế, không "nhìn thấy" quan hệ lân cận giữa các pixel, mà chỉ xử lý ảnh như một dãy số. Giới hạn này giải thích vì sao MLP nhanh chóng đạt trần hiệu năng, và vì sao các kiến trúc như CNN được phát triển để tận dụng cấu trúc không gian của dữ liệu thị giác.

Gợi mở tư duy:

- Nếu giữ nguyên số tham số, một CNN đơn giản có thể vượt MLP bao nhiêu trên Fashion-MNIST?
- Điều này nói gì về mối quan hệ giữa kiến trúc mạng và cấu trúc dữ liệu? Có thể tăng kích thước MLP để khắc phục vấn đề này không?
- Vì sao CNN lại phù hợp tự nhiên hơn cho dữ liệu ảnh?

5. Nên dùng bao nhiêu tầng ẩn và bao nhiêu neuron?

Ta đã dùng cấu hình `Linear(784, 256)` $\rightarrow$ `ReLU` $\rightarrow$ `Linear(256, 10)`. Nhưng tại sao là 256? Tại sao chỉ một tầng ẩn?

Một mặt, nếu tầng ẩn quá nhỏ, mạng không đủ "sức chứa" để mô hình hoá các đặc trưng phức tạp. Mặt khác, nếu tầng ẩn quá lớn hoặc quá nhiều tầng, mạng dễ bị overfitting và tốn tài nguyên tính toán.

Không có một con số "chuẩn" cho mọi bài toán, nhưng có thể tự đặt cho mình những câu hỏi:

- Nếu tăng số neuron từ 256 lên 512, độ chính xác train/test thay đổi thế nào?
- Nếu thêm một tầng ẩn nữa (ví dụ: 784–256–128–10), loss train có giảm nhanh hơn không, và test accuracy có thực sự cải thiện không?
- Khi quan sát khoảng cách giữa train và test accuracy, ta có nhận thấy dấu hiệu underfitting hay overfitting?

Và có lẽ, để trả lời những câu hỏi này, ta cần tiến thành thực nghiệm trên bài toán cụ thể và rút ra insights chính là câu trả lời "chuẩn" nhất cho riêng bài toán đó

6. Một tầng ẩn lớn có tương đương nhiều tầng ẩn nhỏ không?

Về mặt lý thuyết, một MLP chỉ cần một tầng ẩn đủ lớn đã có thể xấp xỉ mọi hàm liên tục (theo Định lý Xấp xỉ Toàn năng - Universal Approximation Theorem). Điều này có thể dẫn đến suy nghĩ rằng ta chỉ cần một tầng ẩn thật rộng là đủ.

Tuy nhiên, trong thực tế, một tầng rất rộng thường khó tối ưu, dễ overfitting và kém hiệu quả về mặt tính toán. Ngược lại, nhiều tầng ẩn nhỏ cho phép mạng học theo cấu trúc phân cấp: các tầng đầu học đặc trưng đơn giản, các tầng sau kết hợp chúng thành biểu diễn trừu tượng hơn. Đây là một trong những lý do khiến các mạng sâu hoạt động hiệu quả hơn trong nhiều bài toán thực tế.

Gợi mở tư duy:

- Với cùng số lượng tham số, mạng sâu hơn có học tốt hơn mạng nông không?
- Cách mạng học đặc trưng có tương đồng gì với cách con người hình thành khái niệm?

7. Gradient biến mất là vấn đề của hàm kích hoạt hay của kiến trúc?

Hiện tượng *vanishing gradient* thường được gán cho các hàm kích hoạt như sigmoid hoặc tanh. Tuy nhiên, bản chất của vấn đề không chỉ nằm ở activation, mà còn liên quan đến độ sâu của mạng, cách khởi tạo trọng số và cách chuẩn hoá dữ liệu.

ReLU giúp giảm hiện tượng gradient biến mất, nhưng không giải quyết triệt để nguyên nhân. Điều này giải thích vì sao các kỹ thuật như Batch Normalization hay Residual Connection được phát triển trong các kiến trúc sâu hơn.

Gợi mở tư duy:

- Nếu dùng tanh nhưng chuẩn hoá dữ liệu tốt, mạng có còn học được không?
- ReLU là giải pháp căn bản hay chỉ là lựa chọn thực dụng?

8. Khi nào MLP lại là lựa chọn phù hợp?

Dù có hạn chế với dữ liệu ảnh, MLP vẫn là mô hình rất hiệu quả trong nhiều bài toán khác, chẳng hạn như dữ liệu bảng, vector đặc trưng đã được trích xuất sẵn, hay embedding trong các hệ thống gợi ý và NLP.

Trong những trường hợp này, MLP khai thác tốt mối quan hệ toàn cục giữa các đặc trưng, đồng thời giữ được sự đơn giản và hiệu quả trong huấn luyện.

Gợi mở tư duy:

- Kiến trúc mạng nên được lựa chọn dựa trên dữ liệu hay bài toán?
- Có tồn tại một kiến trúc phù hợp cho mọi loại dữ liệu không?

9. MLP có thật sự "hiểu" dữ liệu không?

Khi mô hình đạt độ chính xác rất cao trên tập huấn luyện nhưng kém trên tập kiểm thử, ta nói rằng mô hình bị overfitting. Về bản chất, mạng học quá chi tiết các mẫu cụ thể, thay vì trích xuất các quy luật chung.

Các kỹ thuật như regularization, dropout hay early stopping được đưa vào không phải để làm mô hình mạnh hơn, mà để ép mạng học các đặc trưng ổn định và tổng quát hơn.

Gợi mở tư duy:

- Độ chính xác cao có đồng nghĩa với việc mô hình hiểu dữ liệu?
- "Hiểu" trong học máy có giống "hiểu" trong nhận thức con người không?

Kết luận

MLP là bước khởi đầu quan trọng để hiểu học sâu, không phải vì nó là mô hình mạnh nhất, mà vì nó giúp làm rõ những khái niệm nền tảng: phi tuyến, lan truyền gradient, tối ưu và regularization.

Một khi hiểu rõ những quyết định nhỏ trong MLP, đặt activation ở đâu, dùng dạng nào, hay vì sao một số thiết kế tưởng hợp lý lại không hiệu quả, ta đã có nền tảng vững chắc để tiếp cận những kiến trúc sâu hơn như CNN, RNN hay Transformer với tư duy chủ động thay vì chỉ sao chép công thức.