Module 6 - Week 3

Multi Layer Perceptron (continue)

TimeSeries Team

Ngày 26 tháng 11 năm 2025

Những góc nhìn sâu sắc về MLP

Ở bài viết này, ta sẽ bàn đến một số khía cạnh nho nhỏ khi làm việc với Multi Layer Perceptron, ngoài ra sẽ không nhắc lại những phần đã được đề cập trong bài ngày thứ 3 và thứ 4.

1.1 Sự bất biến với biến đối tuyến tính trong mạng neuron

Chứng minh rằng chuẩn hóa giúp bất biến với biến đổi tuyến tính

Giả sử ta có một tập dữ liệu đầu vào $X = \{X_1, X_2, \ldots, X_n\}$. Phép chuẩn hoá (standardization) của X được định nghĩa là:

$$\bar{X} = \frac{X - \mu_X}{\sigma_X},$$

trong đó

$$\mu_X = \frac{1}{n}\sum_{i=1}^{n} X_i, \qquad \sigma_X = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_i - \mu_X)^2}.$$

Xét một phép biến đổi tuyến tính của dữ liệu:

$$Y = aX + b,$$

với $a \neq 0$ và b là hằng số. Khi đó, kỳ vọng và độ lệch chuẩn của Y lần lượt là:

$$\mu_Y = \frac{1}{n}\sum_{i=1}^{n}(aX_i + b) = a\mu_X + b,$$

$$\sigma_Y = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\big((aX_i + b) - \mu_Y\big)^2} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\big(a(X_i - \mu_X)\big)^2} = |a|\sigma_X.$$

Do đó, biến chuẩn hoá của Y là:

$$\bar{Y} = \frac{Y - \mu_Y}{\sigma_Y} = \frac{aX + b - (a\mu_X + b)}{|a|\sigma_X} = \frac{a(X - \mu_X)}{|a|\sigma_X} = \frac{a}{|a|} \cdot \frac{X - \mu_X}{\sigma_X}.$$

Suy ra:

$$\bar{Y} = \begin{cases} \bar{X}, & \text{nếu } a > 0, \\ -\bar{X}, & \text{nếu } a < 0. \end{cases}$$

Tóm lại:

Kết quả trên cho thấy phép chuẩn hoá loại bỏ hoàn toàn ảnh hưởng của:

- **Tịnh tiến** ($+b$): bị triệt tiêu khi trừ đi trung bình.
- **Co giãn tuyến tính** ($\times a$): bị triệt tiêu khi chia cho độ lệch chuẩn.

Nói cách khác, sau khi chuẩn hoá, dữ liệu chỉ còn giữ lại *hình dạng phân bố tương đối*, không phụ thuộc vào độ sáng, độ tương phản hay các phép biến đổi tuyến tính của đầu vào.

Ý nghĩa đối với mạng nơ-ron:

Chuẩn hoá giúp mô hình học được các đặc trưng bất biến trước những thay đổi tuyến tính của dữ liệu, từ đó giúp việc huấn luyện ổn định hơn và khả năng tổng quát hoá tốt hơn.

Điều này giải thích vì sao trong thực tế, các ảnh có độ sáng hoặc tương phản khác nhau vẫn được mô hình nhận diện là cùng một đối tượng sau khi chuẩn hoá.

1.2 Hiểu thêm về Skip Connection - tiền để cho ResNet

Trực giác ban đầu của skip connection rất đơn giản: *thay vì bắt một tầng ẩn phải tự mình học toàn bộ phép biến đổi từ đầu vào x sang đầu ra mới, ta cho nó một "đường tắt" để ít nhất cũng có thể sao chép lại chính x nếu cần.* Nói cách khác, thay vì học trực tiếp

$$h(x) \approx \text{output},$$

ta cho mạng học

$$\text{output} = x + F(x),$$

trong đó $F(x)$ là phần *sai lệch cần điều chỉnh* (residual). Nếu $F(x)$ học kém, ta vẫn còn thành phần x đi qua nguyên vẹn. Đây chính là ý tưởng cốt lõi đằng sau các kiến trúc như ResNet.

Ở góc nhìn lan truyền ngược, skip connection giống như việc ta mở thêm một làn đường cho gradient. Nếu một nhánh đi qua các tầng phi tuyến sâu bên trong bị suy giảm (vanishing gradient), gradient vẫn có thể đi theo nhánh *identity* với đạo hàm xấp xỉ bằng 1, giúp tín hiệu lỗi quay trở lại các tầng trước đó dễ dàng hơn.

Minh hoạ đơn giản trong PyTorch. Ta thiết kế một MLP nhỏ, trong đó đầu ra của tầng ẩn thứ nhất vừa được đưa qua hàm kích hoạt, vừa được giữ lại để cộng (skip) với biểu diễn sau đó:

```
import torch
import torch.nn as nn

class MyMLP(nn.Module):
    def __init__(self, hidden_dim=256):
        super().__init__()
        self.fc1 = nn.Linear(784, hidden_dim)
        self.activation = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim, 10)

    def forward(self, x):
        # x: (B, 784)
        x1 = self.fc1(x)              # (B, 256) - nhánh chính
        x  = self.activation(x1)      # (B, 256) - qua hàm kích hoạt

        # Skip connection: cộng trực tiếp biểu diễn trước activation
        x  = x + x1                   # (B, 256)

        x  = self.fc2(x)              # (B, 10) - logit đầu ra
        return x
```

```
# Test
model = MyMLP(256)
X = torch.rand(32, 784)              # batch 32 ảnh đã flatten
output = model(X)
print(output.shape)                  # torch.Size([32, 10])
```

Một vài góc nhìn thú vị từ ví dụ nhỏ này:

- **Hai con đường cho thông tin:** cùng một tín hiệu x_1 sau `fc1` đi theo hai nhánh: một nhánh qua ReLU (phi tuyến), một nhánh giữ nguyên. Khi cộng lại, mạng vừa tận dụng được phi tuyến, vừa giữ lại thông tin tuyến tính ban đầu.
- **Học "phần chênh lệch" thay vì học từ số 0:** nếu `fc2` học rất kém, output vẫn không quá tệ vì đã có sẵn thành phần x_1. Còn khi mạng học tốt, `fc2` sẽ điều chỉnh phần residual sao cho $x_1 + F(x_1)$ tiệm cận nghiệm tối ưu.
- **Gradient có đường tắt để quay về:** trong lan truyền ngược, đạo hàm của nhánh identity (đường `x1` được cộng thẳng) là 1, giúp một phần gradient quay lại `fc1` mà không bị bóp nhỏ bởi nhiều tầng phi tuyến. Đây chính là tinh thần "deep nhưng vẫn train được" của ResNet.

Mặc dù ví dụ trên chỉ là một MLP nhỏ, song nó đã chứa đầy đủ tinh thần của một khối residual:

$$h(x) = F(x) + x,$$

trong đó F là một mạng con (ở đây là `ReLU + fc2`). Khi thay MLP bằng các khối convolution và xếp chồng hàng chục, hàng trăm lần, ta sẽ thu được kiến trúc ResNet nổi tiếng.

1.3 Dying ReLU và những vấn đề liên quan

Như đã từng đề cập ở bài trước, ReLU (*Rectified Linear Unit*) là một trong những hàm kích hoạt phổ biến nhất nhờ tính đơn giản và hiệu quả:

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

Đặc điểm quan trọng của ReLU là:

- Giá trị âm bị cắt về 0.
- Phần dương có đạo hàm bằng 1, giúp gradient lan truyền tốt.

Chính đặc điểm này giúp mạng trở nên *sparse* (chỉ một phần neuron được kích hoạt), làm mô hình hiệu quả và ít overfitting hơn. Tuy nhiên, mặt trái của ReLU cũng xuất hiện từ chính sự đơn giản đó.

Dying ReLU là gì?

Dying ReLU mô tả hiện tượng khi một neuron ReLU luôn cho đầu ra bằng 0 với *mọi* đầu vào. Điều này xảy ra khi giá trị đưa vào ReLU luôn nằm trong miền âm:

$$Wx + b < 0 \quad \forall x.$$

Khi đó:

- Đầu ra của neuron luôn bằng 0.
- Đạo hàm của ReLU trong miền âm bằng 0.
- Gradient không thể lan truyền ngược qua neuron này.

Neuron bị rơi vào trạng thái này được gọi là một *dead neuron*. Ở kịch bản xấu nhất, nếu phần lớn neuron trong mạng bị chết, toàn bộ mô hình có thể suy biến thành một hàm hằng số, mất hoàn toàn khả năng học.

Vì sao neuron ReLU đã chết thì khó hồi sinh?

Một khi neuron rơi hoàn toàn vào miền âm, gradient tại đó bằng 0 nên các tham số W, b không còn được cập nhật. Điều này khiến neuron bị "kẹt" vĩnh viễn trong trạng thái không hoạt động. Khác với các hàm kích hoạt trơn như sigmoid hay tanh, ReLU không cung cấp bất kỳ gradient nào trong miền âm để "kéo" neuron quay lại.

Nguyên nhân gây ra Dying ReLU

(i) Learning rate quá lớn Quy tắc cập nhật tham số có dạng:

$$W \leftarrow W - \alpha \frac{\partial \mathcal{L}}{\partial W}.$$

Nếu learning rate α quá lớn, một bước cập nhật có thể đẩy trọng số sang vùng giá trị rất âm. Hệ quả là:

$$Wx + b \ll 0 \quad \Rightarrow \quad \text{ReLU}(Wx + b) = 0,$$

khiến neuron bị chết hàng loạt.

(ii) Bias âm lớn Bias là một hằng số được cộng trực tiếp vào tổng tuyến tính. Một bias âm đủ lớn có thể khiến toàn bộ phân phối đầu vào của ReLU bị dịch sang miền âm, kể cả khi trọng số không quá nhỏ. Trong trường hợp này, neuron sẽ luôn bị vô hiệu hoá.

Dying ReLU có luôn xảy ra không?

Không hẳn. Trong thực tế, mỗi bước tối ưu thường sử dụng một mini-batch. Miễn là *không phải toàn bộ dữ liệu trong batch* đẩy neuron vào miền âm, gradient vẫn có thể tồn tại và neuron vẫn học được. Tuy nhiên, với mạng sâu và thiết lập không cẩn thận, hiện tượng này vẫn xảy ra khá phổ biến.

Các hướng khắc phục

(i) Giảm learning rate Learning rate nhỏ hơn giúp các bước cập nhật ổn định, giảm nguy cơ "văng" trọng số sang miền âm lớn.

(ii) Sử dụng biến thể của ReLU Ý tưởng chung là tránh đoạn phẳng có đạo hàm bằng 0 trong miền âm:

- **Leaky ReLU:**
$$f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases} \quad (\alpha \ll 1)$$
- **PReLU:** giống Leaky ReLU nhưng α là tham số học được.
- **ELU, GELU:** dùng hàm trơn để duy trì gradient trong miền âm.

(iii) Khởi tạo tham số hợp lý Các phương pháp như **He initialization** giúp phân phối đầu vào phù hợp hơn với ReLU. Ngoài ra, một số nghiên cứu chỉ ra rằng khởi tạo bất đối xứng ngẫu nhiên có thể giảm khả năng rơi vào bẫy Dying ReLU.

Liên hệ với Skip Connection

Một điểm thú vị là skip connection cũng gián tiếp giúp giảm Dying ReLU. Nhờ nhánh identity, gradient có thể đi vòng qua các ReLU "bị chết", giúp các tầng trước đó vẫn được cập nhật. Đây chính là một trong những lý do ResNet có thể huấn luyện rất sâu mà không sụp đổ.

Tóm lại ReLU mạnh nhưng không hoàn hảo. Hiểu rõ Dying ReLU không chỉ giúp ta chọn đúng activation, mà còn hiểu sâu hơn *vì sao kiến trúc hiện đại như ResNet, GELU hay PReLU ra đời.*

1.4 Batch Normalization như một "liều thuốc" cho Dying ReLU

Từ phân tích ở mục trước, ta thấy rằng Dying ReLU không chỉ đến từ bản thân hàm kích hoạt, mà còn từ *phân phối của tín hiệu đưa vào ReLU*. Cụ thể, khi đầu vào $z = Wx + b$ thường xuyên nằm sâu trong miền âm, ReLU sẽ cho đầu ra bằng 0 và chặn hoàn toàn gradient.

Vấn đề cốt lõi lúc này không còn là "ReLU có tốt hay không", mà là:

Làm thế nào để giữ cho phân phối của z ổn định và nằm trong vùng hoạt động của ReLU?

Batch Normalization (BatchNorm) được đề xuất chính để giải quyết câu hỏi này.

Batch Normalization làm gì?

Ý tưởng của BatchNorm là:

1. Chuẩn hoá đầu vào của mỗi tầng theo *mini-batch*.
2. Ép phân phối của tín hiệu về dạng có trung bình gần 0 và độ lệch chuẩn gần 1.
3. Sau đó cho phép mạng học lại phép co giãn và tịnh tiến thông qua hai tham số học được.

Với một mini-batch $\{z_1, z_2, \ldots, z_m\}$, BatchNorm thực hiện:

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

trong đó μ_B và σ_B^2 là trung bình và phương sai theo batch. Sau đó áp dụng phép biến đổi học được:

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

Vì sao BatchNorm giúp giảm Dying ReLU?

BatchNorm tác động trực tiếp vào gốc rễ của vấn đề:

- Khi tín hiệu được chuẩn hoá quanh 0, xác suất rơi vào miền âm sâu của ReLU giảm đáng kể.
- ReLU sẽ có cả neuron hoạt động (đầu vào dương) và neuron bị chặn, giúp mạng duy trì tính sparse *mà không làm chết toàn bộ neuron.*
- Gradient ít bị suy giảm vì đầu vào hiếm khi bị kẹt cố định ở một phía.

Nói cách khác, BatchNorm giống như việc *liên tục kéo dữ liệu quay về vùng "an toàn" cho ReLU hoạt động.*

BatchNorm và tính bất biến tuyến tính

Liên hệ với phần chuẩn hoá trước đó, BatchNorm còn giúp mạng trở nên bền vững hơn trước các phép biến đổi tuyến tính:

$$z = aX + b.$$

Do đã loại bỏ ảnh hưởng của a và b ở mỗi mini-batch, mạng ít nhạy cảm với việc thay đổi scale hoặc shift của dữ liệu nội bộ. Điều này giúp việc huấn luyện trở nên ổn định hơn, đặc biệt trong các mạng rất sâu.

BatchNorm trong PyTorch

Trong thực tế, BatchNorm thường được đặt *trước ReLU*:

```
nn.Linear(...)
nn.BatchNorm1d(hidden_dim)
nn.ReLU()
```

Cách sắp xếp này đảm bảo rằng ReLU luôn nhận được đầu vào đã được chuẩn hoá, giảm nguy cơ Dying ReLU ngay từ đầu.

Góc nhìn tổng hợp

Có thể xem BatchNorm như một "đệm an toàn" cho ReLU:

- ReLU: tạo phi tuyến mạnh, giúp mô hình học biểu diễn phức tạp.
- BatchNorm: giữ cho phân phối ổn định, tránh việc phi tuyến bị vô hiệu hoá.

Chính sự kết hợp này đã trở thành chuẩn mực trong các kiến trúc hiện đại, từ MLP sâu cho đến CNN và Transformer.

Tóm lại Dying ReLU không phải là lỗi của ReLU, mà là lời nhắc rằng: *phi tuyến mạnh cần đi kèm với cơ chế kiểm soát phân phối.* Batch Normalization chính là một trong những cơ chế như vậy.

1.5 He Initialization (He Normalization) - khởi tạo để ReLU không chết từ đầu

Batch Normalization giúp ổn định *phân phối tín hiệu trong quá trình huấn luyện.* Tuy nhiên, vẫn còn một câu hỏi quan trọng hơn, xuất hiện *ngay từ thời điểm bắt đầu huấn luyện*:

> *Nếu các trọng số ban đầu đã khiến tín hiệu rơi sâu vào vùng âm, thì ReLU sẽ chết ngay từ bước đầu tiên. Làm sao tránh điều đó?*

Câu trả lời nằm ở **cách khởi tạo trọng số**. Đây chính là vai trò của **He Initialization**.

Vì sao không thể khởi tạo trọng số tuỳ ý?

Xét một neuron trong tầng ẩn:

$$z = \sum_{i=1}^{n} w_i x_i$$

Nếu:

- trọng số w_i quá lớn $\Rightarrow$ z có phương sai rất lớn $\Rightarrow$ gradient dễ nổ,

- trọng số w_i quá nhỏ $\Rightarrow z \approx 0 \Rightarrow$ gradient dễ biến mất,

thì mạng sẽ **rất khó học**, dù kiến trúc có đúng đến đâu.
Vấn đề cốt lõi là:

Làm sao để phương sai của tín hiệu được bảo toàn khi đi qua nhiều tầng?

ReLU làm thay đổi phân phối như thế nào?

Khác với tanh hay sigmoid, ReLU *cắt toàn bộ nửa âm* của phân phối đầu vào:

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

Điều này có một hệ quả quan trọng:

- chỉ khoảng $\sim 50\%$ neuron được kích hoạt (giả sử z đối xứng quanh 0),
- phương sai của tín hiệu sau ReLU giảm đi gần một nửa.

Nếu ta vẫn dùng các cách khởi tạo cổ điển (như Xavier), phương sai sẽ *giảm dần qua từng tầng*, và mạng sâu sẽ gần như không học được.

Ý tưởng của He Initialization

He Initialization được thiết kế **riêng cho ReLU**. Ý tưởng rất trực quan:

Vì ReLU làm mất một nửa tín hiệu, ta cần khởi tạo trọng số sao cho phương sai ban đầu lớn hơn để bù lại.

Cụ thể, với một tầng có n đầu vào, He Initialization khởi tạo:

$$w_i \sim \mathcal{N}\left(0, \frac{2}{n}\right) \quad \text{hoặc} \quad w_i \sim \mathcal{U}\left(-\sqrt{\frac{6}{n}}, \sqrt{\frac{6}{n}}\right)$$

Hệ số $\frac{2}{n}$ chính là "liều bù" cho ReLU.

He Initialization giúp giảm Dying ReLU như thế nào?

Từ góc nhìn động lực học:

- Phương sai đầu vào được giữ ổn định qua các tầng.
- Ít neuron rơi sâu vào miền âm ngay từ đầu.
- Gradient lan truyền tốt hơn trong những epoch đầu tiên.

Nói cách khác, He Initialization giúp:

ReLU có cơ hội sống sót và học được trước khi BatchNorm hay optimizer kịp can thiệp.

He Initialization trong PyTorch

Trong PyTorch, He Initialization được sử dụng rất gọn:

```
nn.Linear(in_features, out_features)
# mặc định đã dùng Kaiming (He) initialization cho ReLU
```

Hoặc chủ động hơn:

```
nn.init.kaiming_normal_(layer.weight, nonlinearity='relu')
nn.init.zeros_(layer.bias)
```

PyTorch gọi He Initialization là `Kaiming initialization` (theo tên tác giả Kaiming He).

BatchNorm và He Initialization có trùng vai trò không?

Một nhầm lẫn phổ biến là:

> *Dã dùng BatchNorm thì không cần quan tâm khởi tạo nữa.*

Thực tế:

- He Initialization **ổn định ngay từ bước đầu tiên**.
- BatchNorm **ổn định trong quá trình huấn luyện**.

Chúng **bổ sung cho nhau**, không thay thế nhau.
Tóm lại

> *ReLU không tự nhiên hoạt động tốt. Nó cần được "nuôi sống" bằng khởi tạo đúng (He Initialization) và được "chăm sóc" trong quá trình huấn luyện (Batch Normalization).*