

Hiểu và Xây dựng CNN gốc với góc nhìn từ bộ lọc Kernel

Nhóm AIO_TimeSeries

Ngày 18 tháng 11 năm 2025

Bài viết này giải mã Convolutional Neural Networks (CNN) từ góc độ tư duy bộ lọc (filter perspective) thay vì chỉ liệt kê công thức toán học. mình sẽ đi từ hạn chế của mạng nơ-ron truyền thống đến cách CNN "nhìn" thế giới, cơ chế lan truyền ngược phức tạp và các kỹ thuật nâng cao như 1x1 Convolution.

Phần 1: Tại sao Neural Network truyền thống (MLP) "bó tay" với hình ảnh?
Vấn đề bùng nổ tham số và giới hạn của việc xử lý ảnh dạng phẳng.

Phần 2: Tư duy Bộ lọc (Filter Perspective) & Cơ chế cơ bản
Cách máy tính "nhìn" đặc trưng thông qua Kernel, Stride và Padding.

Phần 3: Kiến trúc CNN - Từ "Phân rã đặc trưng" đến "Hội đồng bỏ phiếu"
Quy trình trích xuất, lọc nhiễu (ReLU) và tổng hợp thông tin (Pooling).

Phần 4: CNN Backpropagation & Sự thật về phép xoay Kernel
Tại sao CNN Backpropagation cần xoay kernel 180 độ .

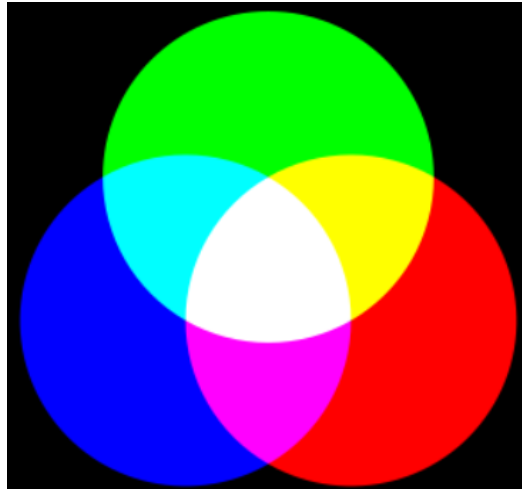
Phần 5: CNN Nâng cao - 1x1 Convolution & Phân biệt chiều không gian
1x1 Conv cho vấn đề bùng nổ chiều sâu.

Phần 1: Tại sao Neural Network truyền thống (MLP) "bó tay" với hình ảnh?

Trước khi nói về giải pháp, hãy nhìn vào vấn đề. Tại sao mình không thể cứ thế ném một bức ảnh vào mạng nơ-ron đa tầng (MLP) thông thường? Để hiểu điều này, ta cần "mổ xẻ" cách máy tính lưu trữ một bức ảnh.

1.1 Bản chất của màu sắc (RGB Color)

Trong thị giác máy tính, màu sắc không phải là khái niệm trừu tượng mà là các con số cụ thể được định nghĩa bởi hệ màu **RGB** (Red - Green - Blue). Mỗi kênh màu này đại diện cho một sắc độ riêng biệt với giá trị chạy từ 0 đến 255 (tương ứng với 256 mức độ khác nhau). Bằng cách phối hợp cường độ sáng của 3 kênh này lại, máy tính có thể tái tạo hàng triệu màu sắc khác nhau mà mắt người nhìn thấy được.



1.2 Biểu diễn ảnh dưới dạng Ma trận (Matrix Representation)

Một bức ảnh kỹ thuật số thực chất là một lưới các điểm ảnh (pixels). Ví dụ, một bức ảnh kích thước 800×600 sẽ được cấu thành từ 800 hàng và 600 cột.

$$\begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,800} \\ w_{2,1} & w_{2,2} & \dots & w_{2,800} \\ \dots & \dots & \dots & \dots \\ w_{600,1} & w_{600,2} & \dots & w_{600,800} \end{bmatrix}$$

(a) Ma trận tổng quát $W_{i,j}$

$$\begin{bmatrix} (100, 100, 50) & (101, 112, 3) & (131, 20, 80) \\ (150, 210, 130) & (10, 120, 130) & (111, 120, 130) \\ (10, 260, 30) & (200, 20, 30) & (100, 20, 3) \end{bmatrix}$$

(b) Giá trị pixel là bộ 3 số (R,G,B)

Tại mỗi vị trí tọa độ (i, j) , giá trị pixel w_{ij} không tồn tại dưới dạng một số đơn lẻ mà là một bộ ba giá trị (r_{ij}, g_{ij}, b_{ij}) , ví dụ như $(0, 233, 256)$ để tạo ra màu xanh ngọc. Để tối ưu hóa cho việc lưu trữ và xử lý tính toán, máy tính thường không gộp chung mà tách riêng các giá trị này thành 3 ma trận độc lập tương ứng với 3 kênh màu Đỏ, Xanh lá và Xanh dương.

$$\begin{bmatrix} 100 & 101 & 131 \\ 150 & 10 & 111 \\ 10 & 200 & 100 \end{bmatrix}, \begin{bmatrix} 100 & 112 & 20 \\ 210 & 120 & 120 \\ 260 & 20 & 20 \end{bmatrix}, \begin{bmatrix} 50 & 3 & 80 \\ 130 & 130 & 130 \\ 30 & 30 & 3 \end{bmatrix}$$

R G B

(c) Tách một ảnh màu thành 3 ma trận R, G, B riêng biệt

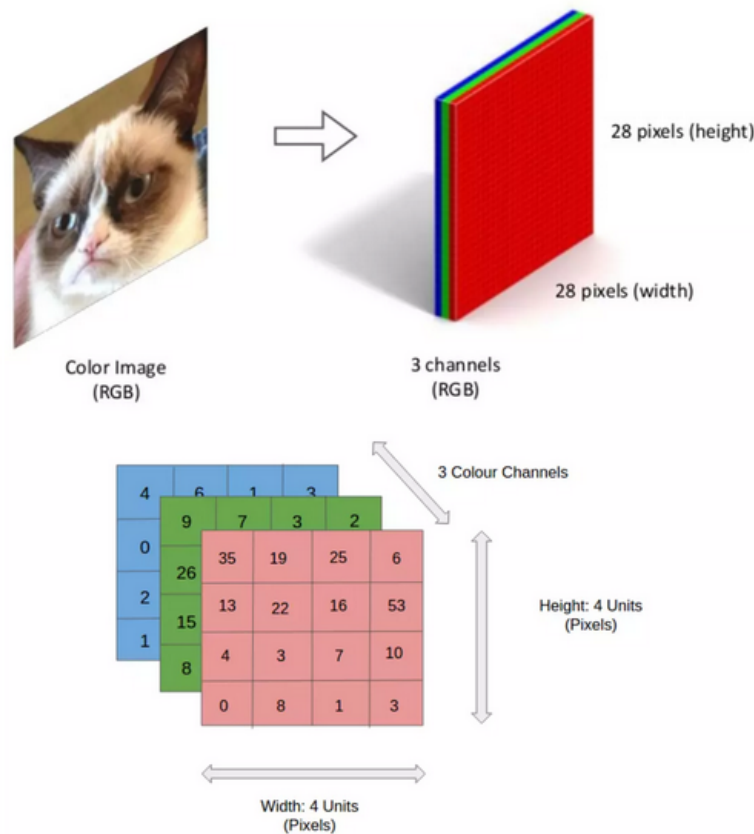
$$\begin{bmatrix} (r_{1,1}, g_{1,1}, b_{1,1}) & (r_{1,2}, g_{1,2}, b_{1,2}) & \dots & (r_{1,800}, g_{1,800}, b_{1,800}) \\ (r_{2,1}, g_{2,1}, b_{2,1}) & (r_{2,2}, g_{2,2}, b_{2,2}) & \dots & (r_{2,800}, g_{2,800}, b_{2,800}) \\ \dots & \dots & \dots & \dots \\ (r_{600,1}, g_{600,1}, b_{600,1}) & (r_{600,2}, g_{600,2}, b_{600,2}) & \dots & (r_{600,800}, g_{600,800}, b_{600,800}) \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} r_{1,1} & r_{1,2} & \dots & r_{1,800} \\ r_{2,1} & r_{2,2} & \dots & r_{2,800} \\ \dots & \dots & \dots & \dots \\ r_{600,1} & r_{600,2} & \dots & r_{600,800} \end{bmatrix}, \begin{bmatrix} g_{1,1} & g_{1,2} & \dots & g_{1,800} \\ g_{2,1} & g_{2,2} & \dots & g_{2,800} \\ \dots & \dots & \dots & \dots \\ g_{600,1} & g_{600,2} & \dots & g_{600,800} \end{bmatrix}, \begin{bmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,800} \\ b_{2,1} & b_{2,2} & \dots & b_{2,800} \\ \dots & \dots & \dots & \dots \\ b_{600,1} & b_{600,2} & \dots & b_{600,800} \end{bmatrix},$$

1.3 Từ Ma trận đến Tensor

Khái niệm **Tensor** xuất hiện để tổng quát hóa các cấu trúc dữ liệu này theo số chiều. Trong khi Vector là Tensor 1D và Ma trận là Tensor 2D (như ảnh xám chỉ cần một ma trận duy nhất để biểu diễn độ sáng từ đen sang trắng), thì ảnh màu lại phức tạp hơn. Vì ảnh màu được tạo thành từ việc chồng 3 ma trận R, G, B lên nhau (collapse on top of each other), nó được định nghĩa là một **Tensor 3D** với kích thước $Height \times Width \times Depth$ (trong đó $Depth = 3$).

color image is 3rd-order tensor



Việc hiểu đúng chiều sâu (Depth) của Tensor là mấu chốt để hiểu cách CNN vận hành sau này.

1.4 Sự bùng nổ tham số (The Parameter Explosion)

Chính cấu trúc Tensor 3D này là nguyên nhân khiến mạng MLP truyền thống "đầu hàng". Chỉ thử làm một phép tính với bức ảnh rất nhỏ kích thước 64×64 : tổng số giá trị đầu vào sẽ là $64 \times 64 \times 3 = 12,288$. Nếu ta kết nối đầu vào này vào một lớp ẩn chỉ gồm 1,000 nơ-ron theo cách kết nối toàn bộ (fully connected) của MLP, số lượng trọng số cần học sẽ lên tới hơn **12 triệu** tham số ($12,288 \times 1,000$). Khối lượng tính toán khổng lồ này không chỉ làm chậm hệ thống mà còn dẫn đến hiện tượng Overfitting nghiêm trọng, buộc mình phải tìm đến giải pháp thông minh hơn là CNN để xử lý cấu trúc Tensor một cách hiệu quả.

Phần 2: Tư duy Bộ lọc (Filter Perspective) & Cơ chế cơ bản

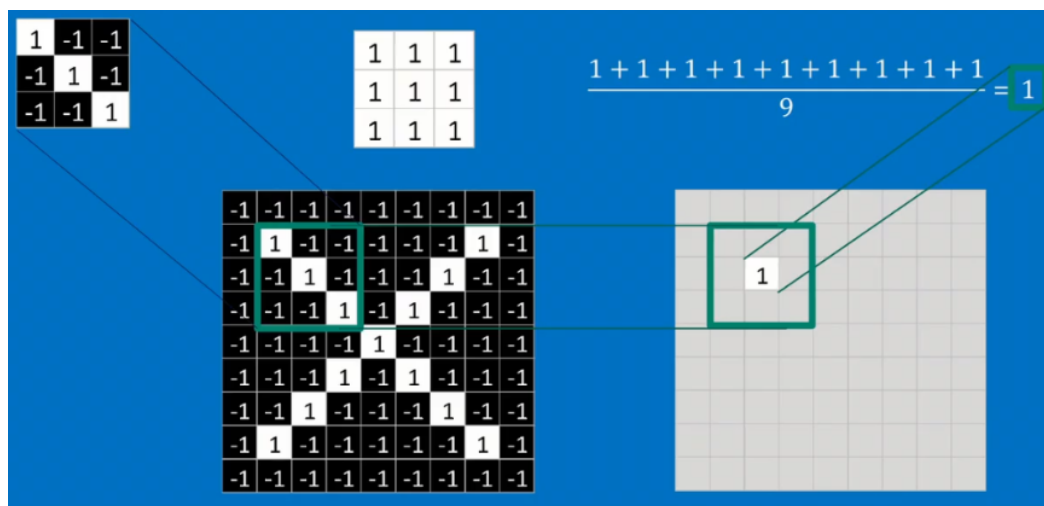
Đây là phần trọng tâm để hiểu CNN từ gốc rễ. Thay vì tư duy theo kiểu "kết nối tất cả mọi thứ" như mạng MLP truyền thống, CNN tiếp cận hình ảnh theo cách con người quan sát: tìm kiếm các đặc trưng (features).

2.1 Kernel/Filter là gì? - Chiếc "đèn pin" soi mẫu

Trong thế giới của CNN, để máy tính hiểu được một bức ảnh, nó cần biết được những đặc trưng nào làm cho bức ảnh đó trở nên độc nhất. Thay vì phân tích từng điểm ảnh riêng lẻ, mình sử dụng **Kernel** (hay Filter) - một ma trận nhỏ (thường là 3×3 hoặc 5×5). Kernel hoạt động như một "cửa sổ trượt", di chuyển quét qua toàn bộ bức ảnh để tìm kiếm sự tồn tại của các mẫu cụ thể, ví dụ như một đường cong, một cạnh dọc, hay một đường chéo. Để hiểu rõ, mình đi vào 1 ví dụ luôn nhé.

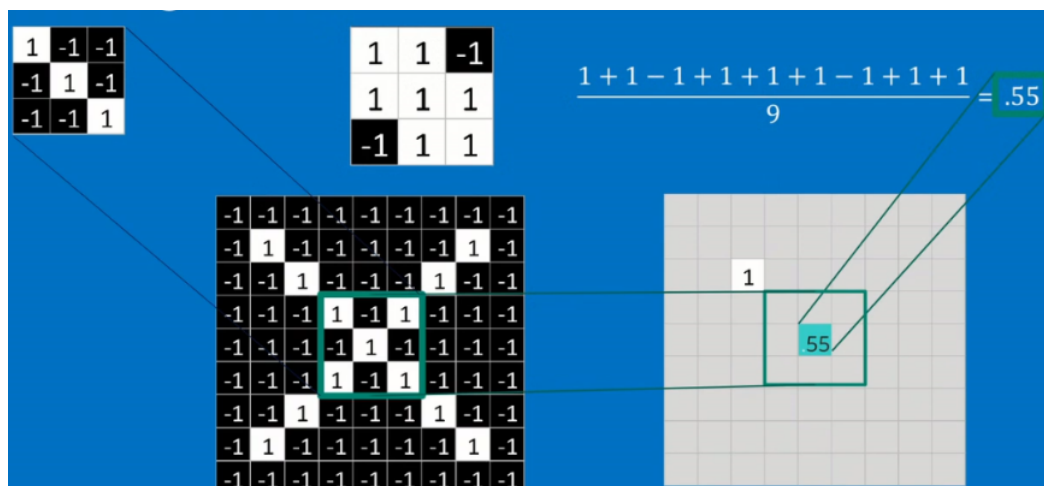
2.2 Cơ chế khớp mẫu (Feature Matching) - Bí mật của phép tích chập

Hãy hình dung mình muốn dạy máy tính nhận diện chữ "X". Chữ "X" được cấu tạo bởi hai đường chéo bắt chéo nhau. Một Kernel được thiết kế để tìm "đường chéo trái" sẽ có các giá trị dương (ví dụ: 1) nằm trên đường chéo đó và các giá trị âm (ví dụ: -1) ở những vị trí còn lại. (Note: ma trận trắng đại diện cho kết quả sau khi áp dụng kernel)



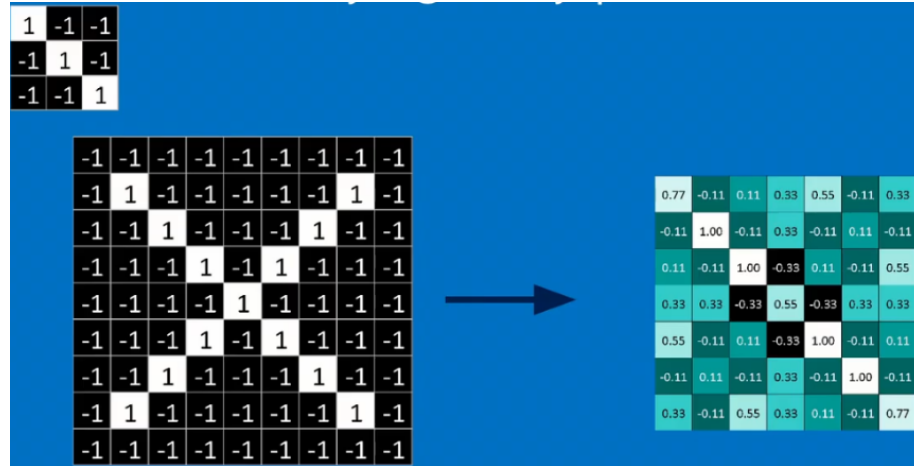
Đây là phần trọng tâm để hiểu CNN từ gốc rễ. Thay vì tư duy theo kiểu "kết nối tất cả mọi thứ" như mạng MLP truyền thống, CNN tiếp cận hình ảnh theo cách con người quan sát: tìm kiếm các đặc trưng (features).

Quá trình "Filtering" (lọc) diễn ra như sau:



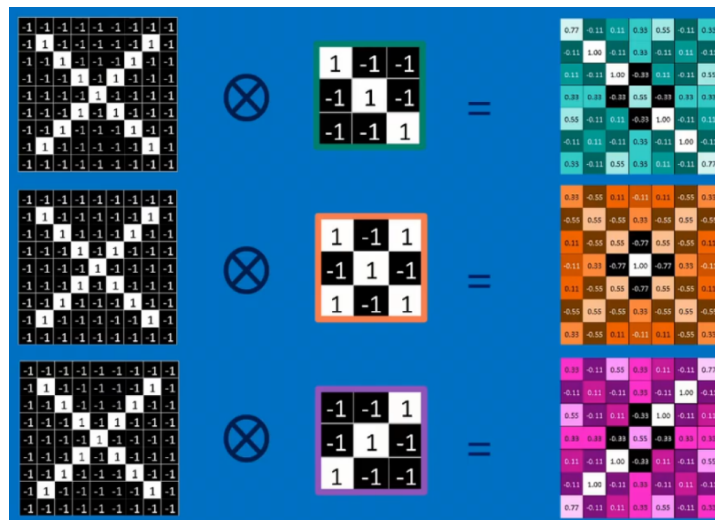
1. **Line up:** Đặt Kernel chồng lên một vùng ảnh cùng kích thước.
2. **Multiply:** Nhân từng giá trị pixel của ảnh với giá trị tương ứng trong Kernel (Element-wise multiplication).
3. **Add:** Cộng tổng tất cả các kết quả lại và chia trung bình.

Kết quả:



- Nếu vùng ảnh bên dưới thực sự có hình đường chéo (khớp với Kernel), phép nhân sẽ tạo ra các số dương lớn → Kết quả tổng rất cao (ví dụ: 1.0 hoặc 100%). Ta nói pattern đó đã được "kích hoạt" (activated).
- Nếu vùng ảnh không khớp (ví dụ: Kernel đường chéo nhưng đặt lên vùng ảnh đường thẳng đứng), các phép nhân dương và âm sẽ triệt tiêu lẫn nhau → Kết quả tổng xấp xỉ 0 hoặc âm.
- Ví dụ tương tự đối với ảnh 3D.

Minh họa các loại Filter khác nhau trích xuất các kiểu đặc trưng khác nhau



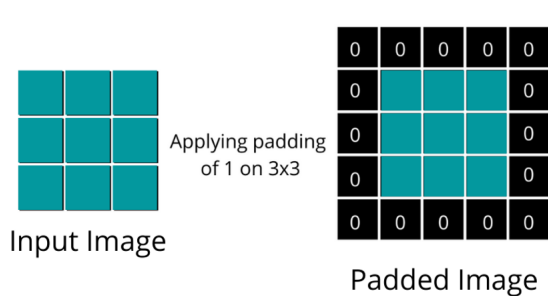
2.3 Các phép toán kiểm soát Bộ lọc (Convolution Operation)

Stride (Bước nhảy - ô vuông kernel di chuyển bao nhiêu bước mỗi lượt): Đây là khoảng cách mà Kernel di chuyển sau mỗi lần tính toán. Nếu $Stride = 1$, Kernel nhích từng pixel một, giữ độ chi tiết cao nhất. Nhưng nếu ta tăng $Stride > 1$ (ví dụ: $Stride = 2$), Kernel sẽ "nhảy cóc" qua các pixel. Điều này có tác dụng giảm kích thước dữ liệu đầu ra (Downsampling) ngay lập tức mà không cần lớp Pooling, giống như việc ta lướt nhanh qua một văn bản để nắm ý chính thay vì đọc từng chữ.

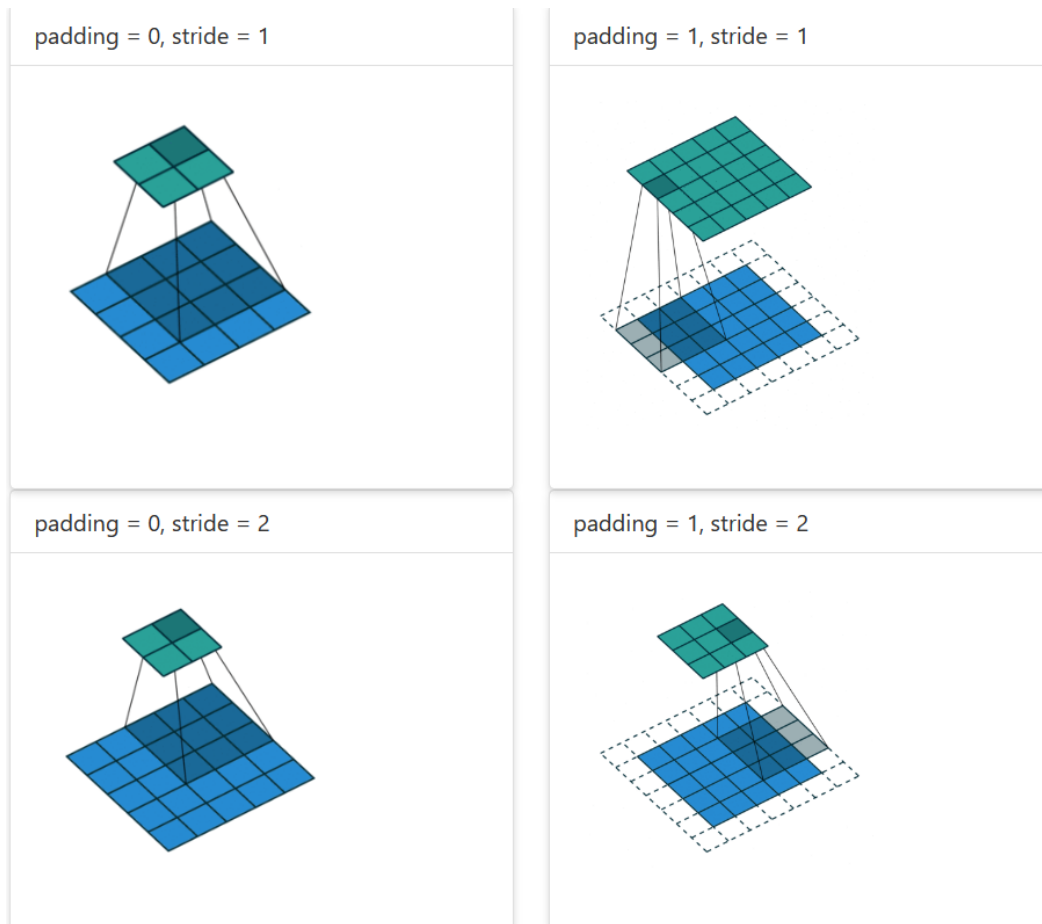
0	0	0	0	0	0	0
0	1	1	1	0	0	0
0	0	1	1	1	0	0
0	0	0	1	1	1	0
0	0	0	1	1	0	0
0	0	1	1	0	0	0
0	0	0	0	0	0	0

Figure 2: Với phần in đậm là tâm của kernel, với $stride = 2$, kernel sẽ đi 2 bước tính từ tâm. Nếu kernel là 2×2 thì lấy pixel trên cùng bên trái làm tâm

Padding (Lề - Pixel được bọc thêm bên ngoài ảnh): Khi Kernel trượt ở rìa bức ảnh, nó thường bị thiếu hụt dữ liệu (không đủ kích thước 3×3 để nhân). Điều này dẫn đến hai vấn đề: ảnh bị thu nhỏ lại sau mỗi lớp tích chập và thông tin ở rìa ảnh bị mất mát. Giải pháp là **Zero-Padding**: thêm một viền các số 0 bao quanh ảnh gốc. Lớp "đệm" này cho phép Kernel đặt tâm ngay tại pixel ngoài cùng, giúp giữ nguyên kích thước không gian ($Height \times Width$) của Feature Map đầu ra.



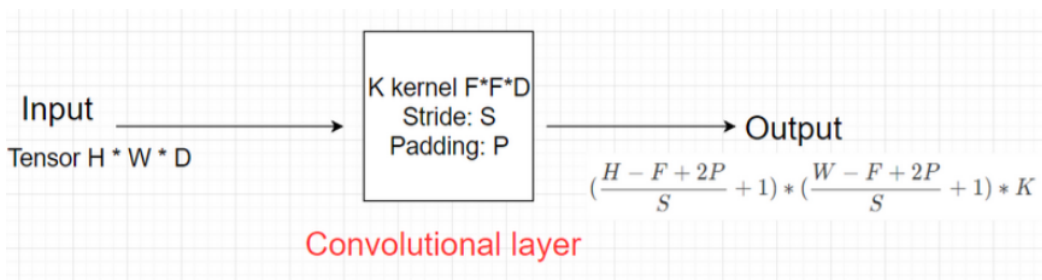
Minh họa Padding và Stride: Note: kích thước ảnh hoặc feature map (i.e. ảnh sau khi áp dụng convolution) phải lớn tương ứng để áp dụng stride.



Công thức kích thước đầu ra: Để thiết kế kiến trúc mạng chính xác, ta cần tính toán được kích thước của ma trận sau khi qua lớp Conv:

$$Output = \frac{Input - Filter + 2 \times Padding}{Stride} + 1$$

Trong đó: *Input* là kích thước ảnh đầu vào, *Filter* là kích thước Kernel, *Padding* là số lớp viền thêm vào, và *Stride* là bước nhảy.

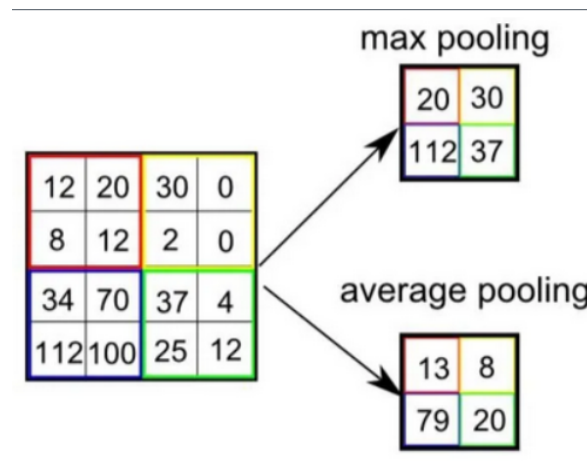


Việc trượt Kernel qua ảnh không phải lúc nào cũng tùy ý, nó được kiểm soát bởi các tham số toán học nghiêm ngặt để định hình dữ liệu đầu ra.

Pooling (Gộp đặc trưng):

- **Max Pooling:** Chỉ giữ lại giá trị lớn nhất trong vùng (đặc trưng nổi bật nhất), giúp nén ảnh và giảm tải tính toán nhưng vẫn giữ được thông tin cốt lõi.

- **Average Pooling:** Lấy giá trị trung bình của toàn bộ các pixel trong vùng, nhằm tổng hợp thông tin một cách “mềm”, phản ánh mức độ hiện diện trung bình của đặc trưng thay vì chỉ tập trung vào điểm mạnh nhất.



Phần 3: Kiến trúc CNN - Từ "Phân rã đặc trưng" đến "Hội đồng bỏ phiếu"

Nếu như Kernel là công cụ để tìm kiếm, thì Kiến trúc CNN chính là chiến lược để sử dụng những tìm kiếm đó. Bản chất của CNN không phải là nhìn thấy ngay con mèo hay chiếc xe hơi, mà là một quá trình **phân rã và tái tạo**.

3.1 Bản chất của CNN: Chia để trị

Thay vì cố gắng nhận diện toàn bộ vật thể ngay lập tức, nó chia nhỏ bài toán thành các tầng đặc trưng (Feature Hierarchy) từ đơn giản đến phức tạp:

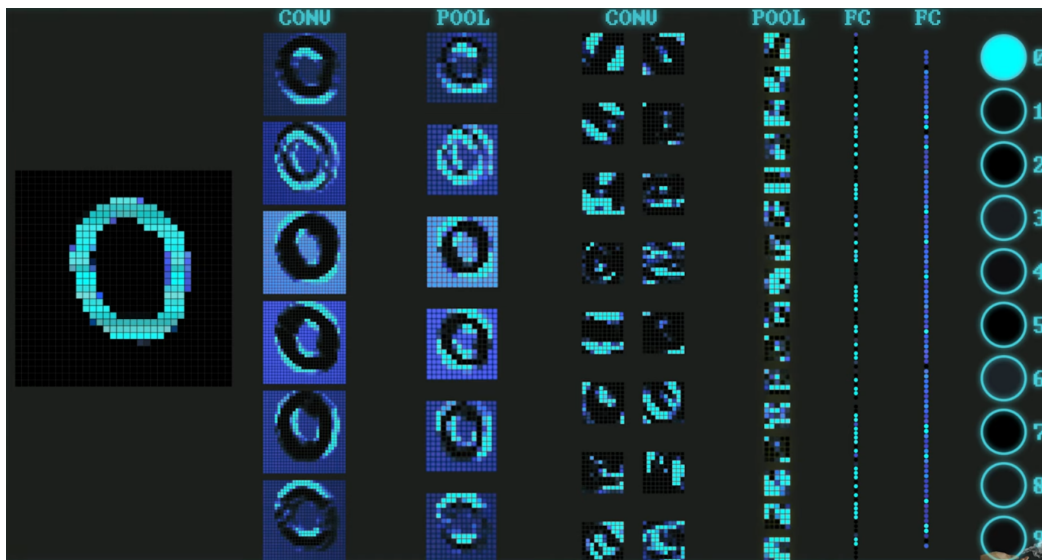


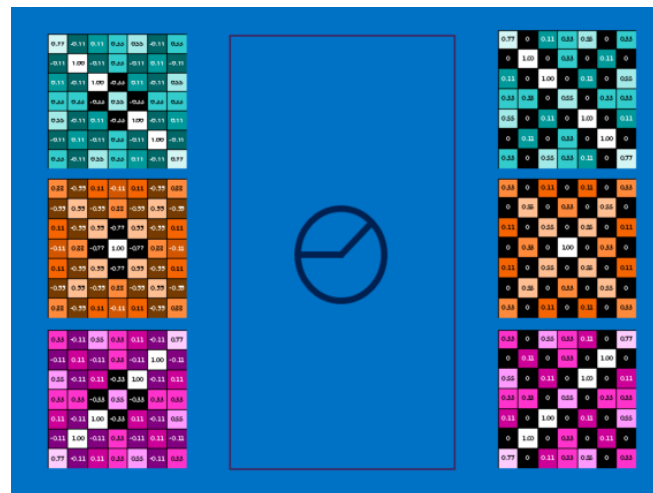
Figure 3: Minh họa sự phân cấp đặc trưng trong CNN: Từ các cạnh đơn giản (trái) đến các chi tiết vật thể phức tạp (phải).

- **Tầng thấp (Low-level features):** Ở các lớp đầu tiên, mạng chỉ nhìn thấy các chi tiết sơ khai nhất như **cạnh**, **đường biên**, **màu sắc**. Trên ảnh minh họa, chúng thường trông giống như các vạch kẻ sọc hoặc nhiễu.
- **Tầng giữa (Mid-level features):** Các lớp tiếp theo ghép nối các cạnh này lại thành các **họa tiết**, **hình thù cơ bản** như đường cong, góc vuông, hình tròn (ví dụ: mắt, bánh xe, hoặc một phần của con số).
- **Tầng cao (High-level features):** Lớp cuối cùng tổng hợp chúng thành các **bộ phận hoàn chỉnh** để nhận diện vật thể.

Mỗi lớp đóng vai trò như một bộ lọc tinh vi hơn lớp trước đó, biến đổi các pixel vô nghĩa thành các đặc trưng có ý nghĩa phân loại.

3.2 ReLU (Rectified Linear Unit) - Người gác cổng của các lá phiếu

Sau mỗi lần Kernel quét qua ảnh (phép tích chập), chúng ta thu được một bản đồ các giá trị. Có nơi giá trị rất dương (khớp mẫu), có nơi giá trị âm (ngược mẫu). Lúc này, **ReLU** xuất hiện với vai trò cực kỳ quan trọng nhưng thường bị hiểu nhầm là chỉ để "làm toán".



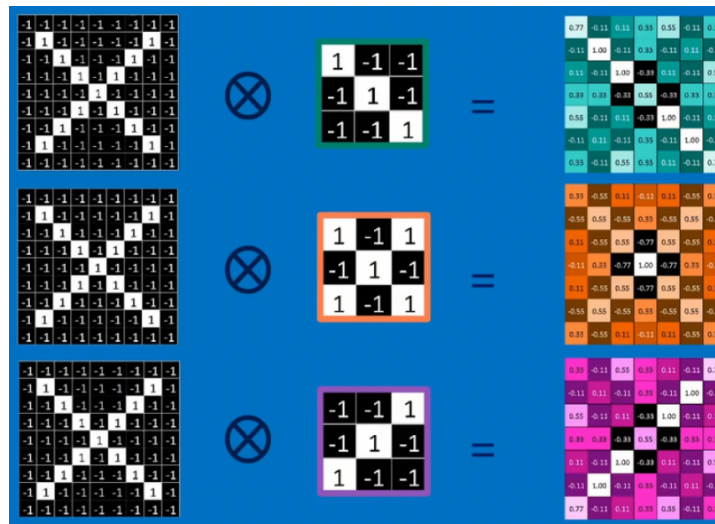
Theo góc nhìn bản chất, ReLU giúp mô hình trả lời câu hỏi: **"Đặc trưng này có xuất hiện hay không?"**.

- Nếu kết quả là số dương: "Có, tôi tìm thấy một cái cạnh ở đây!". Chúng ta giữ nguyên giá trị đó.
- Nếu kết quả là số âm: "Không, hoặc nó ngược lại với cái tôi tìm". ReLU sẽ biến nó thành số 0.

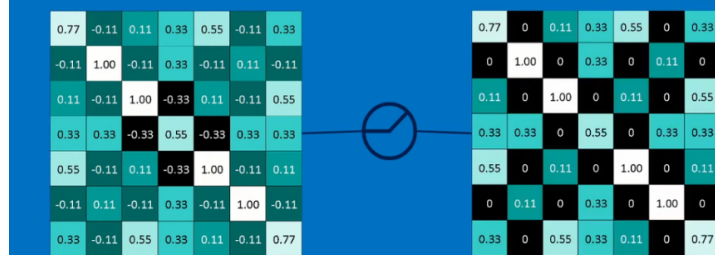
Tại sao phải loại bỏ số âm? Trong cơ chế bỏ phiếu (voting), chúng ta chỉ quan tâm đến sự hiện diện của các bằng chứng (evidence). Việc giữ lại các giá trị âm (nhiều/không khớp) sẽ làm loãng thông tin khi mạng cố gắng tổng hợp các đặc trưng lại với nhau. ReLU giúp thanh lọc tín hiệu, đảm bảo rằng các lớp sau chỉ nhận được những "bằng chứng" chắc chắn nhất.

3.3 Receptive Field - Tầm nhìn mở rộng

Khi chúng ta xếp chồng (stack) các lớp Convolution và Pooling lên nhau, một hiện tượng thú vị xảy ra: **Receptive Field** (Vùng cảm nhận) của nơ-ron ngày càng lớn.



Rectified Linear Units (ReLUs)



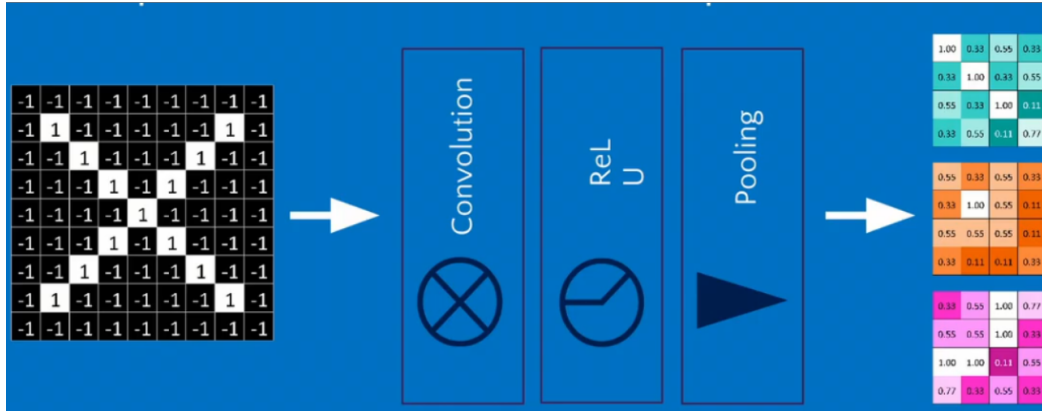
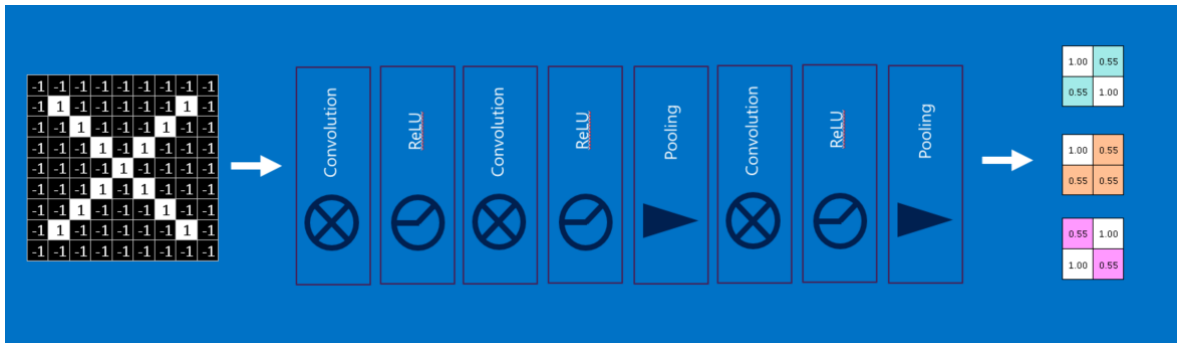


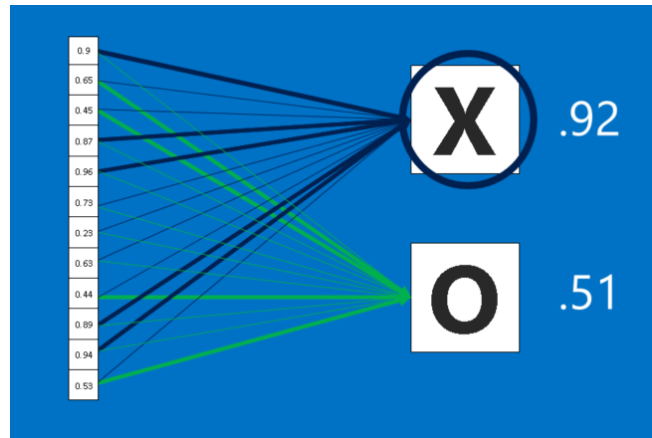
Figure 4: Một điểm pixel ở lớp Conv 1 chỉ đại diện cho vùng 3×3 của ảnh gốc.



Nhưng một điểm pixel ở lớp Conv 3 (sau khi qua Pooling) lại đại diện cho vùng 12×12 hoặc lớn hơn của ảnh gốc. Chính nhờ cơ chế này, các lớp sâu (Deep Layers) có thể "nhìn" thấy tổng thể bức ảnh để đưa ra quyết định, dù chúng được xây dựng từ những bộ lọc rất nhỏ ban đầu.

3.4 Góc nhìn "Hội đồng bỏ phiếu" (A List of Votes)

Cuối cùng, sau khi đi qua toàn bộ quy trình: Tích chập (tìm đặc trưng) $\rightarrow$ ReLU (lọc bằng chứng) $\rightarrow$ Pooling (nén thông tin), dữ liệu từ dạng hình ảnh 3D sẽ được dẹt phẳng (Flatten) thành một danh sách dài các giá trị đặc trưng cao cấp. Lúc này, lớp Fully Connected (FC) đóng vai trò như một "Hội đồng bỏ phiếu", nơi mỗi đặc trưng trở thành một "cử tri" đưa ra quyết định cuối cùng. Ví dụ: "Tôi thấy bánh xe (giá trị cao) + Tôi thấy cửa kính (giá trị cao) $\rightarrow$ Tôi bỏ phiếu cho lớp: Ô TÔ".



Một câu hỏi quan trọng thường bị bỏ qua: Tại sao ở các lớp ẩn này, ta lại ưu tiên dùng ReLU thay vì hàm Sigmoid truyền thống? Câu trả lời nằm ở khả năng "học ngược" của mạng. Hàm Sigmoid nén mọi giá trị vào khoảng (0, 1), điều này vô tình khiến tín hiệu đạo hàm bị nhỏ dần qua từng lớp, dẫn đến vấn đề **Biến mất đạo hàm (Gradient Vanishing)**, làm mạng "quên" mất các đặc trưng đầu vào. Ngược lại, ReLU hoạt động như một "cổng dẫn truyền thẳng": với các giá trị dương, đạo hàm bằng 1. Điều này không chỉ khắc phục triệt để vấn đề biến mất đạo hàm mà còn giúp tín hiệu **truy ngược (flip backward)** một cách nguyên vẹn từ lớp cuối cùng về các lớp đầu. Nhờ đó, MLP có thể học và điều chỉnh các mẫu (patterns) theo chiều ngược lại một cách hiệu quả, đảm bảo rằng mối liên kết giữa "nguyên nhân" (đặc trưng ảnh) và "kết quả" (nhãn phân loại) luôn được duy trì chặt chẽ trong suốt quá trình huấn luyện.

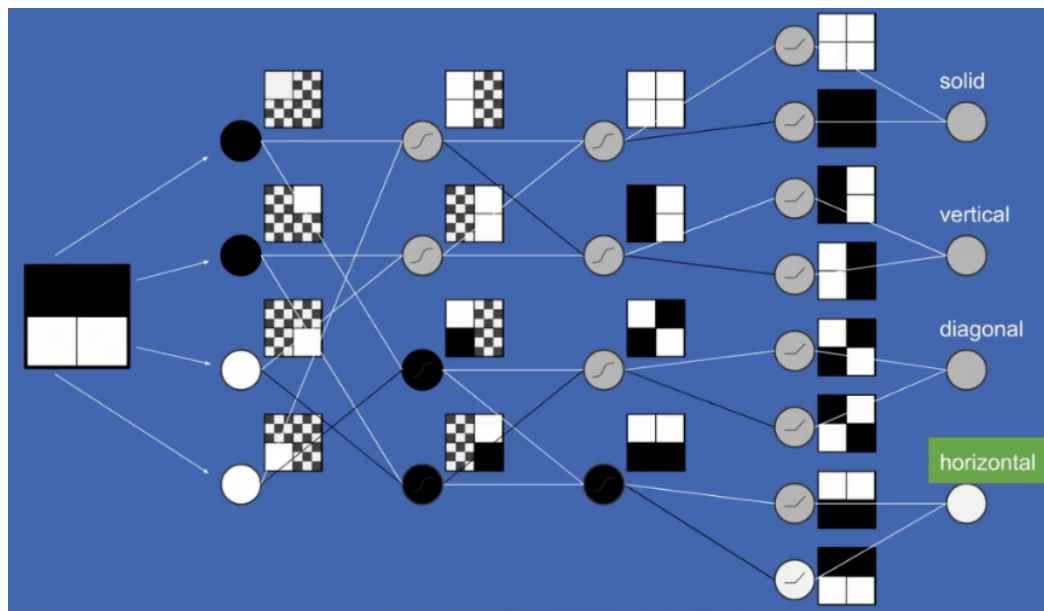


Figure 5: Cơ chế bỏ phiếu của lớp Fully Connected: Các đặc trưng được tổng hợp để đưa ra xác suất phân loại.

Phần 4: CNN Backpropagation & Sự thật về phép xoay Kernel

Đây là phần "khó nhằn" nhất nhưng cũng là nơi chứa đựng vẻ đẹp toán học của CNN. Nếu như Forward Pass là quá trình máy tính "nhìn" bức ảnh, thì Backward Pass là lúc nó "tự kiểm điểm" để rút kinh nghiệm.

Chúng ta sẽ đi sâu vào việc làm thế nào sai số (Loss) được truyền ngược từ lớp cuối cùng về các lớp đầu để cập nhật bộ lọc (Kernel).

4.1 Mục tiêu của chúng ta

Giả sử ta có một lớp Convolution đơn giản:

$$Y = X * K$$

Trong đó:

- X : Ảnh đầu vào (Input).
- K : Bộ lọc (Kernel/Filter).

- Y : Bản đồ đặc trưng đầu ra (Output Feature Map).
- L : Hàm mất mát (Loss Function) cuối cùng của mạng.

Nhiệm vụ của Backpropagation là tìm ra hai đạo hàm riêng (Gradients) quan trọng:

1. $\frac{\partial L}{\partial K}$: Để cập nhật trọng số của bộ lọc (giúp máy học được đặc trưng tốt hơn).
2. $\frac{\partial L}{\partial X}$: Để truyền sai số về cho lớp đứng trước nó (tiếp tục chuỗi Backprop).

4.2 Tính đạo hàm cho Bộ lọc ($\frac{\partial L}{\partial K}$)

Tưởng tượng Kernel K đang trượt trên ảnh X . Mỗi lần trượt, nó tạo ra một giá trị trên Y . Vì vậy, một trọng số trong Kernel sẽ đóng góp vào tất cả các pixel đầu ra mà nó đi qua.

Theo quy tắc chuỗi (Chain Rule), đạo hàm của lỗi đối với một trọng số w trong Kernel là tổng hợp của tất cả các lỗi mà trọng số đó gây ra trên toàn bộ bản đồ đầu ra Y :

$$\frac{\partial L}{\partial K_{ij}} = \sum_m \sum_n \frac{\partial L}{\partial Y_{mn}} \cdot \frac{\partial Y_{mn}}{\partial K_{ij}}$$

Điều thú vị là: Khi bạn triển khai công thức này ra, nó chính là một phép Cross-Correlation giữa ảnh đầu vào X và bản đồ lỗi $\frac{\partial L}{\partial Y}$.

Thực giác: Để biết nên sửa Kernel thế nào, ta lấy "Bản đồ lỗi" ướm lên "Ảnh gốc". Nơi nào ảnh gốc có giá trị lớn mà lỗi cũng lớn, nghĩa là Kernel tại đó cần được điều chỉnh mạnh tay nhất.

4.3 Tính đạo hàm cho Đầu vào ($\frac{\partial L}{\partial X}$) - Sự xuất hiện của Rot180

Đây là phần gây lú lẫn nhất và cũng là lý do tại sao nhiều tài liệu toán học lại nhắc đến việc "xoay 180 độ".

Để tính gradient cho lớp đầu vào X (nhằm truyền về lớp trước), ta cần biết mỗi pixel x_{ij} đã ảnh hưởng đến bao nhiêu pixel trên đầu ra Y . Theo toán học, công thức để tính đạo hàm này chính là phép tích chập (Convolution) giữa bản đồ lỗi $\frac{\partial L}{\partial Y}$ (được lót thêm viền - padded) với Kernel đã bị xoay 180 độ ($rot180(K)$).

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} * rot180(K)$$

Tại sao phải xoay?

- Trong quá trình Forward (đi tới), pixel x_{11} nhân với k_{11} .
- Nhưng khi Backward (đi lùi), để truy ngược dòng chảy từ Y về X , các mối liên hệ không gian bị đảo ngược. Phần tử góc trên-trái của Kernel lúc đi tới thực chất lại tương ứng với phần tử góc dưới-phải khi ta xét từ góc độ lan truyền ngược.

4.4 Sự thật về Cross-Correlation vs. Convolution

Trong các thư viện Deep Learning thực tế (như PyTorch hay TensorFlow), có một sự khác biệt nhỏ giữa lý thuyết và thực hành:

- Lý thuyết Toán học: Phép Convolution chuẩn ($f * g$) yêu cầu phải lật ngược Kernel 180 độ trước khi trượt.
- Thực tế Code: Hầu hết các thư viện sử dụng phép Cross-Correlation (trượt mà KHÔNG lật) cho quá trình Forward Pass để giảm chi phí tính toán.

Chính vì Forward Pass dùng Cross-Correlation (không xoay), nên để đảm bảo tính đúng đắn của đạo hàm, quá trình Backward Pass bắt buộc phải thực hiện phép toán ngược lại - chính là Convolution thực sự (tức là Cross-Correlation với Kernel xoay 180 độ).

Tóm lại quy trình chuẩn:

- **Forward:** Trượt Kernel thẳng (Cross-Correlation).
- **Backward (tính cho X):** Trượt Kernel xoay 180 độ (Convolution chuẩn).

Đây là vẻ đẹp của sự đối xứng trong toán học: Anh đi tới bằng cách trượt xuôi, thì anh phải đi lùi bằng cách trượt ngược để mọi thứ khớp lại hoàn hảo.

Phần 5: CNN Nâng cao - 1x1 Convolution & Phân biệt chiều không gian

Khi thiết kế các mạng CNN sâu (Deep Learning), chúng ta thường gặp một nghịch lý: càng đi sâu, chúng ta càng muốn tạo ra nhiều Feature Map để bắt được nhiều đặc trưng phức tạp. Tuy nhiên, điều này dẫn đến một vấn đề nan giải về chi phí tính toán. Đây chính là lúc **1x1 Convolution** xuất hiện.

5.1 Vấn đề: Sự bùng nổ chiều sâu (High Dimensional Problem)

Hãy tưởng tượng sau một vài lớp tích chập, bạn thu được một khối dữ liệu có độ sâu rất lớn, ví dụ: $H \times W \times 192$ (192 kênh/filters). Nếu tiếp tục thực hiện tích chập 3×3 bình thường trên khối này, số lượng phép tính sẽ bùng nổ vì Kernel phải xử lý đồng thời cả 192 kênh đầu vào. Đây là bài toán "High Dimensionality" khiến mô hình trở nên nặng nề và chậm chạp.

Convolutions

Height and Width
goes down

type	patch size/ stride	output size
convolution	$7 \times 7 / 2$	$112 \times 112 \times 64$
max pool	$3 \times 3 / 2$	$56 \times 56 \times 64$
convolution	$3 \times 3 / 1$	$56 \times 56 \times 192$
max pool	$3 \times 3 / 2$	$28 \times 28 \times 192$
inception (3a)		$28 \times 28 \times 256$
inception (3b)		$28 \times 28 \times 480$
max pool	$3 \times 3 / 2$	$14 \times 14 \times 480$
inception (4a)		$14 \times 14 \times 512$
inception (4b)		$14 \times 14 \times 512$
inception (4c)		$14 \times 14 \times 512$
inception (4d)		$14 \times 14 \times 528$
inception (4e)		$14 \times 14 \times 832$
max pool	$3 \times 3 / 2$	$7 \times 7 \times 832$
inception (5a)		$7 \times 7 \times 832$
inception (5b)		$7 \times 7 \times 1024$
avg pool	$7 \times 7 / 1$	$1 \times 1 \times 1024$
dropout (40%)		$1 \times 1 \times 1024$
linear		$1 \times 1 \times 1000$
softmax		$1 \times 1 \times 1000$

Number of
Features Filter
goes up...

Figure 6: Vấn đề chiều sâu: Số lượng kênh (Channels) tăng lên quá cao (ví dụ: 192) gây áp lực tính toán.

5.2 Cách tiếp cận: Tóm tắt chứ không vứt bỏ

Để giải quyết vấn đề này, ta cần giảm số lượng kênh xuống (ví dụ từ 192 xuống 64).

- **Cách sai:** Xóa bớt các Filter đi (Drop Filters). Điều này làm mất mát thông tin quan trọng mà mạng đã học được.
- **Cách đúng:** "Tóm tắt" các Filter lại. Chúng ta cần một cơ chế để **gộp** thông tin từ 192 kênh này lại thành 64 kênh tinh gọn hơn mà vẫn giữ được nội dung cốt lõi.

Đó là lý do 1×1 Convolution ra đời. Nó hoạt động như một bộ nén thông tin thông minh.

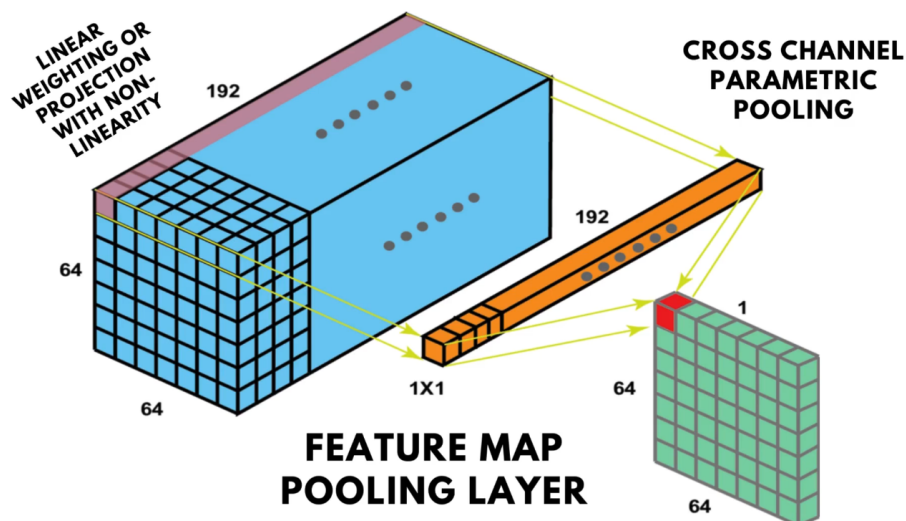


Figure 7: Cơ chế 1×1 Conv: Một thanh "xuyên tảo" đi qua tất cả các kênh để tổng hợp thông tin tại một điểm ảnh.

5.3 Bản chất của 1x1 Conv: "Single Neuron" trên trục sâu

Thoạt nhìn, một Kernel kích thước 1×1 có vẻ vô dụng vì nó không nhìn thấy các pixel lân cận (không thay đổi kích thước không gian $H \times W$). Tuy nhiên, bí mật nằm ở chỗ: Kernel này không phẳng!

Một Kernel 1×1 trong CNN thực chất là một khối vector có kích thước $1 \times 1 \times Depth$ (với Depth bằng đúng số kênh đầu vào).

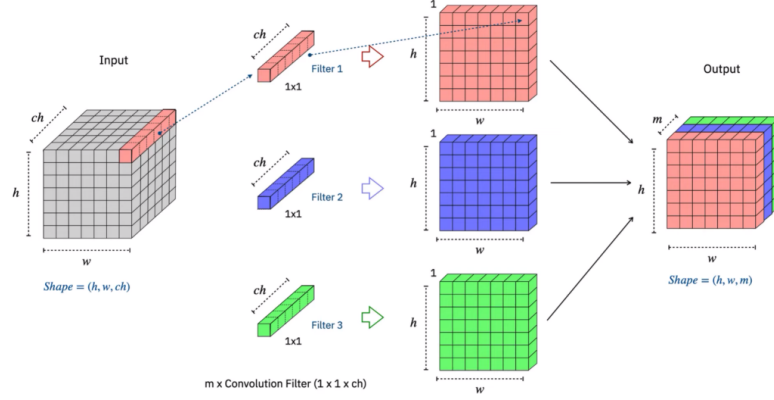


Figure 8: Minh họa trực quan: Kernel 1x1 thực chất trải dài hết chiều sâu của khối dữ liệu đầu vào.

Tại sao nó quan trọng?

- **Pointwise Convolution (Tích chập điểm):** Vì Kernel chỉ tác động lên đúng một vị trí tọa độ (x, y) nhưng xuyên suốt tất cả các kênh, nó thực hiện phép tính tổng hợp thông tin (Feature Fusion) tại điểm đó.
- **Cross Channel Parametric Pooling:** Nó hoạt động như một lớp Pooling có trọng số chạy dọc theo trục kênh. Ví dụ: nó có thể học cách kết hợp đặc trưng "mắt" (kênh 10) và "mũi" (kênh 20) để tạo ra đặc trưng "khuôn mặt" (kênh output), trong khi bỏ qua các kênh nhiễu khác.

Về mặt toán học, giá trị đầu ra là tổng có trọng số của tất cả các kênh tại một điểm ảnh, cộng với bias và đi qua hàm kích hoạt:

$$\text{Output}(x, y) = \sigma \left(\sum_{c=1}^{C_{in}} W_c \cdot \text{Input}(x, y, c) + b \right)$$

Điều này tương đương với việc áp dụng một mạng Neural nhỏ (Fully Connected) lên từng pixel riêng biệt.

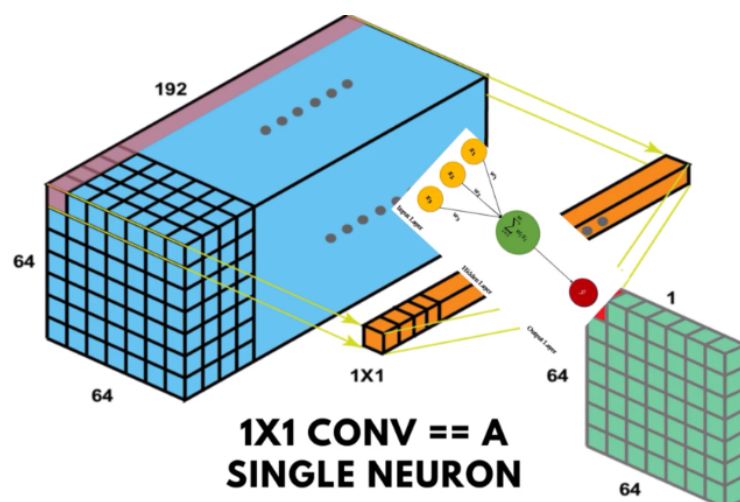


Figure 9: 1x1 Convolution tương đương với một Single Neuron áp dụng lên chiều sâu của từng pixel.

5.4 Ứng dụng: Kiến trúc Bottleneck

Ứng dụng kinh điển nhất của 1×1 Conv là tạo ra kiến trúc **Bottleneck** (thường thấy trong Inception, ResNet). Quy trình "ép nhỏ" dữ liệu diễn ra như sau:

1. Dùng 1×1 Conv để **giảm số chiều** (ví dụ: 256 kênh $\rightarrow$ 64 kênh), giúp loại bỏ các thông tin dư thừa.
2. Thực hiện tích chập 3×3 đặt đổ trên số kênh ít ỏi này (tính toán rất nhanh).
3. Dùng 1×1 Conv một lần nữa để **tăng số chiều** trở lại (64 kênh $\rightarrow$ 256 kênh) để khôi phục độ sâu đặc trưng cho các lớp sau.

Nhờ "chiếc cổ chai" hẹp ở giữa, mô hình có thể xây dựng sâu hơn (deeper) nhưng số lượng tham số lại ít hơn rất nhiều so với bình thường.

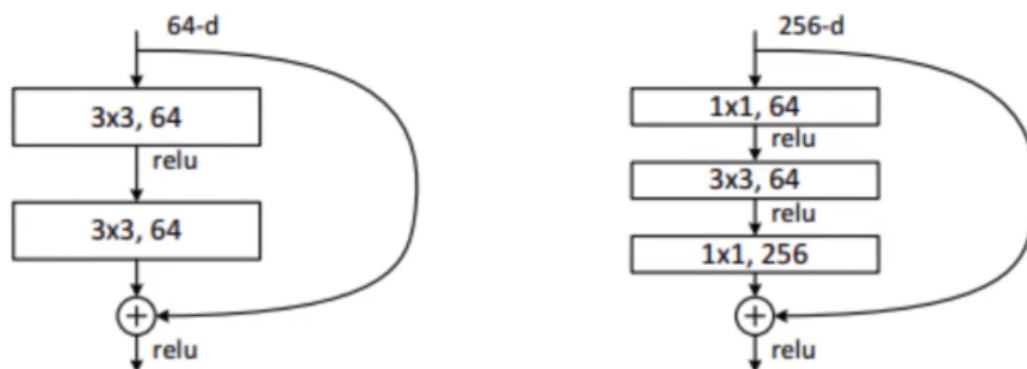


Figure 10: Kiến trúc Bottleneck: Sử dụng 1x1 Conv để nén dữ liệu trước khi xử lý, giúp tối ưu hóa hiệu năng.

5.5 So sánh nhanh: Pooling vs. 1x1 Convolution

Trong thực tế, hai kỹ thuật này không đối đầu mà **bổ trợ lẫn nhau**. Chúng thường xuất hiện cùng nhau (như trong Inception Block) để tối ưu hóa mô hình theo ba khía cạnh chính: Không gian, Khả

năng học và Chi phí tính toán.

1. Về chiều biến đổi (Dimensionality):

Pooling: Giảm kích thước không gian chiều rộng và cao ($H \times W$), nhưng bắt buộc phải **giữ nguyên** số lượng kênh (Filters).

1x1 Convolution: Giữ nguyên kích thước không gian ($H \times W$), nhưng cho phép **thay đổi** số lượng kênh (Filters) - có thể giảm để nén hoặc tăng để mở rộng.

2. Về khả năng học (Learnability):

Pooling: Là một thao tác tĩnh (Fixed operation). Nó không có trọng số để huấn luyện; bạn chỉ cần cài đặt tham số ban đầu (như kernel size, stride) và nó sẽ chạy cố định mãi mãi.

1x1 Convolution: Là một thao tác động có thể học (Learnable). Nó sở hữu bộ trọng số riêng và được tối ưu hóa liên tục thông qua quá trình **Gradient Descent** để tìm ra cách kết hợp các kênh tốt nhất.

3. Về chi phí tính toán (Computation Cost):

Pooling: Chi phí rất thấp (Low computation) vì chỉ thực hiện các phép so sánh (Max) hoặc cộng trừ đơn giản (Avg).

1x1 Convolution: Chi phí cao hơn vì bản chất là các phép nhân ma trận dày đặc và cần cập nhật trọng số ngược. Tuy nhiên, nó lại giúp "tiết kiệm" chi phí cho các lớp 3×3 phía sau nhờ việc giảm số kênh đầu vào (cơ chế Bottleneck).

4. Lợi ích phụ (Non-linearity):

1x1 Convolution thường đi kèm với hàm kích hoạt (như ReLU), giúp **tăng tính phi tuyến** cho mô hình, cho phép mạng học được các hàm phức tạp hơn mà không làm thay đổi cấu trúc không gian.

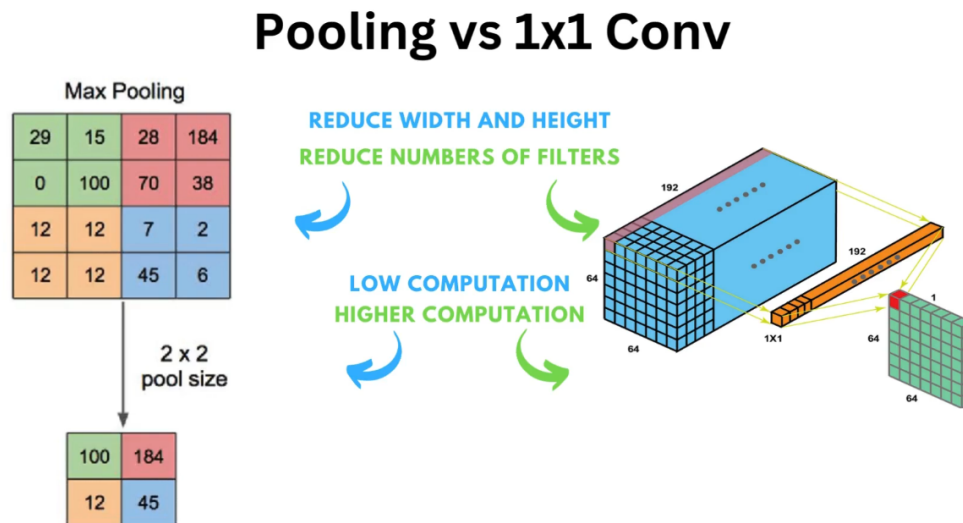


Figure 11: Pooling nén dọc (XY) còn 1x1 Conv nén ngang (Depth)