Ghi chép bài giảng: Stanford CS231n

Lecture 4: Neural Networks and Backpropagation

Ho Hong Phuc Nguyen

9 tháng 2, 2026

Mục lục

1 Neural Networks (Mạng Nơ-ron) 3
1.1 Cấu trúc cơ bản . . . 3
1.2 Tại sao cần nhiều lớp (Deep Architecture)? . . . 3
1.3 Tại sao cần hàm phi tuyến (Non-linearity)? . . . 4
1.4 Cảm hứng sinh học và sự thận trọng . . . 4

2 Activation Functions (Hàm kích hoạt) 4
2.1 Sigmoid và Tanh (Các hàm cổ điển) . . . 5
2.2 ReLU (Rectified Linear Unit) . . . 5
2.3 Các biến thể hiện đại . . . 5
2.4 Quy tắc sử dụng . . . 5

3 Architecture Design (Thiết kế kiến trúc) 6
3.1 Neural Network Size và Regularization . . . 6

4 Backpropagation (Lan truyền ngược) 6
4.1 Tại sao cần Backpropagation? (Why Backprop?) . . . 6
4.2 Computational Graph (Đồ thị tính toán) . . . 6
4.3 Các cổng cơ bản (Gate patterns) . . . 7
4.4 Vectorized Backpropagation . . . 8

1 Neural Networks (Mạng Nơ-ron)

1.1 Cấu trúc cơ bản

Trước đây, chúng ta sử dụng hàm tuyến tính $f = Wx$. Bây giờ, để giải quyết các vấn đề phức tạp hơn mà một đường thẳng không thể phân tách, ta cần một hàm phi tuyến.

Một mạng nơ-ron 2 lớp (2-layer Neural Network) được định nghĩa như sau:

$$f = W_2 \max(0, W_1 x) \tag{1}$$

Trong đó:

- $x \in \mathbb{R}^D$: Đầu vào (input).
- $W_1 \in \mathbb{R}^{H \times D}$: Trọng số lớp thứ nhất (biến đổi input sang hidden layer).
- $\max(0, \cdot)$: Hàm kích hoạt phi tuyến (Non-linearity), ở đây là hàm ReLU.
- $W_2 \in \mathbb{R}^{C \times H}$: Trọng số lớp thứ hai (biến đổi hidden layer sang output score).

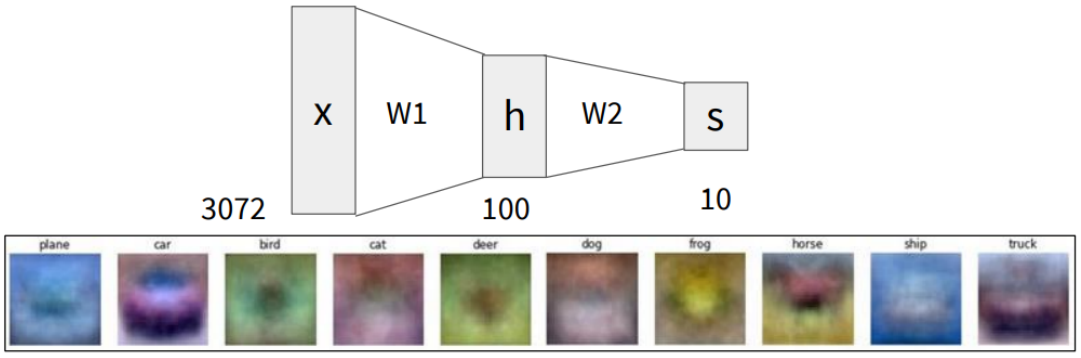
1.2 Tại sao cần nhiều lớp (Deep Architecture)?

Sự khác biệt cốt lõi nằm ở khả năng biểu diễn dữ liệu (Representation Power):

- **Giới hạn của Linear Classifier (Mạng 1 lớp):**
  - Chỉ có thể học được **một khuôn mẫu (template) duy nhất** cho mỗi lớp (class).
  - *Ví dụ:* Với bộ dữ liệu CIFAR-10, lớp "xe hơi"(Car) sẽ là trung bình cộng của tất cả các xe (xe đỏ, xe xanh, xe quay trái, xe quay phải...). Kết quả là ta nhận được một "chiếc xe ma"mờ nhạt, không rõ ràng. Nếu input khớp với mẫu trung bình này thì score cao, ngược lại thì thấp.
- **Sức mạnh của Hidden Layers (Mạng nhiều lớp):**
  - Lớp ẩn cho phép mạng học được nhiều khuôn mẫu trung gian hơn.
  - **Số lượng Template:** Thay vì bị giới hạn ở 10 khuôn mẫu (tương ứng 10 class đầu ra), nếu lớp ẩn có 100 nơ-ron, mạng có thể học tới **100 khuôn mẫu (templates)** khác nhau.
  - **Học các bộ phận (Part-based Features):** Các nơ-ron ở lớp ẩn không nhất thiết phải học toàn bộ hình ảnh con vật. Thay vào đó, chúng học các đặc trưng cục bộ như: *mắt, mũi, tai, bánh xe, vô lăng...*
  - **Tái sử dụng đặc trưng (Distributed Representation):**

    "Chó, mèo, ếch, ngựa đều có mắt."

    Một nơ-ron chuyên phát hiện "mắt"có thể được dùng chung (share) để nhận diện cả chó, mèo và ngựa. Việc kết hợp các đặc trưng rời rạc này (mắt + tai nhọn + lông = mèo) giúp mạng nơ-ron linh hoạt và mạnh mẽ hơn nhiều so với Linear Classifier.



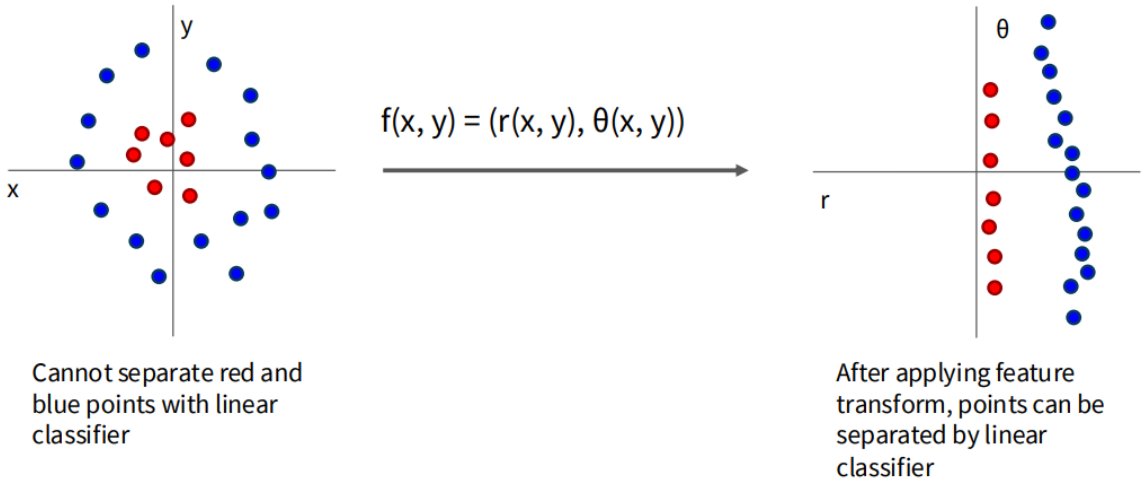
Learn 100 templates instead of 10. Share templates between classes

1.3 Tại sao cần hàm phi tuyến (Non-linearity)?

Nếu không có hàm kích hoạt phi tuyến (ví dụ bỏ hàm max), công thức trở thành:

$$f = W_2 W_1 x = W_3 x \tag{2}$$

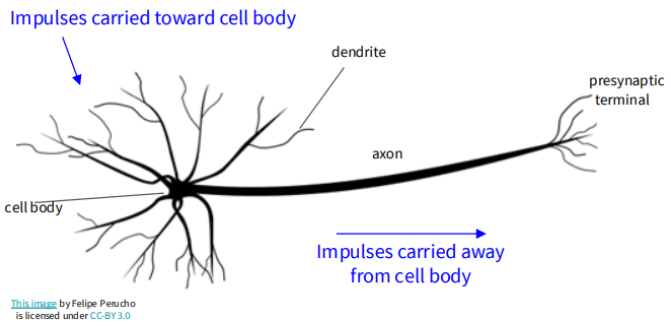
Khi đó, W_3 chỉ là một ma trận tuyến tính hợp nhất. Mạng nhiều lớp sẽ quy về mạng 1 lớp (Linear Classifier) và mất khả năng học các mẫu phức tạp. Hàm phi tuyến là yếu tố sống còn để mạng nơ-ron xấp xỉ được các hàm phức tạp ("Universal Approximation Theorem").



1.4 Cảm hứng sinh học và sự thận trọng

Mạng nơ-ron nhân tạo lấy cảm hứng từ nơ-ron sinh học:

- **Dendrites (Đuôi gai):** Nhận tín hiệu đầu vào (x).
- **Cell body (Thân tế bào):** Tổng hợp tín hiệu (tổng trọng số Wx).
- **Axon (Sợi trục):** Truyền tín hiệu đi tiếp thông qua một ngưỡng kích hoạt (Activation function).



Lưu ý: Mối liên hệ này rất lỏng lẻo. Nơ-ron sinh học phức tạp hơn nhiều với nhiều loại xung điện và cơ chế hóa học khác nhau. Không nên so sánh tuyệt đối.

2 Activation Functions (Hàm kích hoạt)

Việc chọn hàm kích hoạt nào là một siêu tham số (Hyperparameter), nhưng có các quy tắc chung dựa trên thực nghiệm.

2.1 Sigmoid và Tanh (Các hàm cổ điển)

- **Sigmoid:** $\sigma(x) = \frac{1}{1+e^{-x}}$. Nén giá trị vào khoảng $[0, 1]$.
- **Tanh:** Nén giá trị vào khoảng $[-1, 1]$.

Vấn đề: Hai hàm này hiện nay **rất ít khi được dùng** cho các lớp ẩn (hidden layers) vì hiện tượng **Vanishing Gradient** (Biến mất đạo hàm).

> *Giải thích:* Khi đầu vào x quá lớn hoặc quá nhỏ (nằm ở hai đầu "bão hòa"của hàm), đạo hàm của Sigmoid/Tanh tiến về 0. Trong quá trình Backpropagation, các gradient này được nhân liên tiếp nhau (Chain Rule), khiến gradient tổng bị triệt tiêu về 0, làm trọng số không thể cập nhật được nữa.

2.2 ReLU (Rectified Linear Unit)

$$f(x) = \max(0, x) \tag{3}$$

Đây là lựa chọn **mặc định** hiện nay.

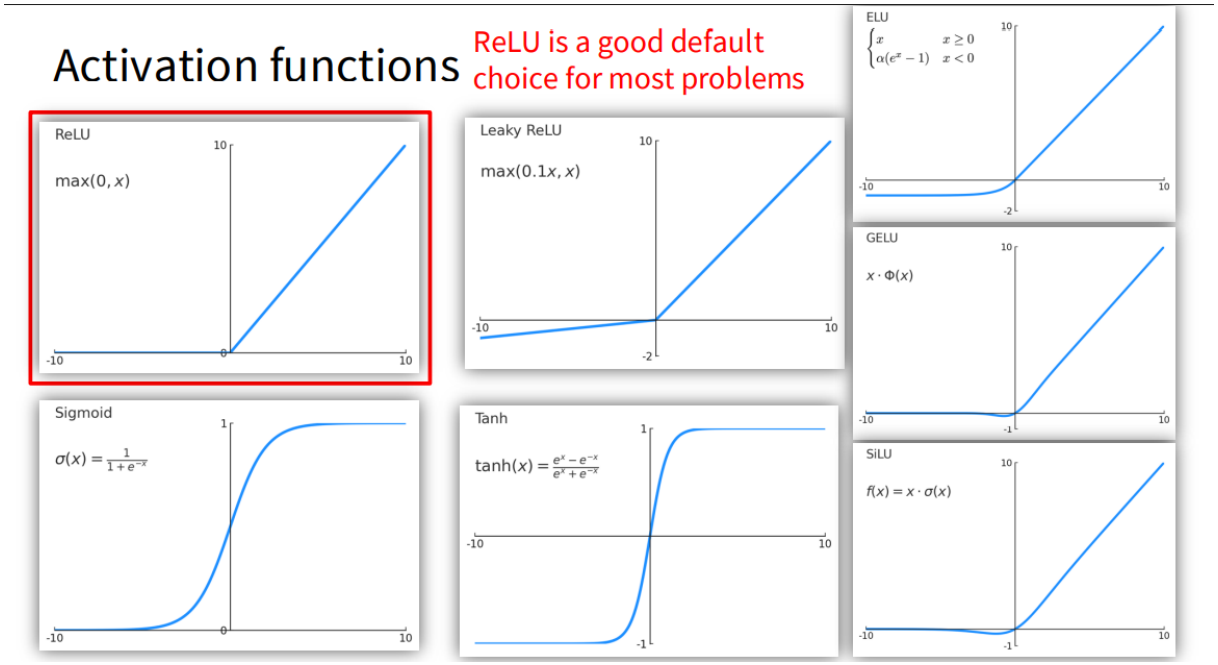
- **Ưu điểm:** Tính toán cực nhanh, hội tụ nhanh hơn Sigmoid/Tanh, không bị bão hòa ở miền dương.
- **Nhược điểm:** "Dead ReLU Nếu nơ-ron bị rơi vào miền âm và không bao giờ kích hoạt, gradient sẽ luôn bằng 0 và nơ-ron đó sẽ "chết"vĩnh viễn.

2.3 Các biến thể hiện đại

- **Leaky ReLU:** $f(x) = \max(0.01x, x)$. Cho phép một gradient nhỏ đi qua khi $x < 0$ để tránh Dead ReLU.
- **ELU, GELU:** Các biến thể mượt mà hơn, thường dùng trong các mô hình Transformer (BERT, GPT) hoặc CNN hiện đại (EfficientNet dùng SiLU/Swish).

2.4 Quy tắc sử dụng

- **Hidden Layers:** Thường dùng **cùng một loại** hàm kích hoạt cho tất cả các lớp ẩn (thường là ReLU).
- **Output Layer:** Khác biệt tùy vào bài toán (ví dụ: Softmax cho phân loại đa lớp, Sigmoid cho phân loại nhị phân, hoặc Linear cho hồi quy).



3 Architecture Design (Thiết kế kiến trúc)

3.1 Neural Network Size và Regularization

Câu hỏi: *Nên dùng mạng nhỏ để tránh Overfitting hay mạng lớn + Regularization?*

Câu trả lời của giảng viên: **Luôn ưu tiên mạng lớn kết hợp với Regularization mạnh.**

- Mạng nhỏ có ít tham số (capacity thấp), dễ bị kẹt ở các điểm cực tiểu cục bộ tồi (bad local minima).
- Mạng lớn đảm bảo khả năng học được các mẫu phức tạp. Vấn đề Overfitting của mạng lớn nên được giải quyết bằng **Regularization Strength (λ)** (L2, Dropout) thay vì giảm kích thước mạng.

Quy tắc: Kích thước mạng không nên dùng để chống Overfitting. Hãy dùng Regularization để làm điều đó.

4 Backpropagation (Lan truyền ngược)

Đây là giải thuật cốt lõi để tính gradient $\nabla_W L$.

4.1 Tại sao cần Backpropagation? (Why Backprop?)

Bạn có thể tự hỏi: *"Tại sao không tính đạo hàm một lần cho xong mà phải đi ngược từng bước?"* Câu trả lời nằm ở sự **hiệu quả tính toán** và **tính mô đun hóa**.

Có 2 cách tiếp cận ngây thơ (naive) mà chúng ta **không** dùng:

1. **Numerical Gradient (Đạo hàm số):** Ta thử cộng một lượng cực nhỏ h vào từng trọng số W và xem Loss thay đổi thế nào:
$$\frac{dL}{dW} \approx \frac{L(W+h) - L(W)}{h} \tag{4}$$
Vấn đề: Nếu mạng có 1 triệu trọng số (parameters), ta phải thực hiện phép tính này 1 triệu lần (mỗi lần lại phải chạy forward pass lại từ đầu). Quá tốn kém và chậm chạp!
2. **Symbolic Differentiation (Đạo hàm công thức tổng):** Ta viết ra một phương trình toán học khổng lồ đại diện cho toàn bộ mạng nơ-ron từ đầu đến cuối, sau đó dùng các quy tắc đạo hàm để tìm công thức của Gradient.
Vấn đề: Với các mạng sâu (Deep Networks), biểu thức này sẽ cực kỳ phức tạp (nested functions), dễ sai sót và không thể tối ưu hóa bộ nhớ.

Giải pháp Backpropagation: Backpropagation sử dụng **Đồ thị tính toán (Computational Graph)** để chia nhỏ bài toán khổng lồ thành các bài toán con cực nhỏ và đơn giản.

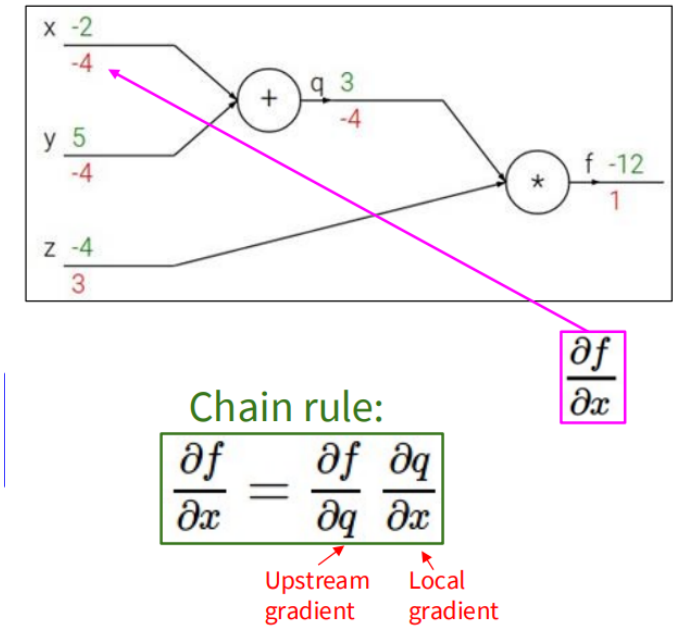
- **Tính cục bộ (Local Process):** Mỗi cổng (Gate) trong đồ thị **không cần biết** toàn bộ mạng nơ-ron phức tạp thế nào. Nó chỉ cần biết:
  1. Đầu vào của chính nó (x, y).
  2. Đạo hàm từ phía sau truyền tới (*Upstream Gradient*).
- **Hiệu quả (Efficiency):** Chỉ cần **1 lần Forward pass** (tính Loss) và **1 lần Backward pass** (tính Gradient) là ta có được đạo hàm cho tất cả hàng triệu trọng số cùng lúc.

4.2 Computational Graph (Đồ thị tính toán)

Chúng ta biểu diễn hàm mất mát dưới dạng đồ thị các phép toán. Backpropagation hoạt động dựa trên quy tắc chuỗi (**Chain Rule**) đi từ cuối đồ thị ngược về đầu.

Với mỗi nút (node) trong đồ thị:

$$\text{Downstream Gradient} = \text{Local Gradient} \times \text{Upstream Gradient} \tag{5}$$



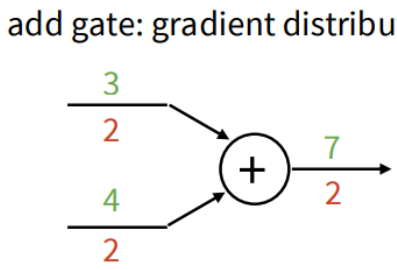
4.3 Các cổng cơ bản (Gate patterns)

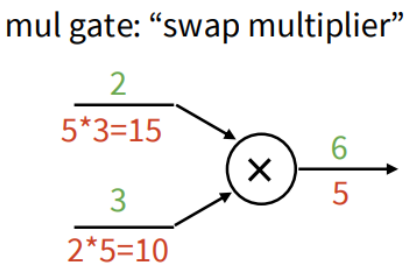
Giả sử ta có x, y là đầu vào và z là đầu ra của cổng. Gradient từ phía sau truyền về là $\frac{\partial L}{\partial z}$.

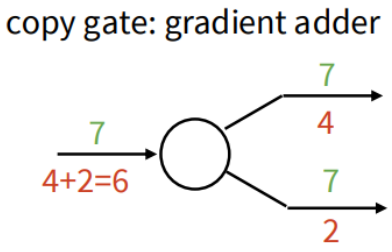
1. **Add Gate (Cổng cộng):** $z = x + y$.
   - Đạo hàm cục bộ: $\frac{\partial z}{\partial x} = 1, \frac{\partial z}{\partial y} = 1$.
   - **Quy tắc Backprop:** Gradient tổng $\frac{\partial L}{\partial z}$ được phân phối đều cho cả hai nhánh.
   - **Vai trò:** *Gradient Distributor* (Bộ phân phối Gradient).
2. **Copy Gate (Cổng sao chép - Phân nhánh):** $z_1 = x, z_2 = x, \ldots$
   - Xảy ra khi một biến x đi vào nhiều node khác nhau.
   - **Quy tắc Backprop:** Gradient tại x sẽ bằng **tổng** các gradient từ các nhánh con trả về:

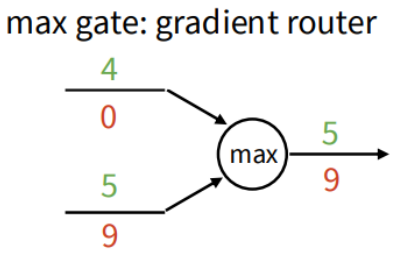
$$\frac{\partial L}{\partial x} = \sum \frac{\partial L}{\partial z_i} \tag{6}$$

   - **Vai trò:** *Gradient Adder* (Bộ cộng Gradient).
3. **Mul Gate (Cổng nhân):** $z = x \cdot y$.
   - Đạo hàm cục bộ: $\frac{\partial z}{\partial x} = y, \frac{\partial z}{\partial y} = x$.
   - **Quy tắc Backprop:** Gradient truyền về x là gradient tổng nhân với giá trị của y (và ngược lại).
   - **Vai trò:** *Swap Multiplier* (Bộ nhân hoán đổi).
4. **Max Gate (Cổng cực đại):** $z = \max(x, y)$.
   - **Quy tắc Backprop:** Gradient chỉ được truyền về nhánh có giá trị lớn hơn (nhánh thắng cuộc), nhánh còn lại nhận gradient 0.
   - **Vai trò:** *Gradient Router* (Bộ định tuyến Gradient).









4.4 Vectorized Backpropagation

Trong thực tế, x, W là các vector hoặc ma trận.

- Đạo hàm của một vector hàm theo vector biến input sẽ tạo ra **Jacobian Matrix**.
- Tuy nhiên, ta không bao giờ tính toán trực tiếp Jacobian Matrix (vì quá lớn, ví dụ 4096 input $\rightarrow$ 4096 output sẽ tạo ma trận 4096^2, tốn hàng trăm GB bộ nhớ).
- Thay vào đó, ta dùng tính chất của phép toán để tính gradient trực tiếp (thường là phép nhân ma trận chuyển vị).
- **Kiểm tra kích thước (Sanity Check):** Gradient của một biến $\nabla_x L$ **luôn luôn** có cùng kích thước (shape) với biến đó x.