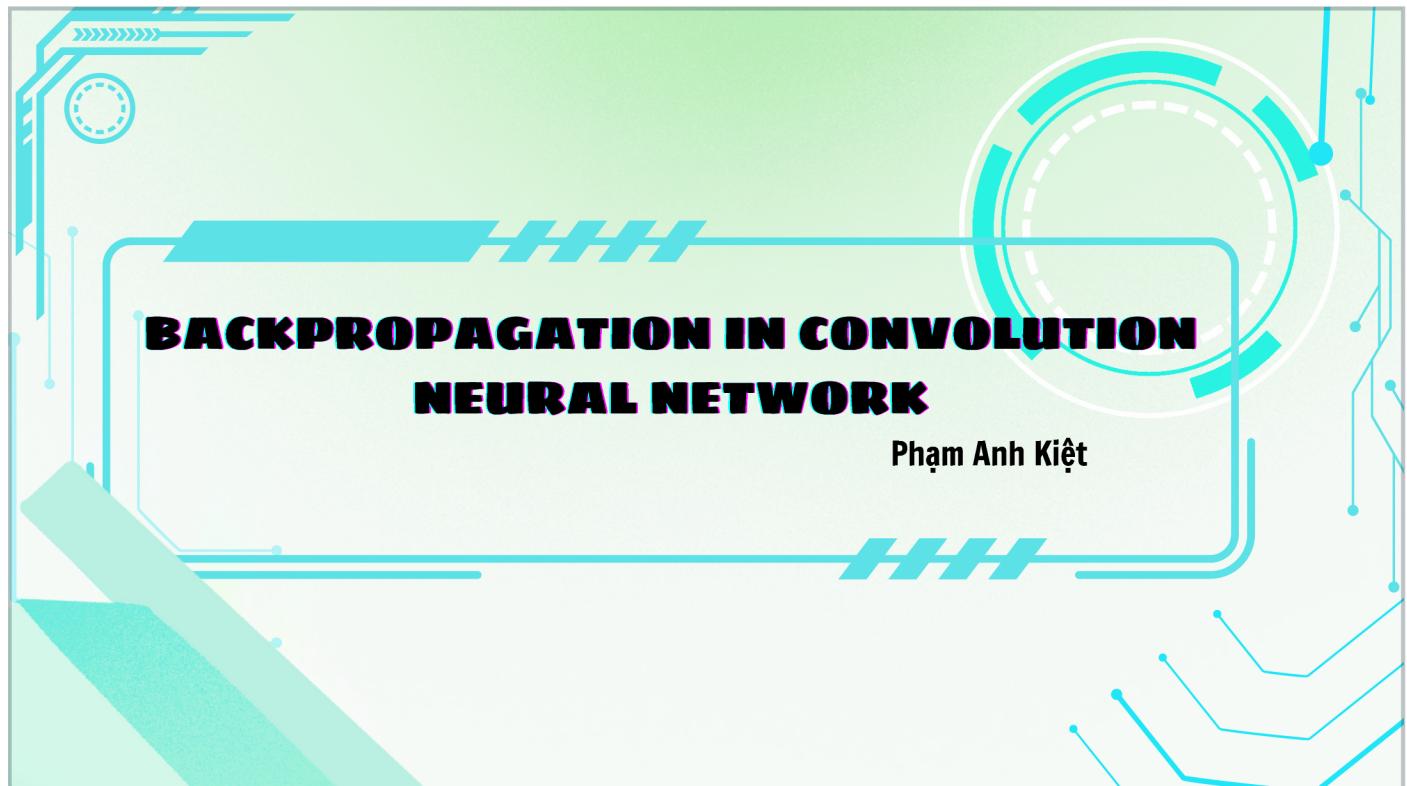




# **BACKPROPAGATION IN CONVOLUTION NEURAL NETWORK**

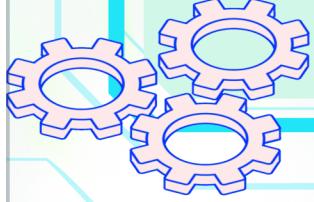
Phạm Anh Kiệt



# **NỘI DUNG TRÌNH BÀY**

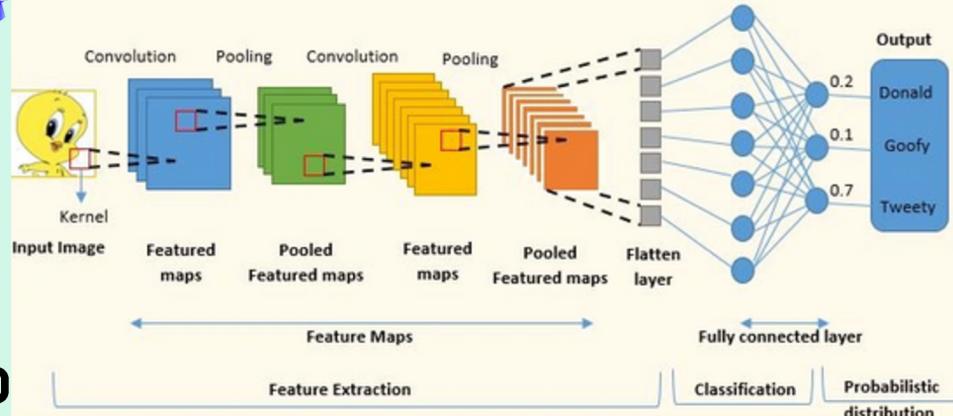


1. Tổng quan về mạng CNN
2. Backpropagation trong CNN
3. Code Demo



# TỔNG QUAN VỀ CONVOLUTION NEURAL NETWORK

*A Typical Convolutional Neural Network (CNN)*

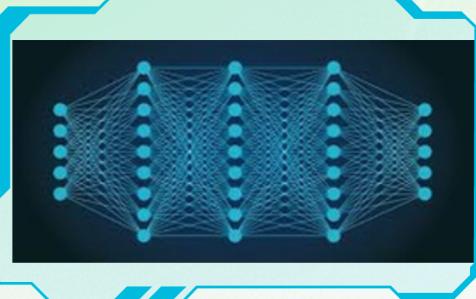


Tại sao lại dùng CNN đối với dữ liệu đầu vào là ảnh hoặc video?

### Mạng neural network thông thường:

Giả sử ta có đầu vào là 1 ảnh màu  $200 \times 200$  biểu diễn dưới dạng 1 tensor là  $200 \times 200 \times 3$ . Tổng số pixel của input là  $200 \times 200 \times 3 = 12000$  nodes (neural).

Giả sử số lượng node trong hidden layer 1 là 500 node thì số lượng tham số giữa input và layer sẽ là:  $12000 \times 500 + 500$  (bias) = 60500 tham số.



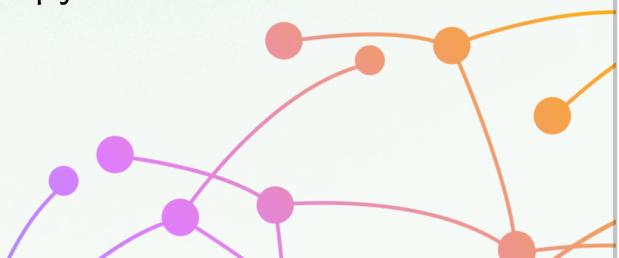
Khi số layer nhiều hơn hoặc kích thước ảnh tăng làm cho số lượng tham số rất lớn.  
=> Cần tìm cách giảm số lượng tham số của mô hình -> CNN

# CONVOLUTION NEURAL NETWORK

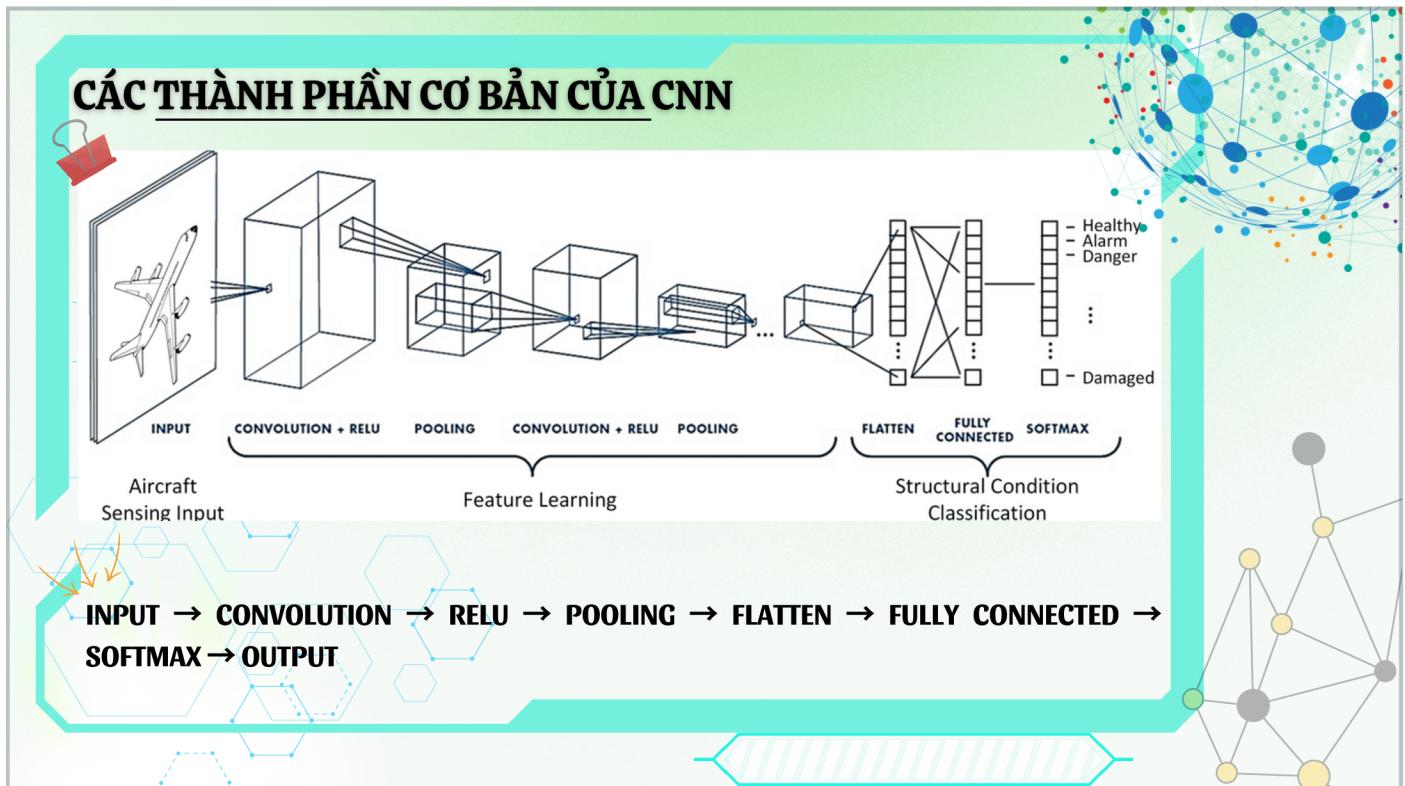
Mạng CNN (Convolutional Neural Network) là một loại mạng nơ-ron nhân tạo, được thiết kế đặc biệt để xử lý dữ liệu có cấu trúc lưới, chẳng hạn như hình ảnh hoặc chuỗi thời gian.

CNN được ứng dụng rộng rãi trong các lĩnh vực như nhận diện hình ảnh, xử lý ngôn ngữ tự nhiên, phát hiện đối tượng, và thị giác máy tính.

Đối với bài toán về xử lý ảnh thì mạng CNN sẽ trích xuất những đặc trưng quan trọng của ảnh (giảm kích thước ảnh) do đó giả quyết được bài toán của mạng neural network thông thường.



## CÁC THÀNH PHẦN CƠ BẢN CỦA CNN



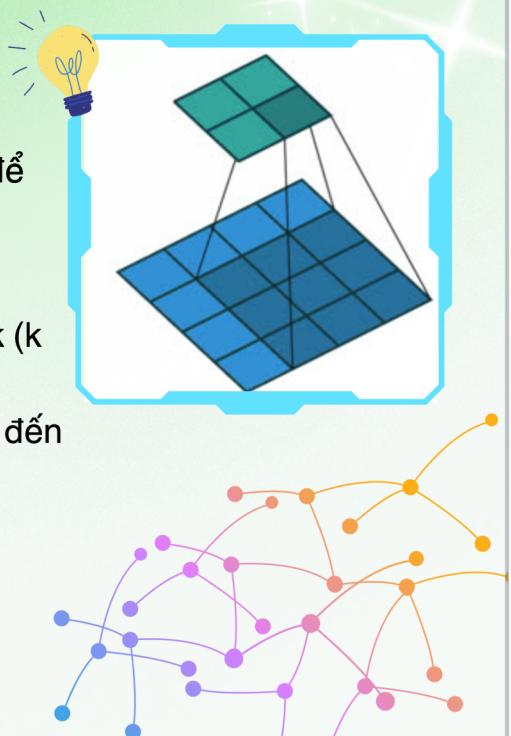
# CONVOLUTION LAYER

## Chức năng:

Tạo ra bản đồ đặc trưng (activation map / feature map) để làm đầu vào cho những lớp tiếp theo.

## Cách hoạt động:

- Kernel (filter, mask): là ma trận vuông có kích thước  $k * k$  ( $k$  thường là số lẻ).
- Kernel trượt trên ma trận ảnh và thực hiện tích chập cho đến khi hết ma trận.

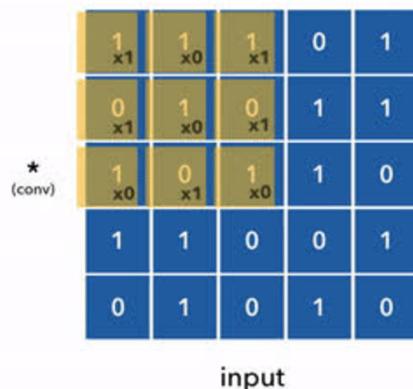


**Cách tính tích chập:** Với đầu vào  $X(H \times W)$ , Kernel( $F \times F$ ), bias, stride s, padding p:

$$Z[i, j] = \sum_{m=0}^{F-1} \sum_{n=0}^{F-1} X[i+m, j+n] \cdot W[m, n] + b$$



filters



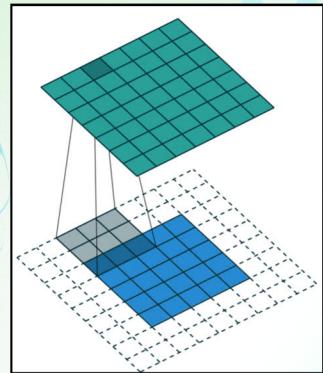
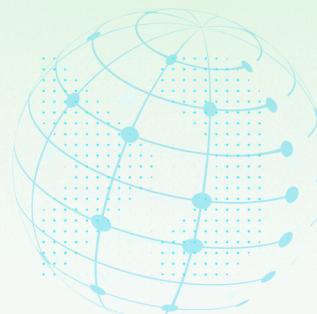
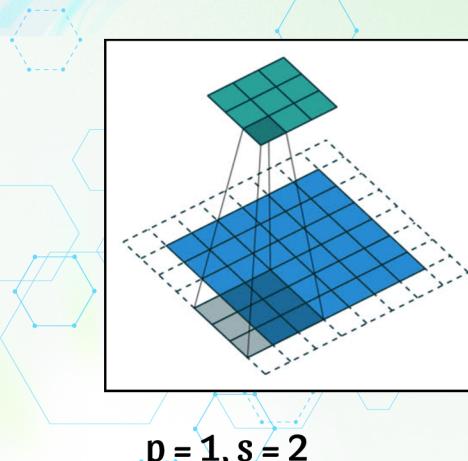
output

**Vấn đề:** Nếu chỉ dùng kernel chập lên ảnh ban đầu thì các pixel phía biên của ảnh sẽ được chập ít hơn so với bên trong.

# SỬ DỤNG PADDING VÀ STRIDE

1.Padding: thêm các vector 0 vào mỗi phía của ma trận ảnh → các pixel phần biên sẽ được chập nhiều hơn.

2.Stride (bước nhảy): nếu bước nhảy càng lớn thì kích thước ma trận đầu ra sẽ giảm



Giả sử ma trận đầu vào có kích thước  $m \times n$ , kernel có kích thước là  $k \times k$ , padding là  $p$  và stride là  $s \rightarrow$  kích thước ma trận sau khi chập là:

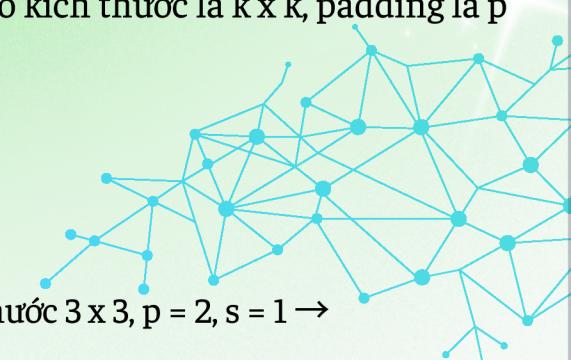
$$\left( \frac{m-k+2p}{s} + 1, \frac{n-k+2p}{s} + 1 \right)$$

Ví dụ: ma trận ảnh có kích thước  $5 \times 5$ , kernel có kích thước  $3 \times 3$ ,  $p = 2$ ,  $s = 1 \rightarrow$

Output:

$$\left( \frac{5-3+4}{1} + 1, \frac{5-3+4}{1} + 1 \right) = (6, 6)$$

Lưu ý: Thông thường trong mỗi convolution layer thường sử dụng nhiều kernel để học được nhiều đặc trưng ảnh.



# TÍCH CHẬP CHO ẢNH MÀU

Ảnh màu (3 kênh màu r, g, b) sẽ được biểu diễn một tensor có kích thước  $H * W * D$  ( $D = 3$ ).

Ảnh sẽ được tách thành 3 kênh màu riêng biệt và hình thành 3 ma trận  $H * W$ .

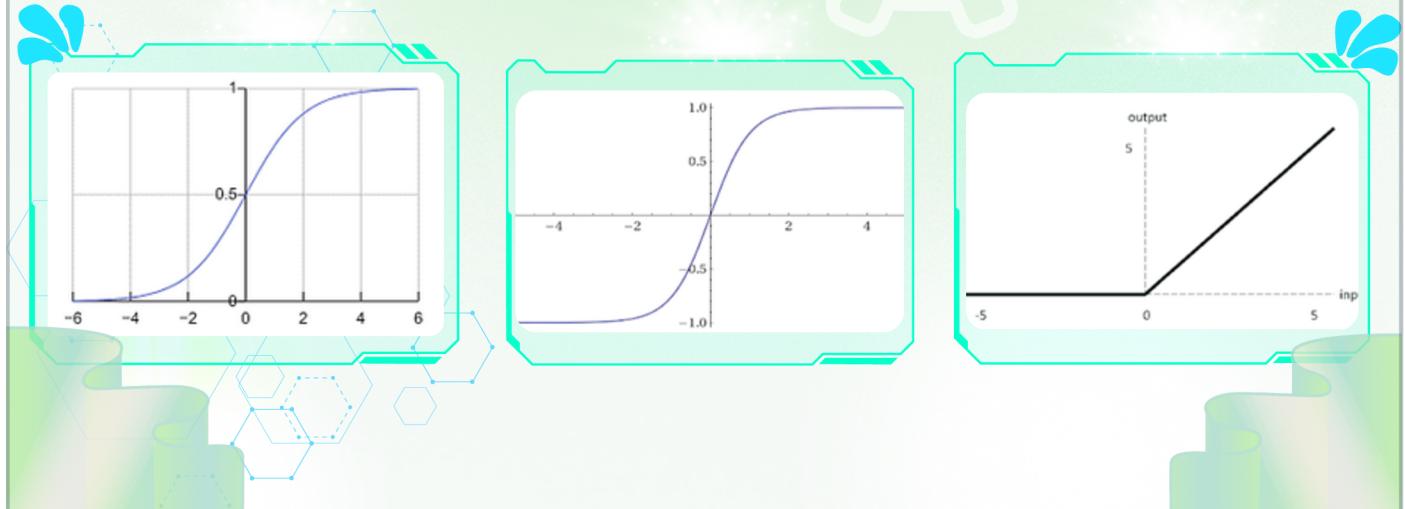
| Input Volume (+pad 1) (7x7x3)   | Filter W0 (3x3x3)  | Filter W1 (3x3x3)   | Output Volume (3x3x2)                                       |
|---|--|---|---|
| $x[:, :, 0]$  | $w0[:, :, 0]$  | $w1[:, :, 0]$   | $o[:, :, 0]$  |
| 0 0 0 0 0 0 0<br>0 0 0 1 0 2 0<br>0 1 0 2 0 1 0<br>0 1 0 2 2 0 0<br>0 2 0 0 2 0 0<br>0 2 1 2 2 0 0<br>0 0 0 0 0 0 0 | -1 0 1<br>0 0 1<br>1 -1 1<br>-1 0 1<br>1 -1 1<br>0 1 0<br>0 -1 0 | 0 1 -1<br>0 -1 0<br>0 -1 1<br>-1 0 0<br>1 -1 0<br>1 -1 0<br>1 0 0 | 2 3 3<br>3 7 3<br>8 10 -3<br>-8 -8 -3<br>-3 1 0<br>-3 -8 -5 |
| $x[:, :, 1]$  | $w0[:, :, 1]$  | $w1[:, :, 1]$   | $o[:, :, 1]$  |
| 0 0 0 0 0 0 0<br>0 2 1 2 1 1 0<br>0 2 1 2 0 1 0<br>0 0 2 1 0 1 0<br>0 1 2 2 2 2 0<br>0 0 1 2 0 1 0<br>0 0 0 0 0 0 0 | -1 0 1<br>1 0 0<br>0 1 0<br>1 0 0<br>0 -1 0<br>0 1 0<br>0 0 0    | -1 1 -1<br>0 -1 -1<br>1 0 0                                       | -8 -8 -3<br>-3 1 0<br>-3 -8 -5                              |
| $x[:, :, 2]$  | $w0[:, :, 2]$  | $w1[:, :, 2]$   | $o[:, :, 2]$  |
| 0 0 0 0 0 0 0<br>0 2 1 2 0 2 0<br>0 1 0 0 1 0 0<br>0 0 1 0 0 0 0<br>0 1 0 2 1 0 0<br>0 2 2 1 1 1 0<br>0 0 0 0 0 0 0 | 0 0 0<br>1 0 0<br>0 1 0<br>1 0 0<br>0 -1 0<br>0 1 0<br>0 0 0     | 0 0 0<br>1 0 0<br>1 0 0<br>1 0 0                                  | 0 0 0   |
|   | Bias b0 (1x1x1)  | Bias b1 (1x1x1)   |   |
|   | $b0[x, :, 0]$  | $b1[z, :, 0]$   |   |
|   | 1  | 0   |   |

toggle movement



# ACTIVATION FUNCTION

- Một số hàm kích hoạt thường hay dùng như: tanh, sigmoid, Relu, Softmax ... để tăng tính phi tuyến tính cho mạng.
- Kích thước đầu vào và đầu ra của hàm activation sẽ không thay đổi.



# RELU

Một số ưu điểm của Relu:

1

Dễ dàng tính toán.

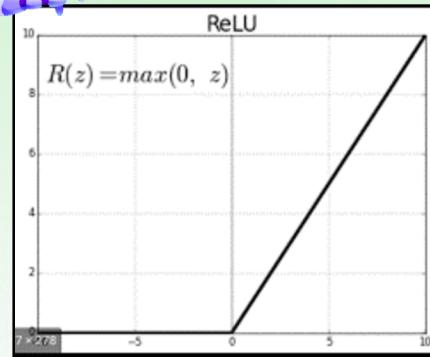
2

Tăng tốc quá trình training.

3

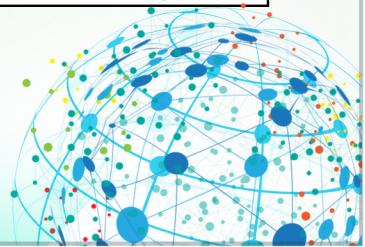
Nó không bị chặn như tanh và sigmoid nên hạn chế được vanishing gradient.

Khi đầu vào là âm thì kết quả cho ra sẽ là 0.



$$f(x) = \begin{cases} x & (\text{if } x > 0) \\ 0 & (\text{if } x \leq 0) \end{cases}$$

$$f'(x) = \begin{cases} 1 & (\text{if } x > 0) \\ 0 & (\text{if } x \leq 0) \end{cases}$$



# POOLING LAYER

- Pooling layer thường được dùng giữa các convolution layer để giảm kích thước dữ liệu nhưng vẫn giữ được những đặc trưng quan trọng của ảnh.
  - Pooling layer thường dùng kernel có kích thước  $K \times K$ .
  - Với mỗi ma trận, trên vùng kích thước  $K \times K$  ta tìm maximum hoặc average rồi viết vào ma trận kết quả.
- Quy tắc về stride, padding vẫn áp dụng như cũ.



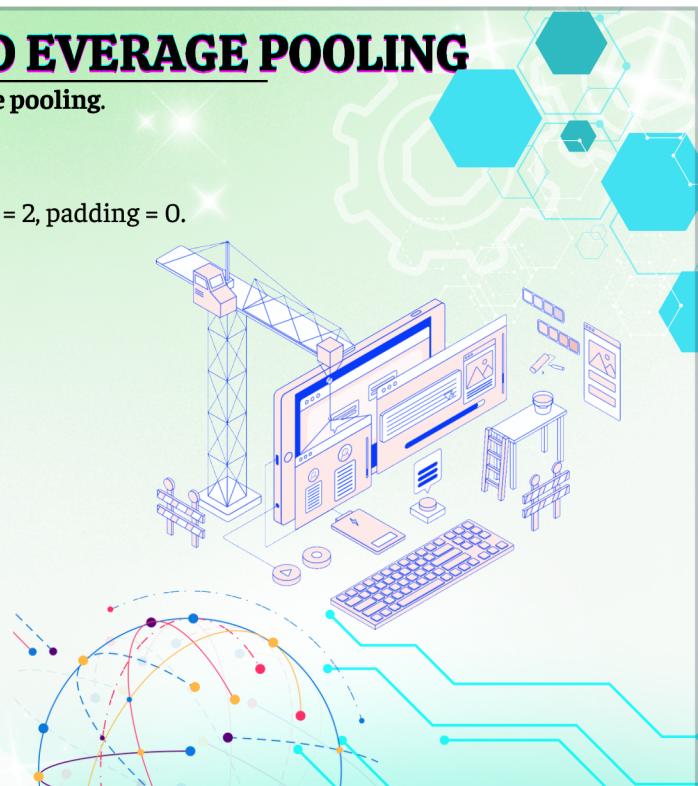
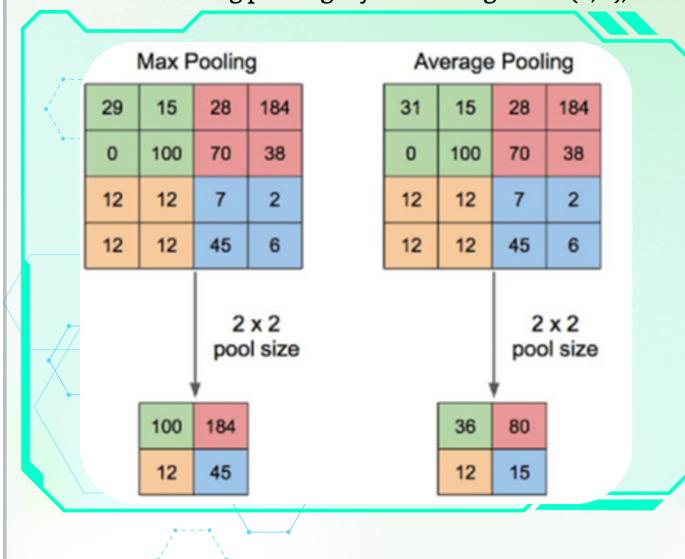
# MAX POOLING AND AVERAGE POOLING

Có 2 loại pooling layer phổ biến là **max pooling** và **average pooling**.

**Max pooling:** chọn giá trị lớn nhất trong vùng con.

**Average pooling:** tính giá trị trung bình trong vùng con

Hầu hết khi dùng pooling layer thì dùng size = (2, 2), stride = 2, padding = 0.



## FULLY CONNECTED LAYER

Sau khi ảnh truyền qua nhiều convolutional layer và pooling layer thì model đã học được tương đối các đặc điểm của ảnh.

Output của layer cuối cùng có kích thước là  $H \times W \times D$  sẽ được chuyển về 1 vector có kích thước  $(H \times W \times D)$  sẽ là đầu vào của FC. Mỗi phần tử của vector sẽ kết nối hết với tất cả các node trong FC.

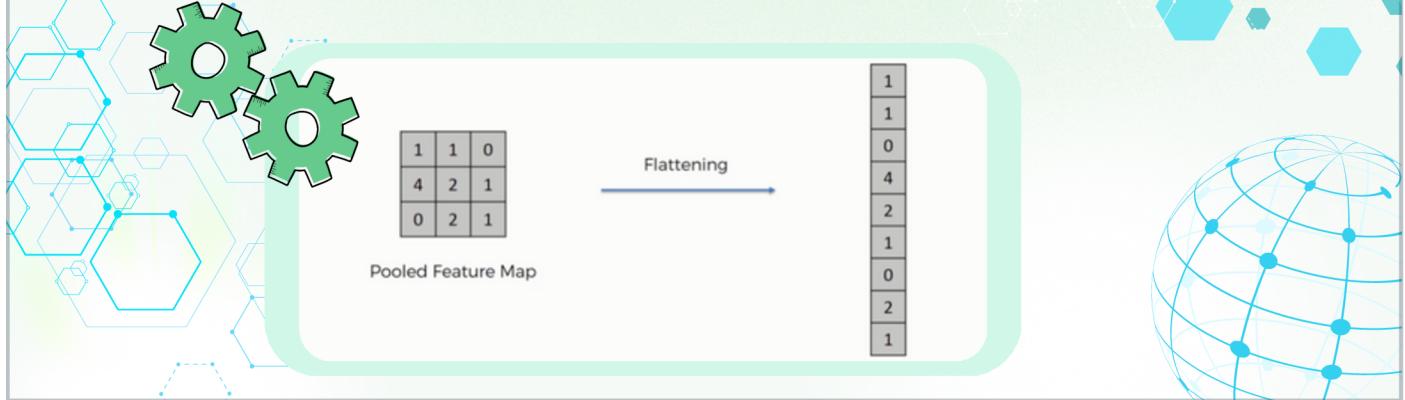
Ví dụ về flatten mảng có kích thước  $H \times W$ :

|   |   |   |
|---|---|---|
| 1 | 1 | 0 |
| 4 | 2 | 1 |
| 0 | 2 | 1 |

Pooled Feature Map

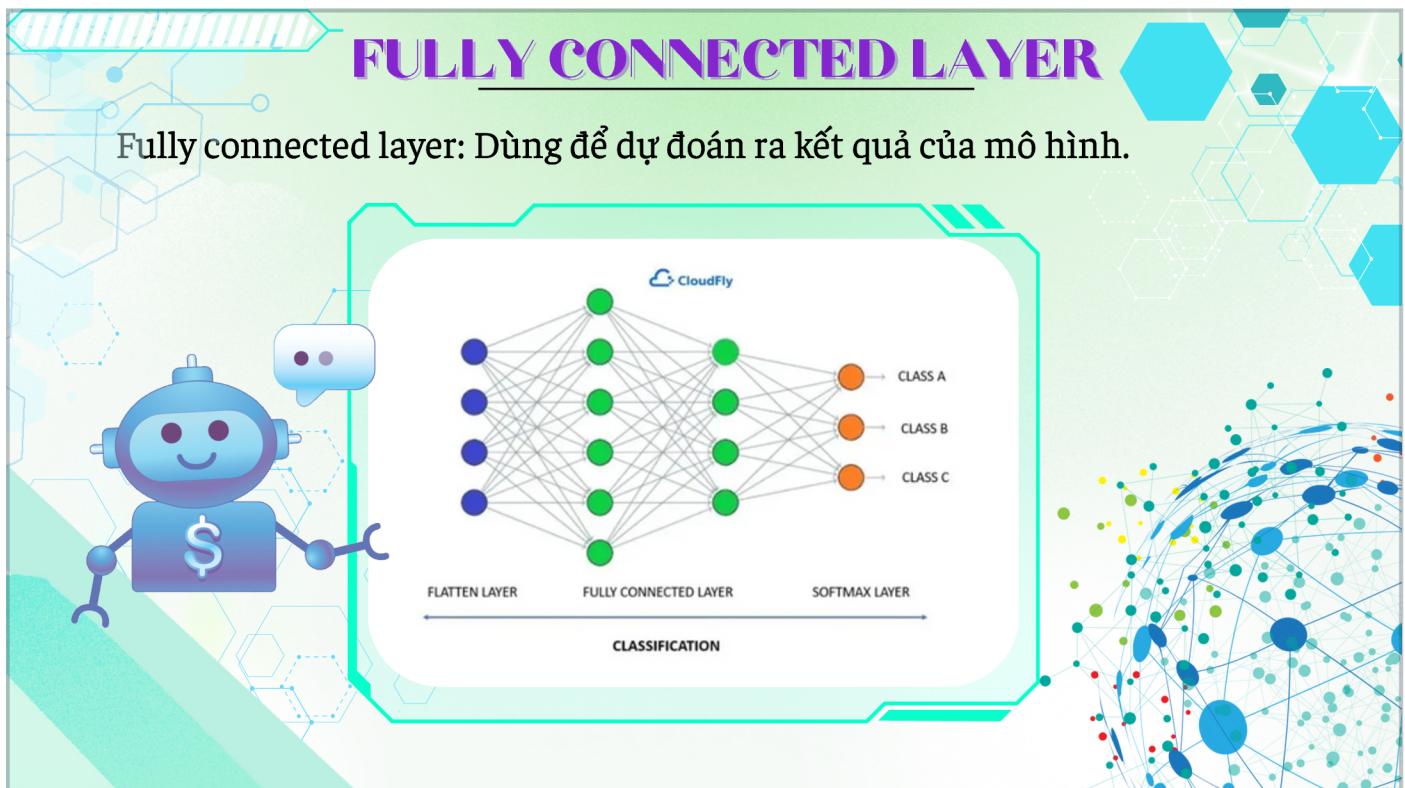
Flattening

|   |
|---|
| 1 |
| 1 |
| 0 |
| 4 |
| 2 |
| 1 |
| 0 |
| 2 |
| 1 |



# FULLY CONNECTED LAYER

Fully connected layer: Dùng để dự đoán ra kết quả của mô hình.



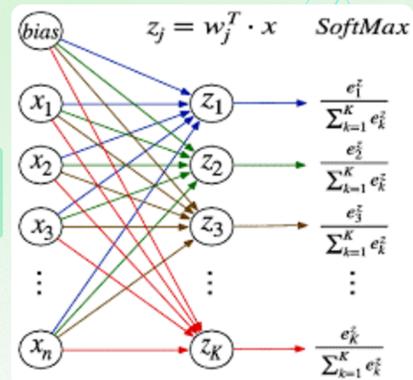
# SOFTMAX

Thường được dùng cho bài toán multi-class classification.

(Thường dùng ở cuối FC trước lớp Output).

Mục đích: chuyển đổi các giá trị đầu ra thành các xác suất

Giúp xác định lớp nào có xác suất xảy ra cao nhất.



# Công thức của hàm softmax

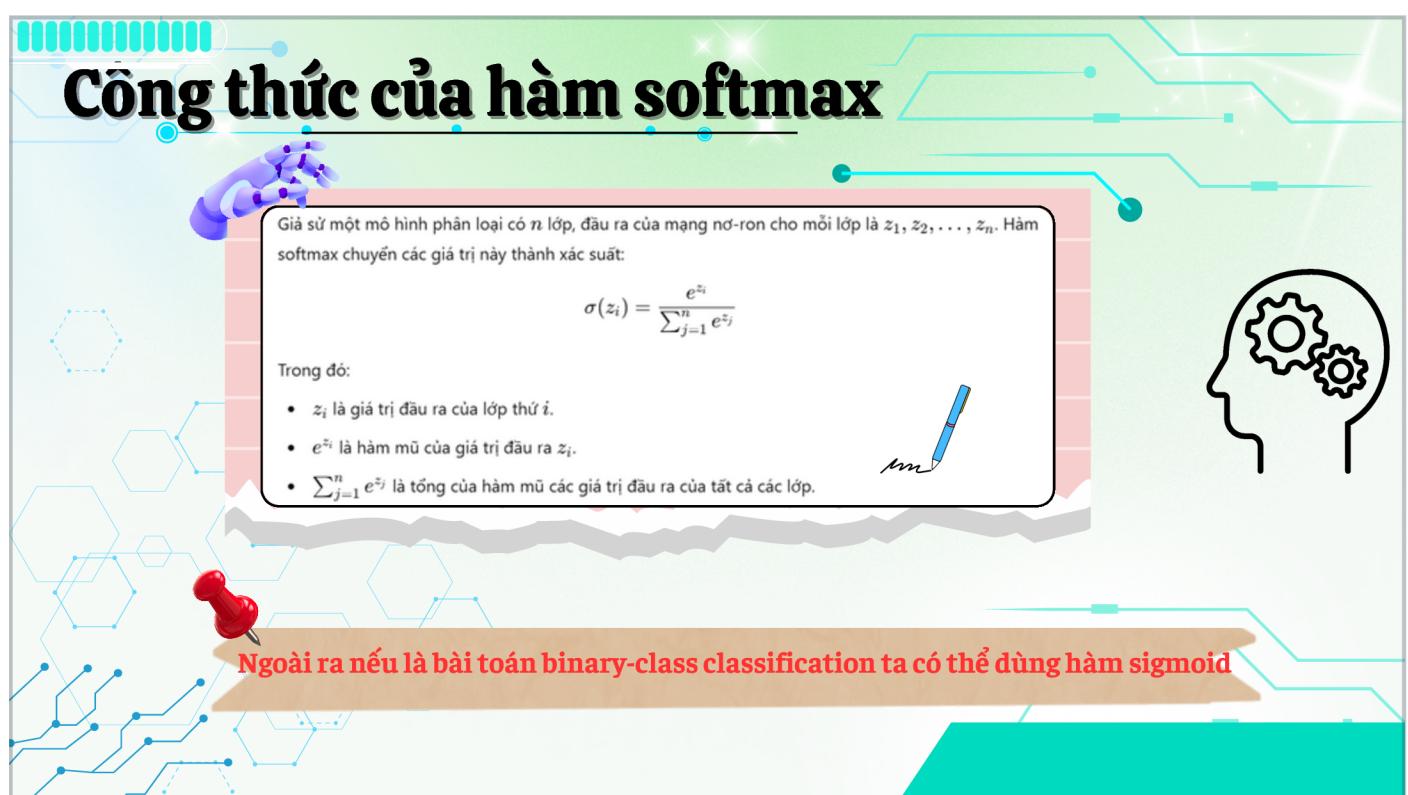
Giả sử một mô hình phân loại có  $n$  lớp, đầu ra của mạng nơ-ron cho mỗi lớp là  $z_1, z_2, \dots, z_n$ . Hàm softmax chuyển các giá trị này thành xác suất:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}}$$

Trong đó:

- $z_i$  là giá trị đầu ra của lớp thứ  $i$ .
- $e^{z_i}$  là hàm mũ của giá trị đầu ra  $z_i$ .
- $\sum_{j=1}^n e^{z_j}$  là tổng của hàm mũ các giá trị đầu ra của tất cả các lớp.

Ngoài ra nếu là bài toán binary-class classification ta có thể dùng hàm sigmoid



# LOSS FUNCTION

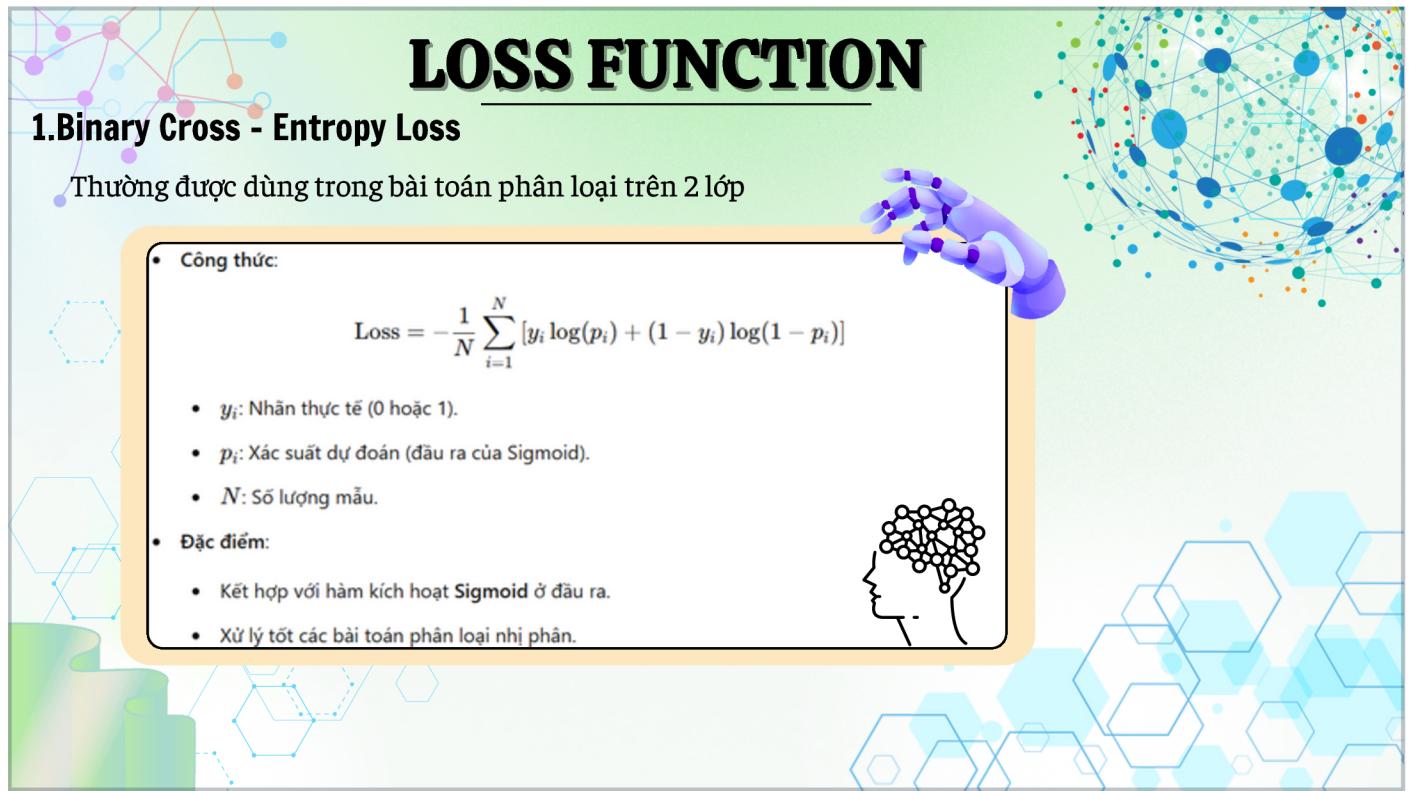
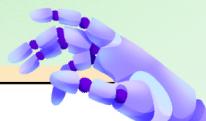
## 1. Binary Cross - Entropy Loss

Thường được dùng trong bài toán phân loại trên 2 lớp

### Công thức:

$$\text{Loss} = -\frac{1}{N} \sum_{i=1}^N [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

- $y_i$ : Nhãn thực tế (0 hoặc 1).
  - $p_i$ : Xác suất dự đoán (đầu ra của Sigmoid).
  - $N$ : Số lượng mẫu.
- **Đặc điểm:**
- Kết hợp với hàm kích hoạt Sigmoid ở đầu ra.
  - Xử lý tốt các bài toán phân loại nhị phân.



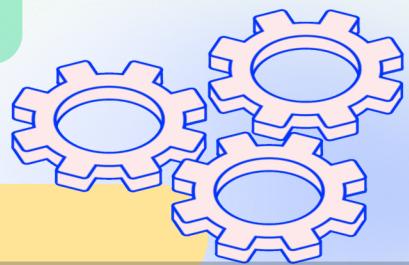
## 2. Categorical Cross - Entropy Loss



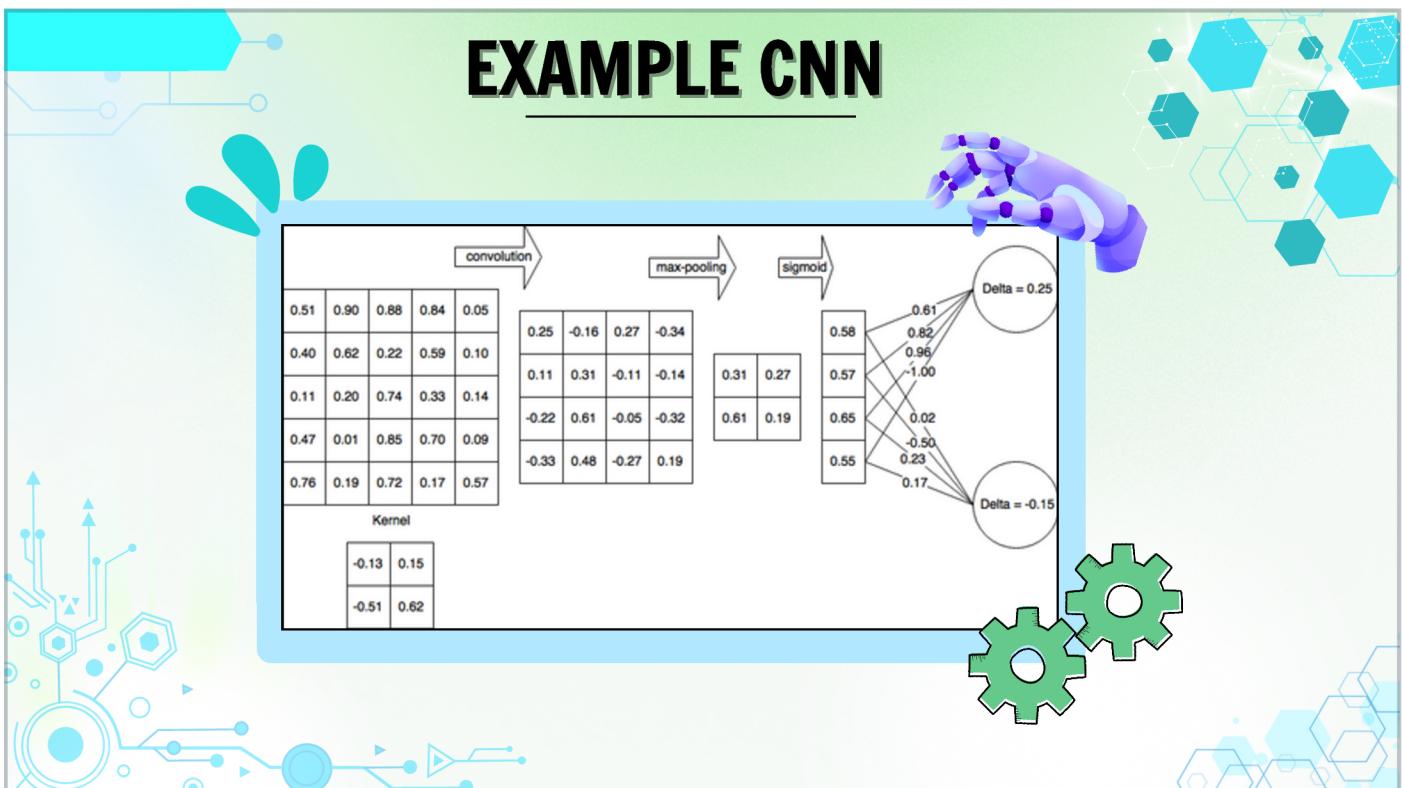
Công thức:

$$\text{Loss} = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^C y_{ij} \log(p_{ij})$$

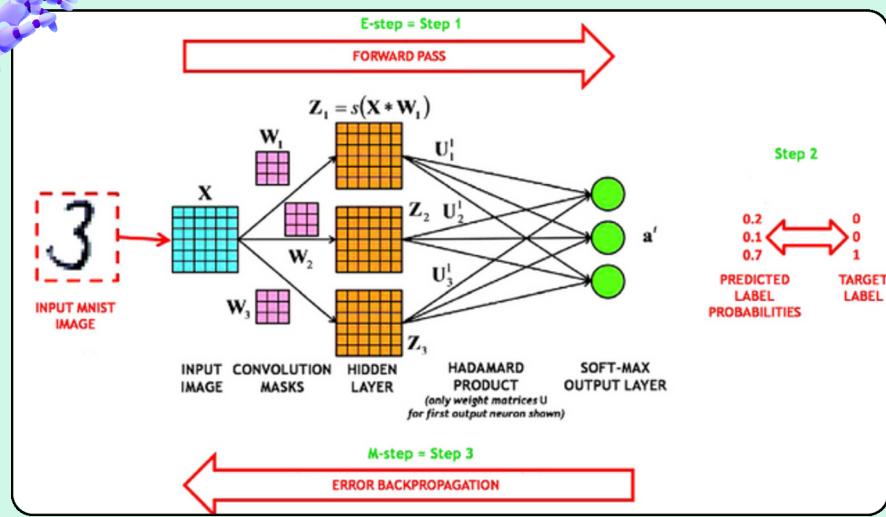
- $y_{ij}$ : Nhãn thực tế (dạng one-hot vector).
  - $p_{ij}$ : Xác suất dự đoán cho lớp  $j$  (đầu ra của Softmax).
  - $C$ : Số lớp.
- **Đặc điểm:**
- Kết hợp với hàm kích hoạt Softmax ở đầu ra.
  - Xử lý tốt các bài toán phân loại nhiều lớp.



# EXAMPLE CNN



# KỸ THUẬT BACKPROPAGATION IN CNN



# BACKPROPAGATION

1. Mục đích: cập nhật trọng số (weight, bias) cho mô hình CNN.

→ Fully connected layer.

→ Kernel, bias của Convolution layer.

Để giảm thiểu loss (đầu ra dự đoán chính xác hơn).

2. Quá trình backpropagation:

a Lan truyền tiến từ đầu vào đến đầu ra.

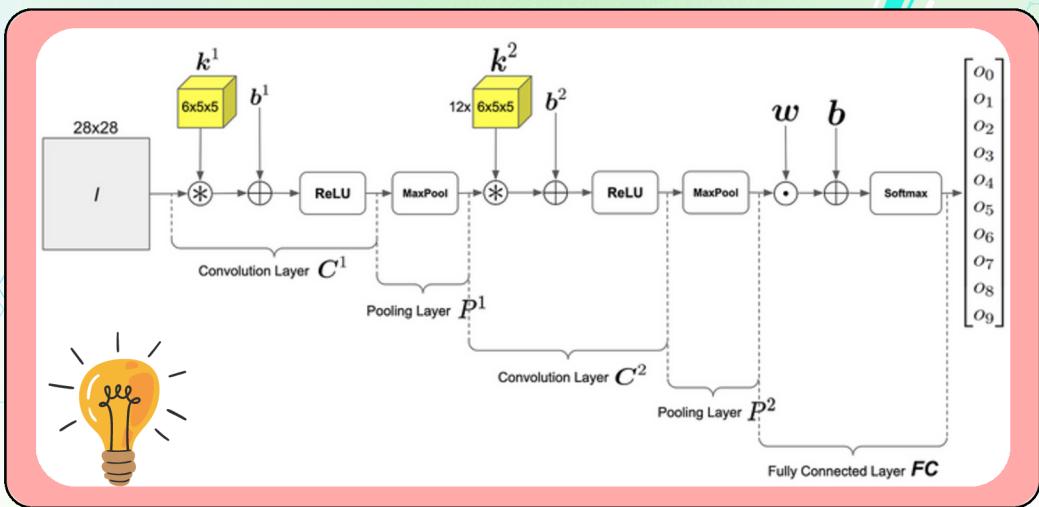
b Tính toán Loss.

c Lan truyền ngược từ đầu ra đến đầu vào dựa trên đạo hàm của hàm loss.

Hàm loss để minh họa cho kỹ thuật này là Categorical Cross – Entropy loss.



# BACKWARD



# CHAIN RULE

## Chain Rule

$$\frac{d}{dx}[f(g(x))] = f'(g(x))g'(x)$$

$$\begin{aligned}F &= f(u) = 2u^2 + 3u - 1 \\u(x) &= 2x - 2 \\F'(x) &= f'(u).u'(x) \\&= (4u + 3).2 \\&= (4.(2x - 2) + 3).2 \\&= 16x - 10\end{aligned}$$

**Chain rule trong backward:** Vì ta cần tính đạo hàm của hàm loss theo các trọng số W, b ở các layer phía trước đó.

$$W, b \rightarrow f(W, b) \rightarrow g(f(W, b)) \rightarrow Loss(g)$$

# BACKWARD ĐẾN ĐẦU VÀO CỦA SOFTMAX

• Tiến trình đầu ra: Đầu ra của FC-->Softmax--> Output

• Đạo hàm của hàm Categorical Cross – Entropy loss theo Z:

★ Hàm loss:

$$L = \sum_{i=1}^C Y_i \log(y_i)$$

$$\frac{\partial L}{\partial y_i} = \frac{Y_i}{y_i}$$

★ Hàm Softmax:

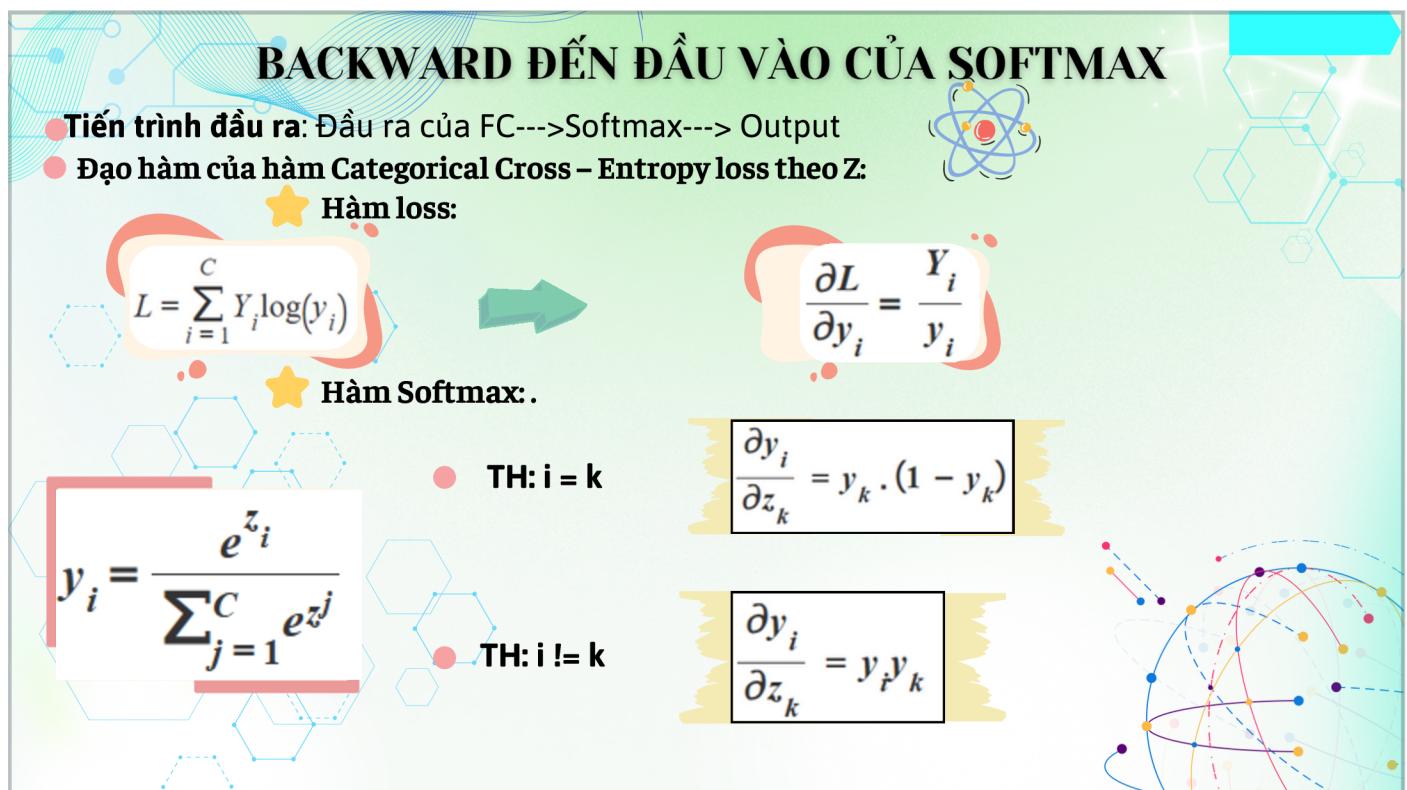
$$y_i = \frac{e^{z_i}}{\sum_{j=1}^C e^{z_j}}$$

• TH:  $i = k$

$$\frac{\partial y_i}{\partial z_k} = y_k \cdot (1 - y_k)$$

• TH:  $i \neq k$

$$\frac{\partial y_i}{\partial z_k} = y_i y_k$$

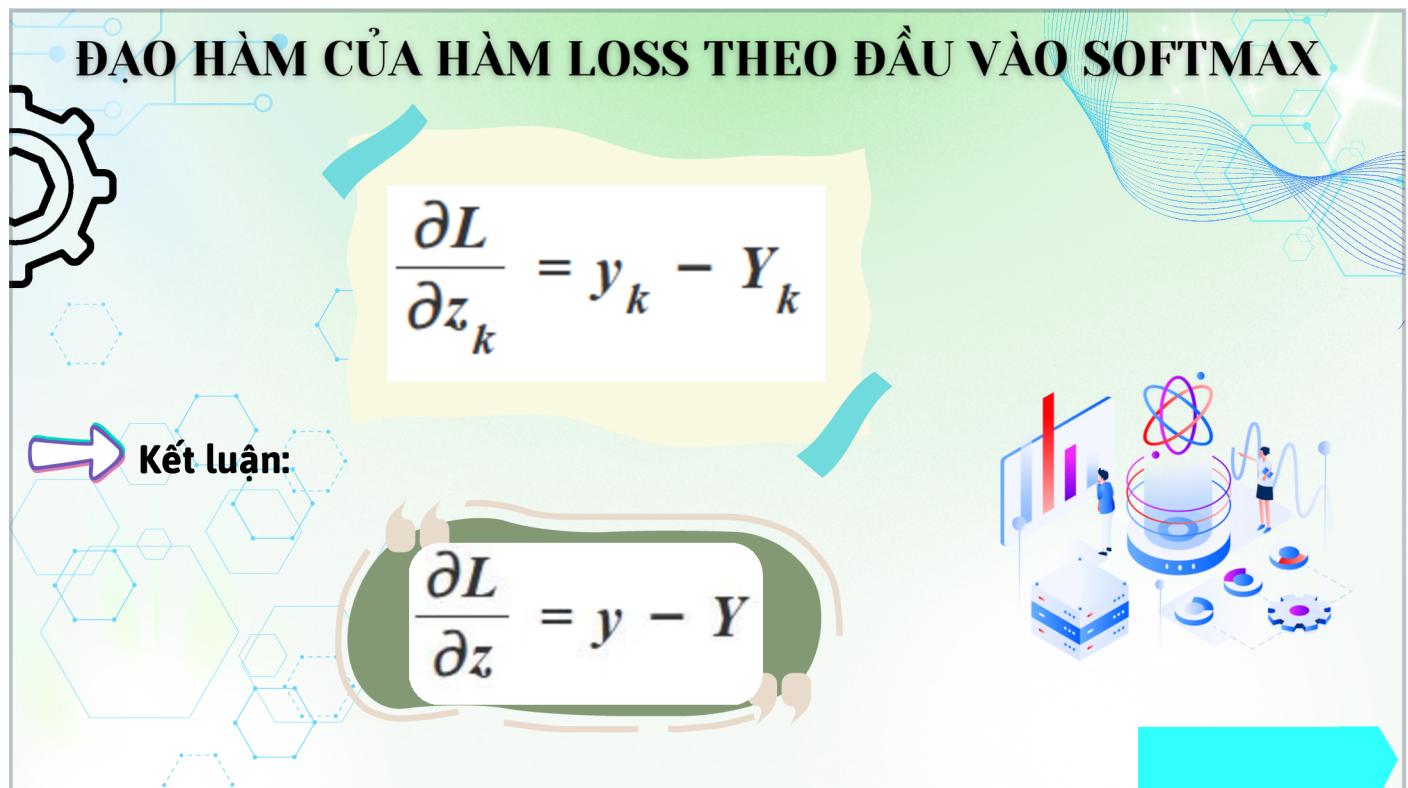


## • ĐẠO HÀM CỦA HÀM LOSS THEO ĐẦU VÀO SOFTMAX

$$\frac{\partial L}{\partial z_k} = y_k - Y_k$$

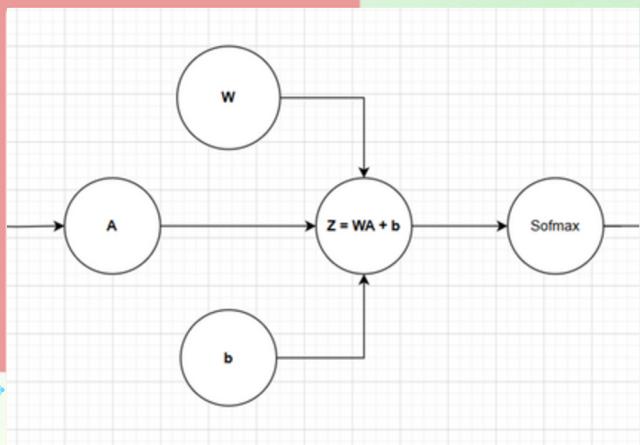
Kết luận:

$$\frac{\partial L}{\partial z} = y - Y$$



## BACKWARD QUA FULLY CONNECTED LAYER

Đầu ra của FC:  $Z = WA + b$



Cần tính đạo hàm của hàm loss theo  $W, b, A$ .



● Đạo hàm của hàm L theo A:

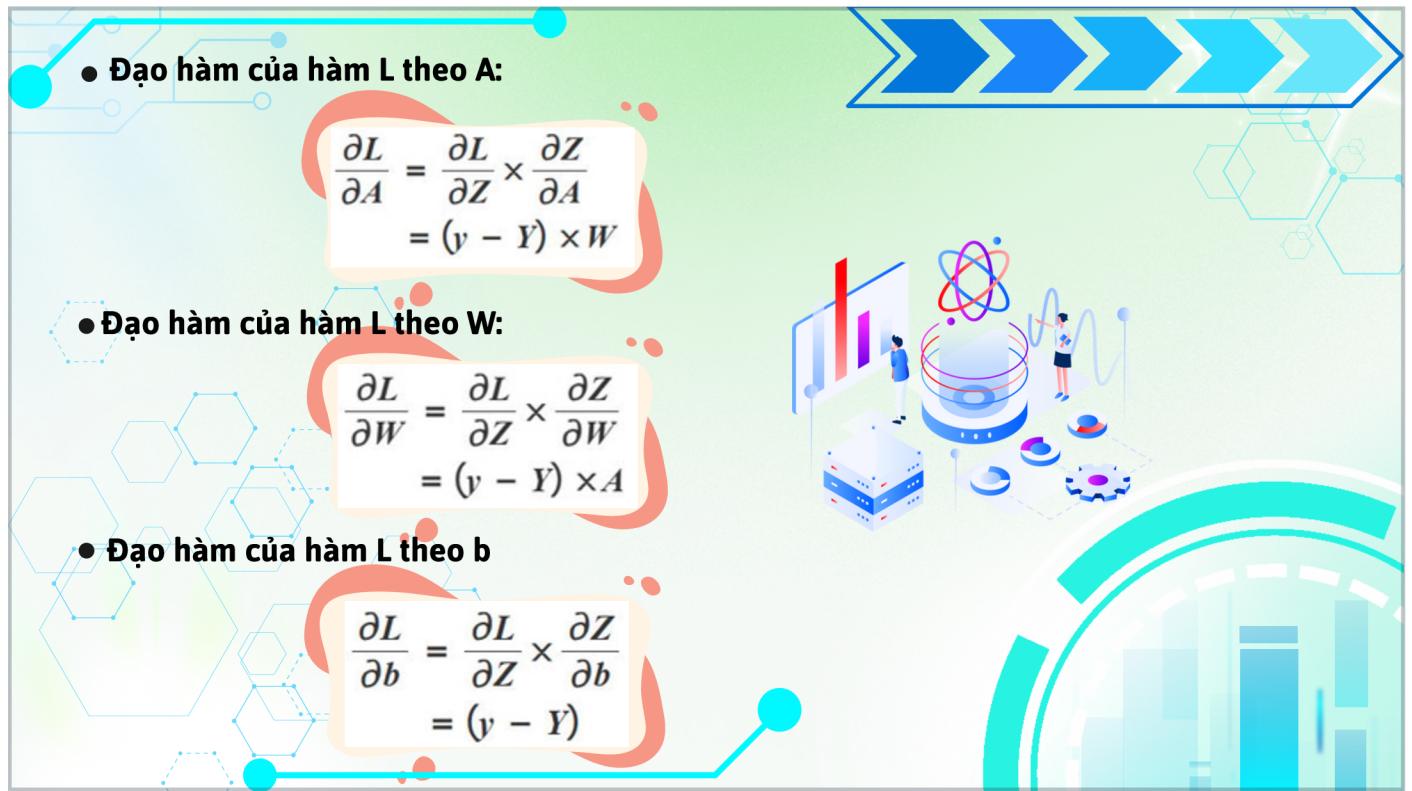
$$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial Z} \times \frac{\partial Z}{\partial A} \\ = (y - Y) \times W$$

● Đạo hàm của hàm L theo W:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Z} \times \frac{\partial Z}{\partial W} \\ = (y - Y) \times A$$

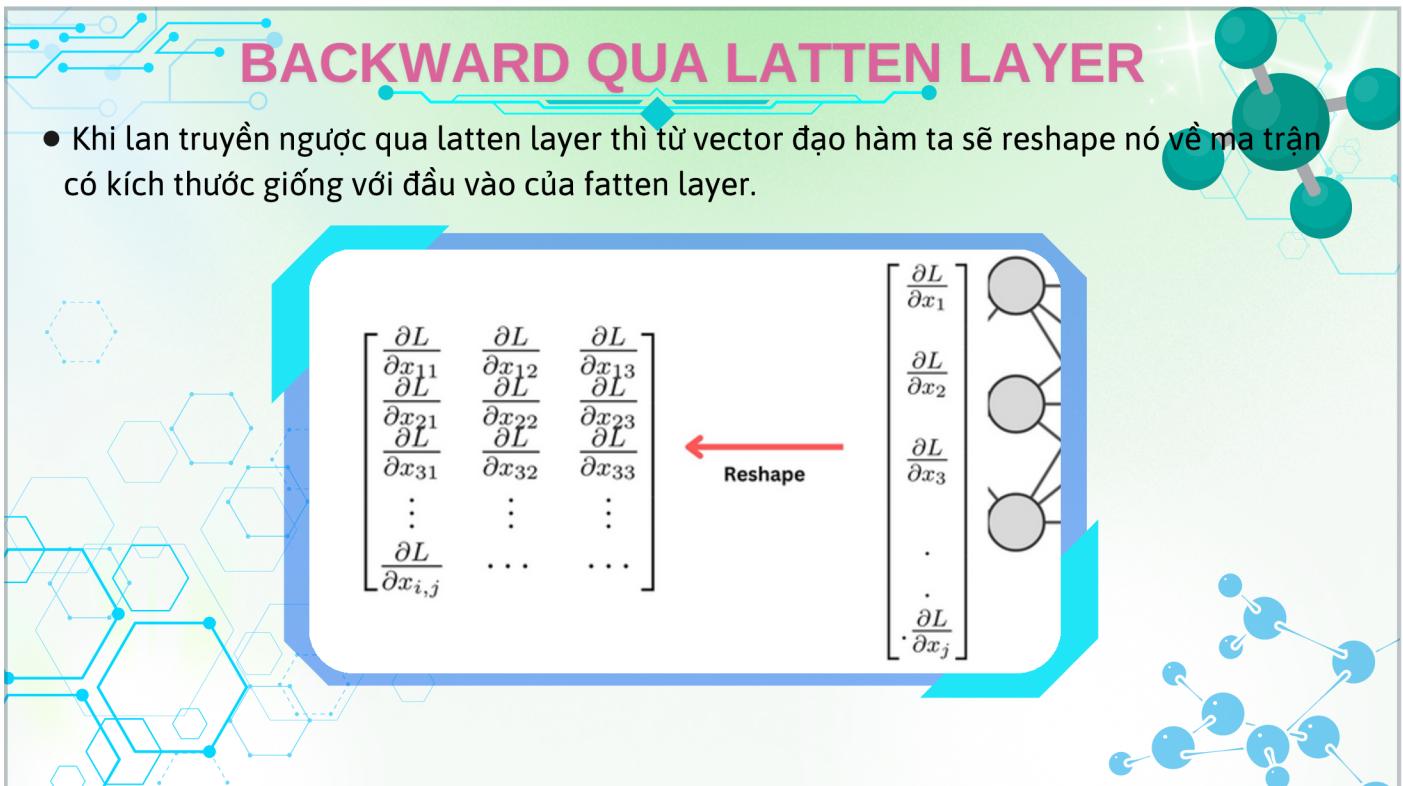
● Đạo hàm của hàm L theo b

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial Z} \times \frac{\partial Z}{\partial b} \\ = (y - Y)$$



## BACKWARD QUA LATTEN LAYER

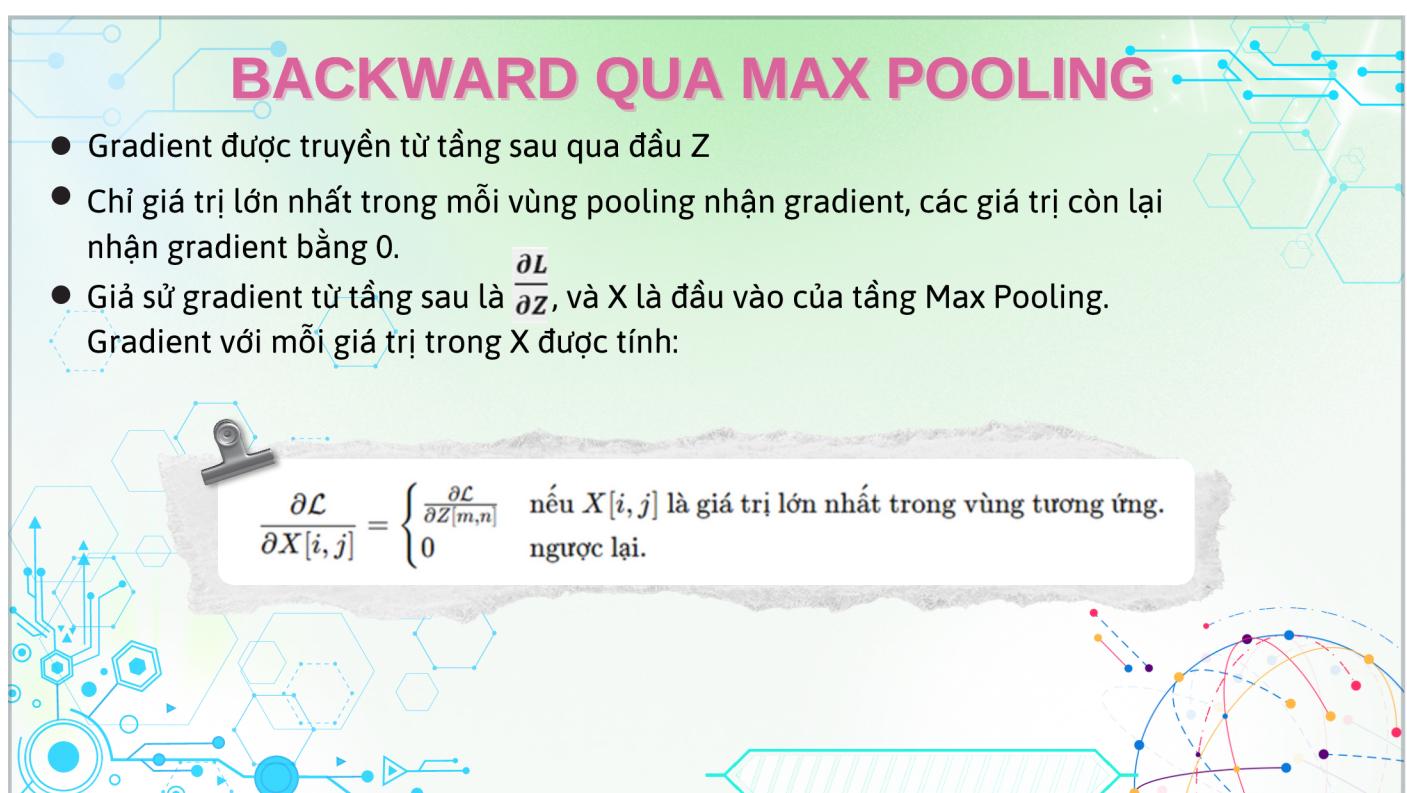
- Khi lan truyền ngược qua latten layer thì từ vector đạo hàm ta sẽ reshape nó về ma trận có kích thước giống với đầu vào của flatten layer.



# BACKWARD QUA MAX POOLING

- Gradient được truyền từ tầng sau qua đầu Z
- Chỉ giá trị lớn nhất trong mỗi vùng pooling nhận gradient, các giá trị còn lại nhận gradient bằng 0.
- Giả sử gradient từ tầng sau là  $\frac{\partial L}{\partial Z}$ , và X là đầu vào của tầng Max Pooling.  
Gradient với mỗi giá trị trong X được tính:

$$\frac{\partial \mathcal{L}}{\partial X[i, j]} = \begin{cases} \frac{\partial \mathcal{L}}{\partial Z[m, n]} & \text{nếu } X[i, j] \text{ là giá trị lớn nhất trong vùng tương ứng.} \\ 0 & \text{ngược lại.} \end{cases}$$



# BACKWARD QUA RELU

Đạo hàm của hàm Relu:  $f = \max(0, z)$

$$f'(x) = \begin{cases} 1 & \text{nếu } x > 0, \\ 0 & \text{nếu } x \leq 0. \end{cases}$$

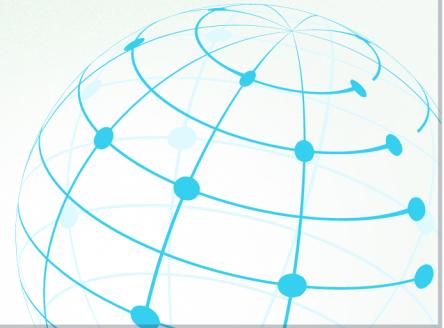
Quá trình Backpropagation:

- Gradient từ tầng sau  $\frac{\partial L}{\partial Z}$  được nhận tại đầu ra của ReLU
- Gradient tại đầu vào  $X$  của ReLU được tính bằng cách nhân gradient tầng sau với đạo hàm của ReLU:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Z} \cdot f'(X)$$

Công thức chi tiết:

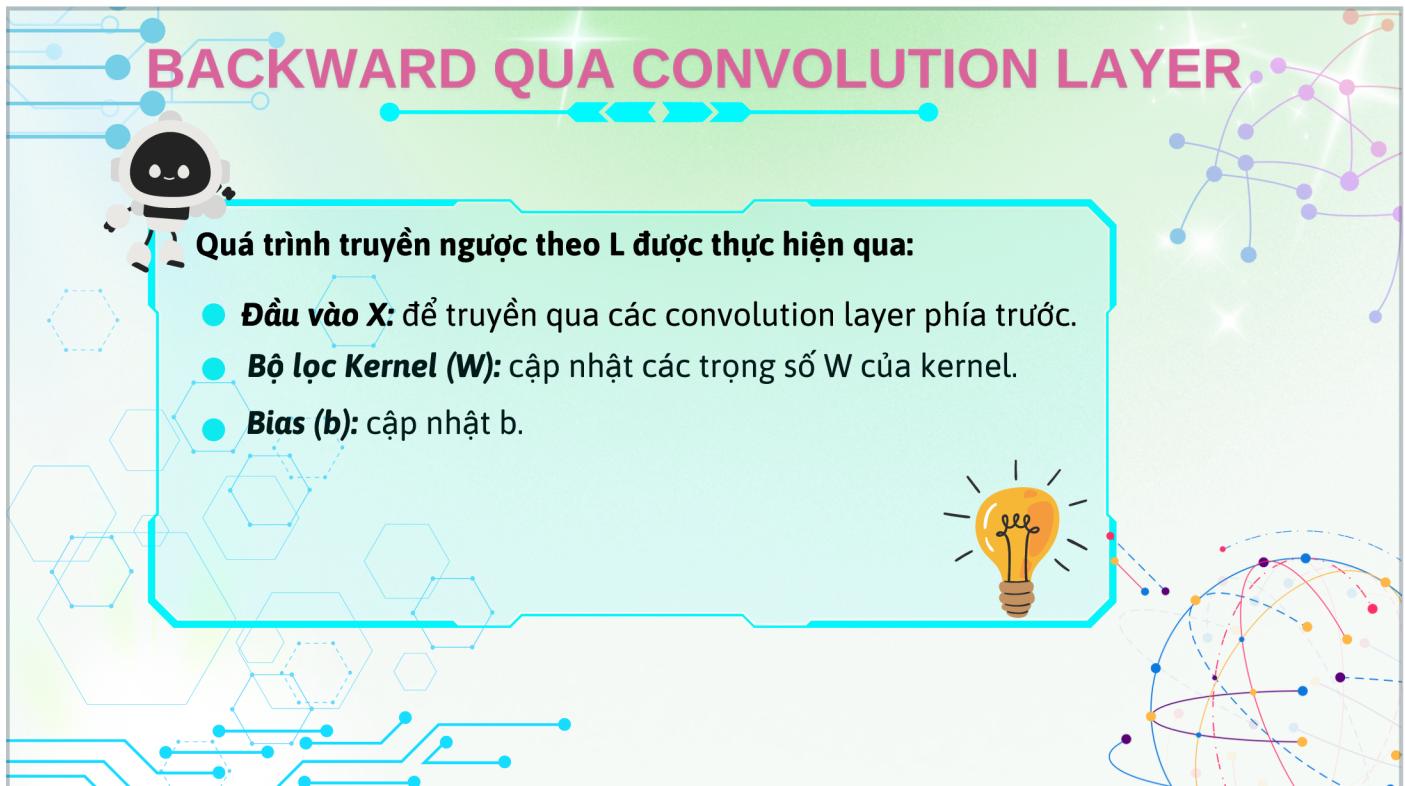
$$\frac{\partial L}{\partial X[i,j]} = \begin{cases} \frac{\partial L}{\partial Z[i,j]} & \text{nếu } X[i,j] > 0, \\ 0 & \text{nếu } X[i,j] \leq 0. \end{cases}$$



# BACKWARD QUA CONVOLUTION LAYER

Quá trình truyền ngược theo L được thực hiện qua:

- Đầu vào  $X$ : để truyền qua các convolution layer phía trước.
- Bộ lọc Kernel ( $W$ ): cập nhật các trọng số  $W$  của kernel.
- Bias ( $b$ ): cập nhật  $b$ .



## GRADIENT VỚI ĐẦU VÀO X ( $\frac{\partial L}{\partial X}$ )

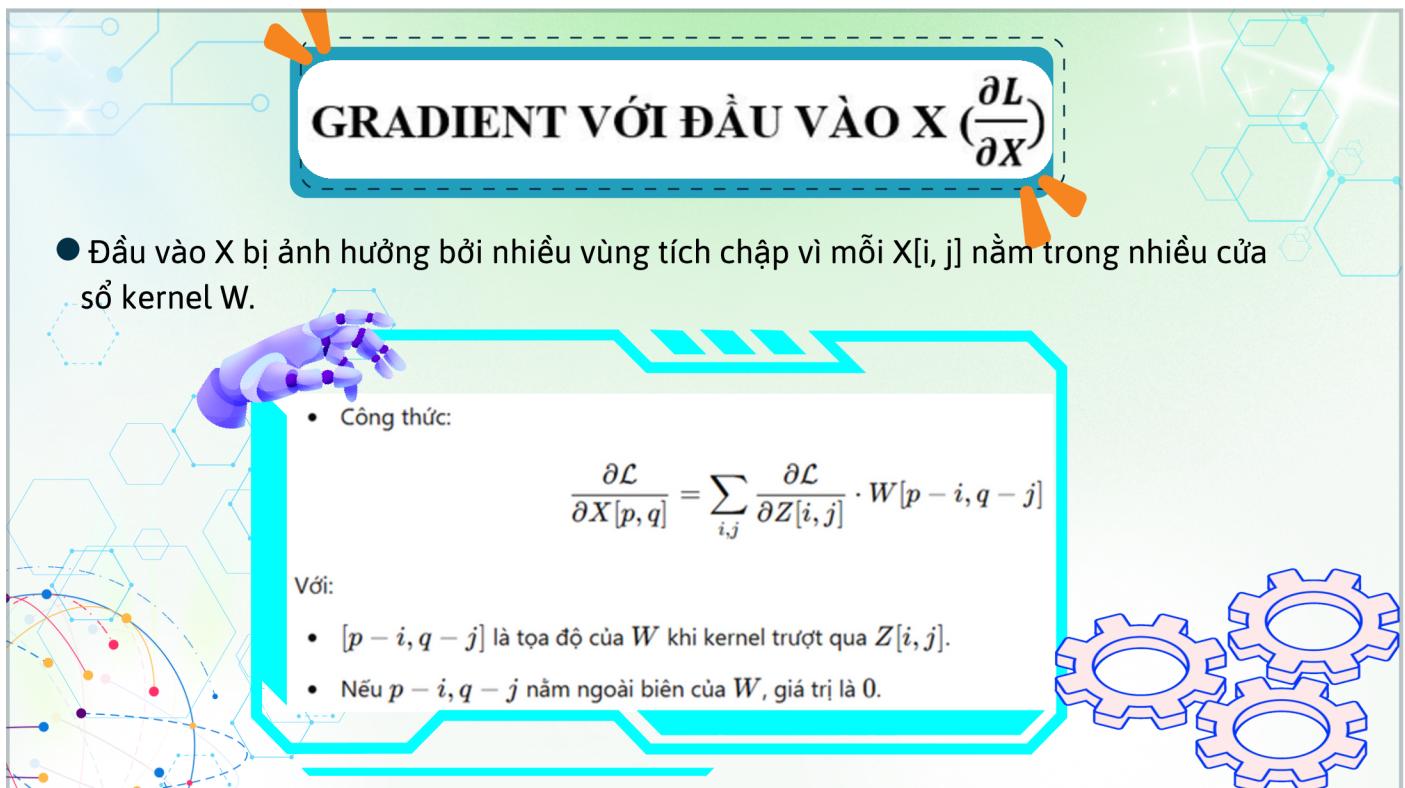
- Đầu vào X bị ảnh hưởng bởi nhiều vùng tích chập vì mỗi  $X[i, j]$  nằm trong nhiều cửa sổ kernel W.

- Công thức:

$$\frac{\partial \mathcal{L}}{\partial X[p, q]} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial Z[i, j]} \cdot W[p - i, q - j]$$

Với:

- [ $p - i, q - j$ ] là tọa độ của W khi kernel trượt qua  $Z[i, j]$ .
- Nếu  $p - i, q - j$  nằm ngoài biên của W, giá trị là 0.

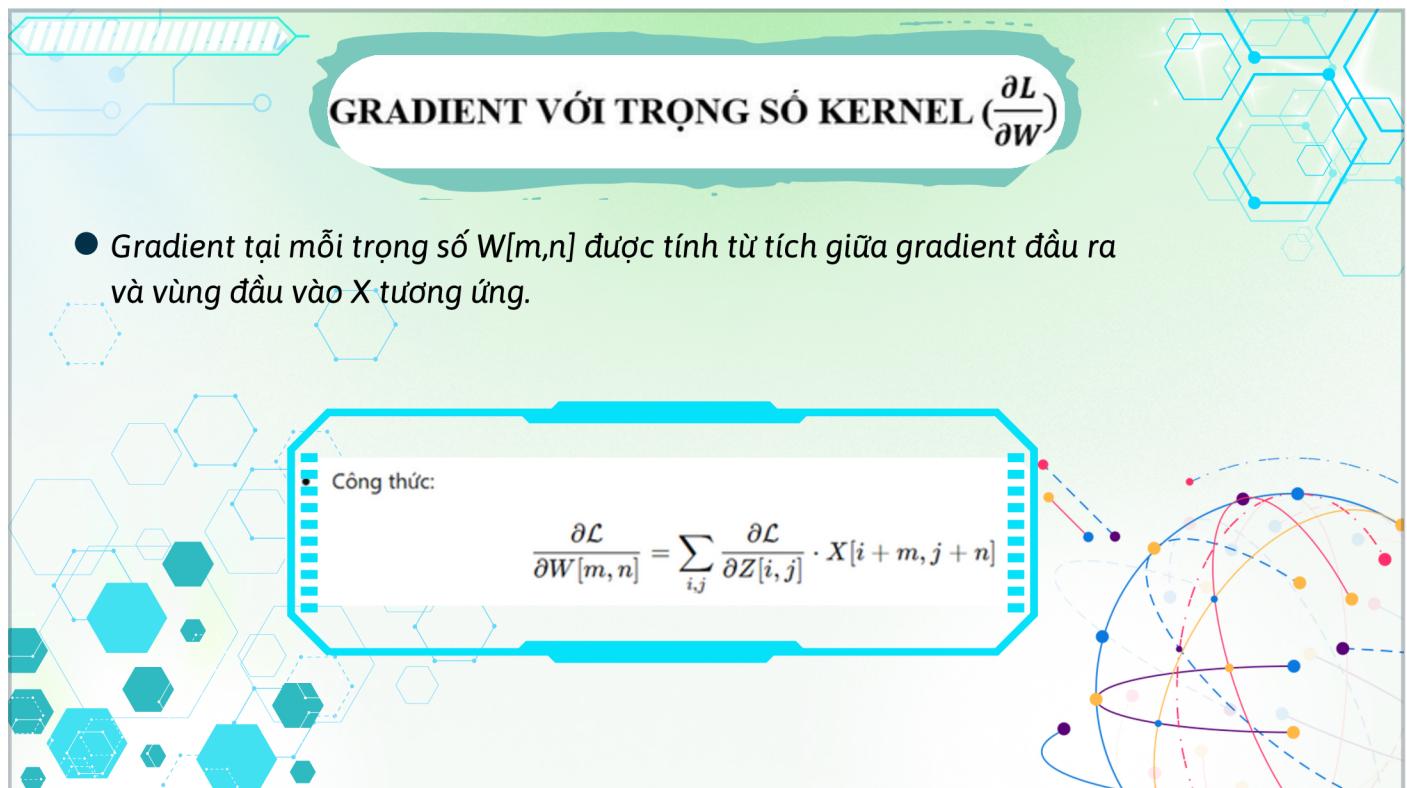


## GRADIENT VỚI TRỌNG SÓ KERNEL ( $\frac{\partial L}{\partial W}$ )

- Gradient tại mỗi trọng số  $W[m,n]$  được tính từ tích giữa gradient đầu ra và vùng đầu vào  $X$  tương ứng.

Công thức:

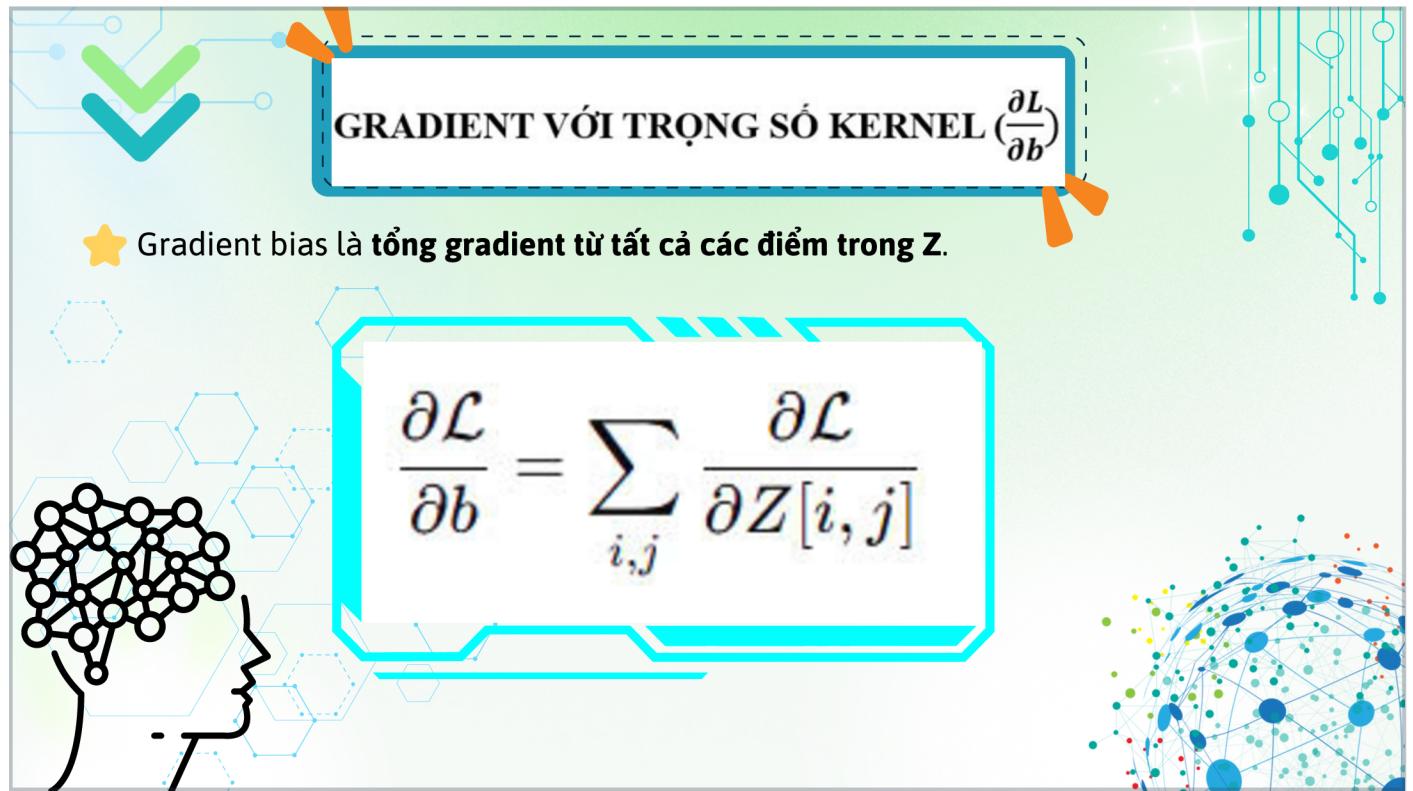
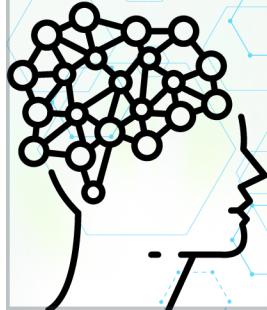
$$\frac{\partial \mathcal{L}}{\partial W[m, n]} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial Z[i, j]} \cdot X[i + m, j + n]$$



## GRADIENT VỚI TRỌNG SỐ KERNEL ( $\frac{\partial L}{\partial b}$ )

★ Gradient bias là **tổng gradient** từ tất cả các điểm trong Z.

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial Z[i, j]}$$



# CẬP NHẬT W, b

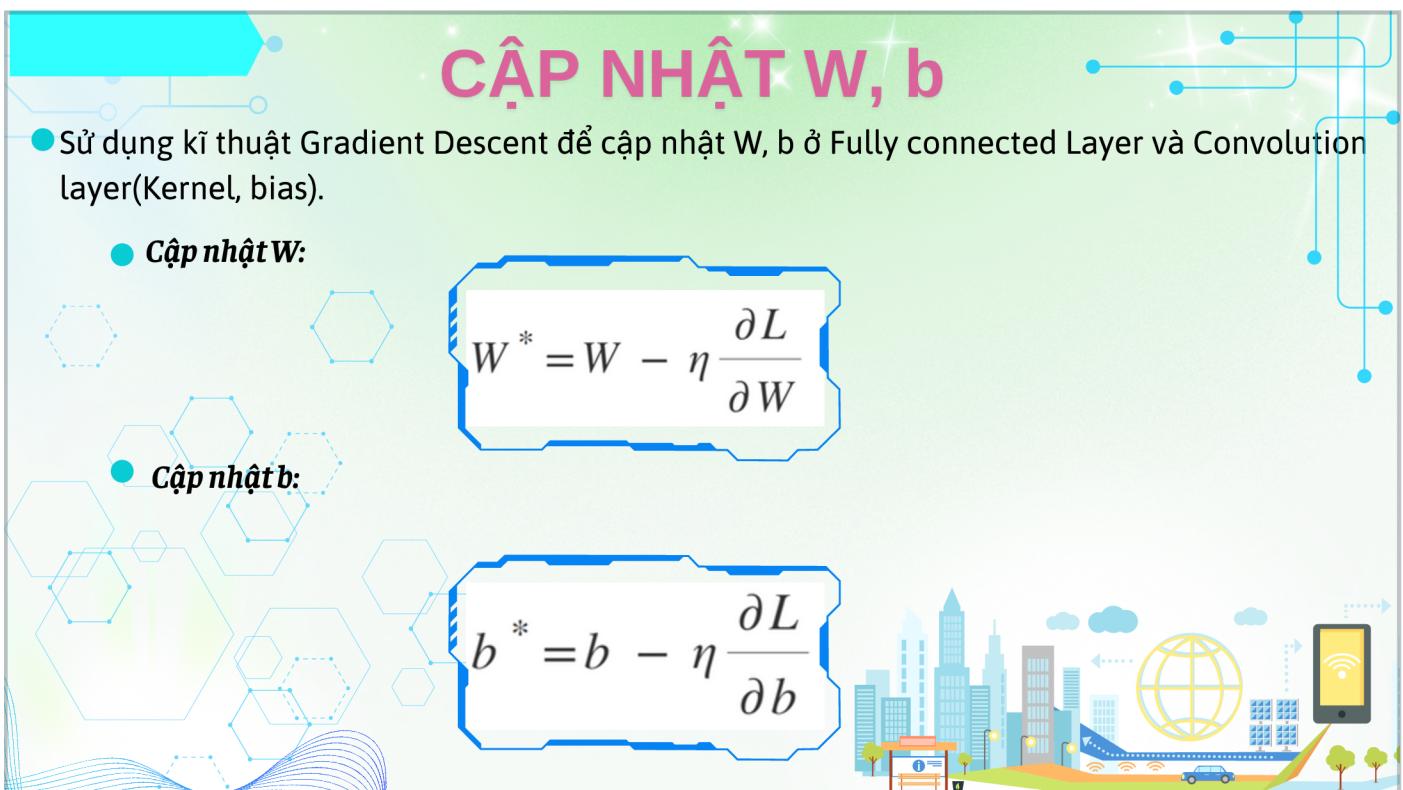
- Sử dụng kỹ thuật Gradient Descent để cập nhật W, b ở Fully connected Layer và Convolution layer(Kernel, bias).

**Cập nhật W:**

$$W^* = W - \eta \frac{\partial L}{\partial W}$$

**Cập nhật b:**

$$b^* = b - \eta \frac{\partial L}{\partial b}$$



# BATCH NORMALIZATION

- **Epoch:** là một lượt hoàn tất của quá trình huấn luyện trên toàn bộ dữ liệu huấn luyện.
- **Batch:** Khi dữ liệu quá lớn---> không thể đưa toàn bộ dữ liệu vào huấn luyện---> chia dữ liệu ban đầu thành các tập dữ liệu nhỏ hơn để training, tính toán lỗi, gradient, và cập nhật tham số---> Batch (Mini – batch).
- **Batch Normalization:** dùng để chuẩn hóa đầu ra của mỗi lớp trước khi đưa vào các lớp tiếp theo.

**Giả sử:** X là một batch đầu vào với  $m$  mẫu. Với mỗi feature map (kích thước  $d$ ), BN thực hiện các bước sau:

1. Tính giá trị trung bình(mean) và phương sai(variance):

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

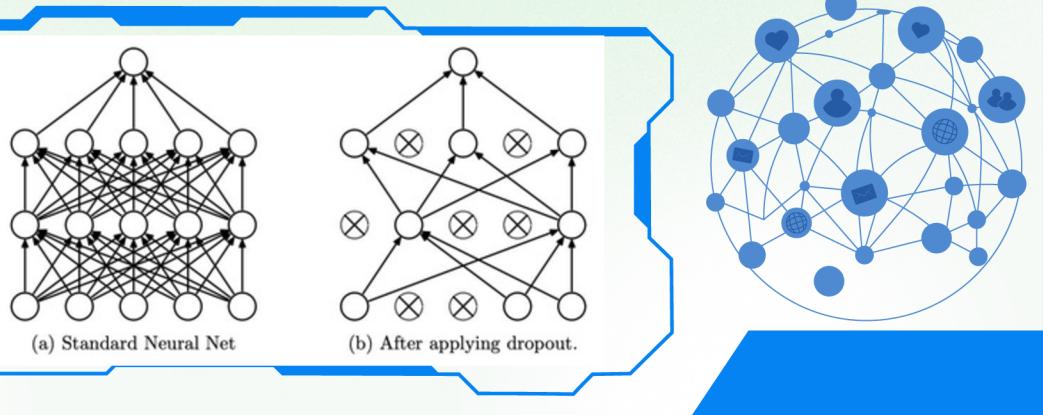
2. Chuẩn hóa đầu vào:

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

nhằm cải thiện tốc độ huấn luyện, sự ổn định, và hiệu suất tổng thể của mạng.

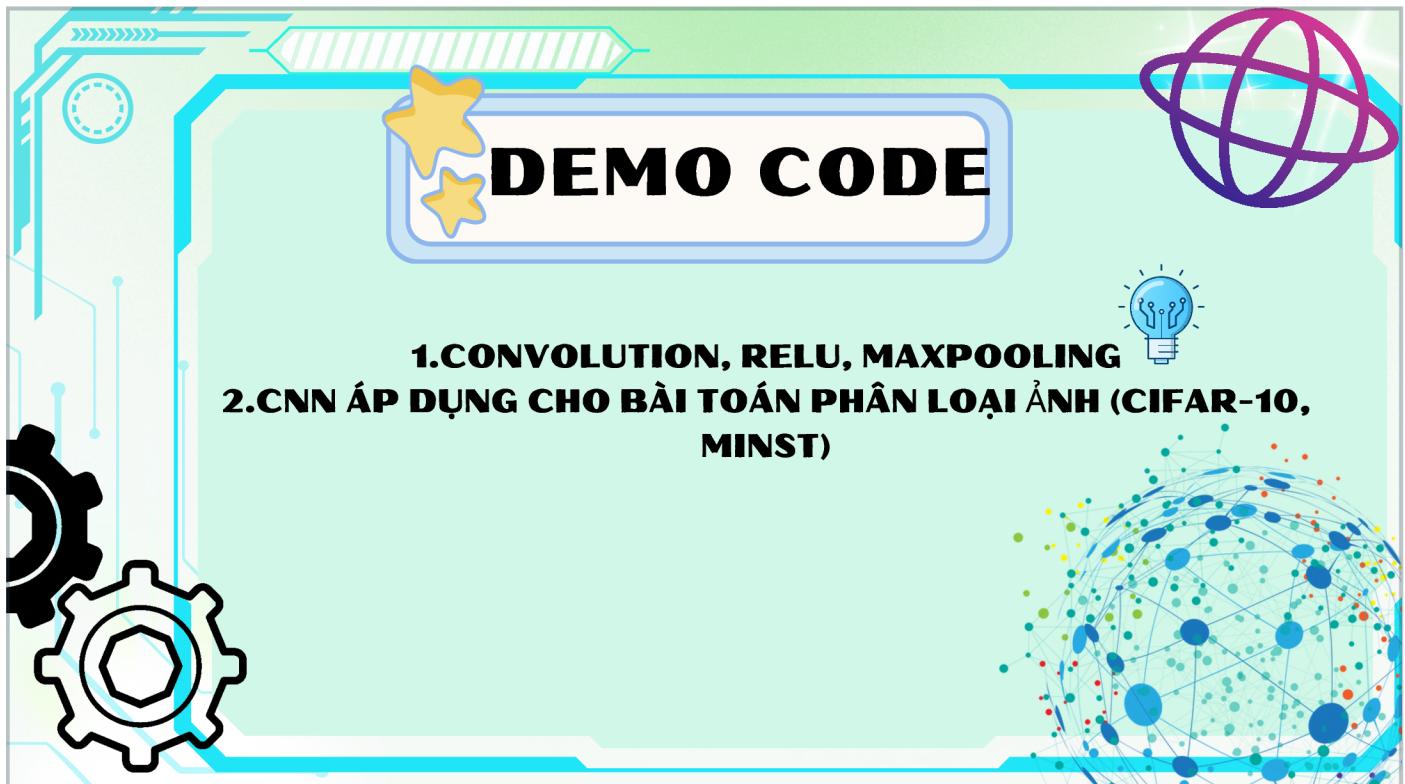
# KĨ THUẬT DROPOUT

- **Dropout** là một kỹ thuật regularization phổ biến trong huấn luyện mạng nơ-ron nhân tạo nhằm giảm thiểu tình trạng overfitting và cải thiện khả năng tổng quát hóa (generalization) của mô hình.
- **Cách hoạt động:** trong mỗi lần huấn luyện, dropout layer sẽ tiến hành tắt ngẫu nhiên một số neural với xác suất  $p$ . Các neural còn lại sẽ tham gia vào quá trình tính toán đầu ra và gradient.
- **Dropout** không áp dụng cho quá trình test.



# DEMO CODE

- 1. CONVOLUTION, RELU, MAXPOOLING**
- 2. CNN ÁP DỤNG CHO BÀI TOÁN PHÂN LOẠI ẢNH (CIFAR-10, MNIST)**



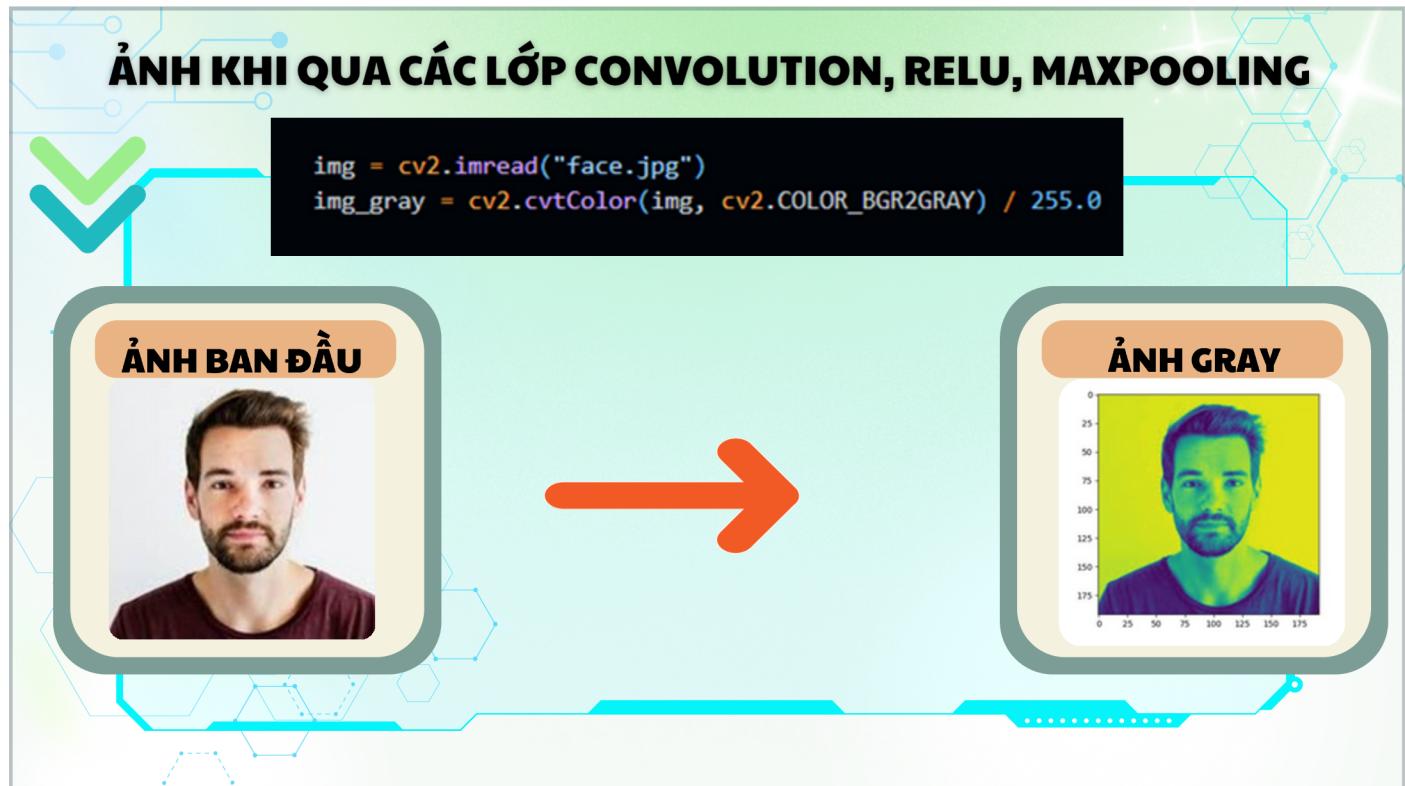
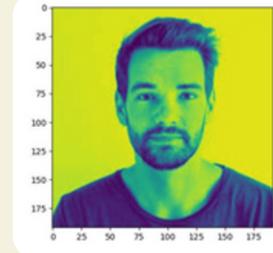
## ẢNH KHI QUA CÁC LỚP CONVOLUTION, RELU, MAXPOOLING

```
img = cv2.imread("face.jpg")
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) / 255.0
```

ẢNH BAN ĐẦU



ẢNH GRAY



# CONVOLUTION LAYER

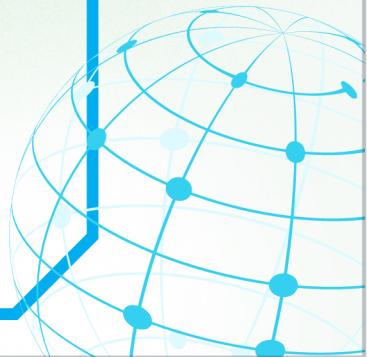
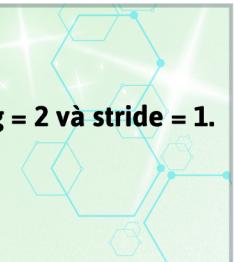
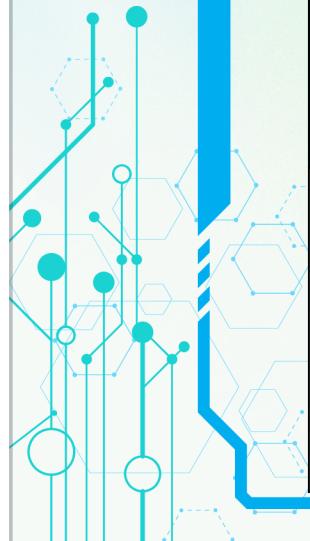
```
class Conv2D:  
    def __init__(self, num_filters, kernel_size, input, padding=0, stride=1):  
        self.num_filters = num_filters  
        self.kernel_size = kernel_size  
        self.input = input  
        self.padding = padding  
        self.stride = stride  
  
        # He initialization  
        self.kernels = np.random.randn(num_filters, kernel_size, kernel_size) * np.sqrt(2.0/input.shape[0])  
        self.biases = np.zeros(num_filters)  
  
    def forward(self):  
        self.input = np.pad(self.input, ((self.padding, self.padding),  
                                      (self.padding, self.padding)), 'constant')  
        output_height = int((self.input.shape[0] - self.kernel_size) / self.stride + 1)  
        output_width = int((self.input.shape[1] - self.kernel_size) / self.stride + 1)  
  
        self.output = np.zeros((output_height, output_width, self.num_filters))  
  
        for i in range(0, output_height, self.stride):  
            for j in range(0, output_width, self.stride):  
                input_slice = self.input[i:i+self.kernel_size, j:j+self.kernel_size]  
                for k in range(self.num_filters):  
                    self.output[i, j, k] = np.sum(input_slice * self.kernels[k]) + self.biases[k]  
  
        return self.output
```

# CONVOLUTION LAYER

- Khởi tạo lớp Conv2D với đầu vào là: 8 kernels, kernel\_size là 3, input là img\_gray, padding = 2 và stride = 1.

```
np.random.seed(24)
img_gray_conv2d = Conv2D(8, 3, img_gray, 2, 1).forward()

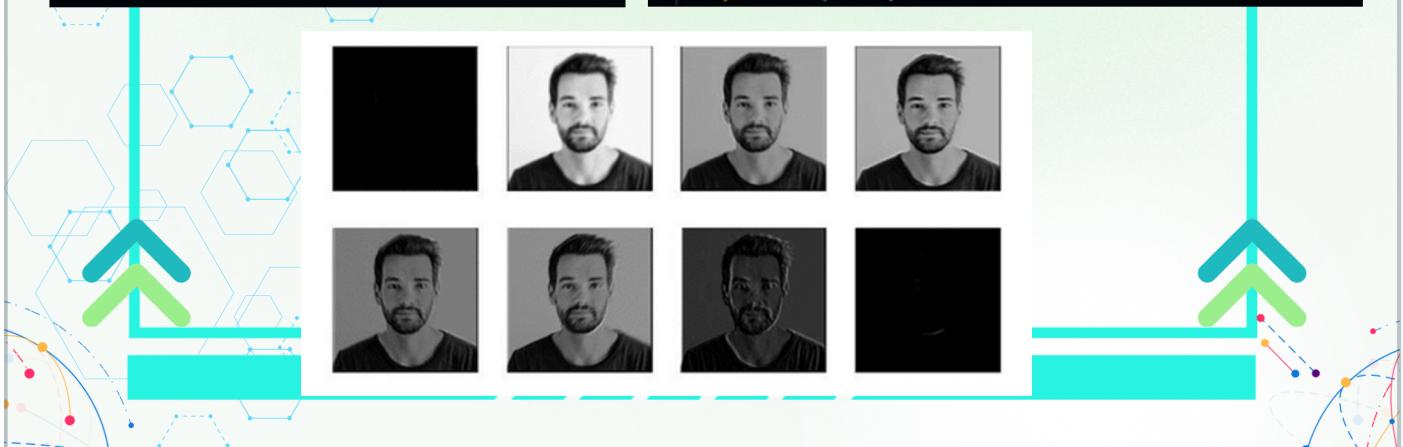
fig = plt.figure(figsize=(8, 10))
for i in range(8):
    plt.subplot(4, 2, i + 1)
    plt.imshow(img_gray_conv2d[:, :, i], cmap = 'gray')
    plt.axis('off')
```



# RELU

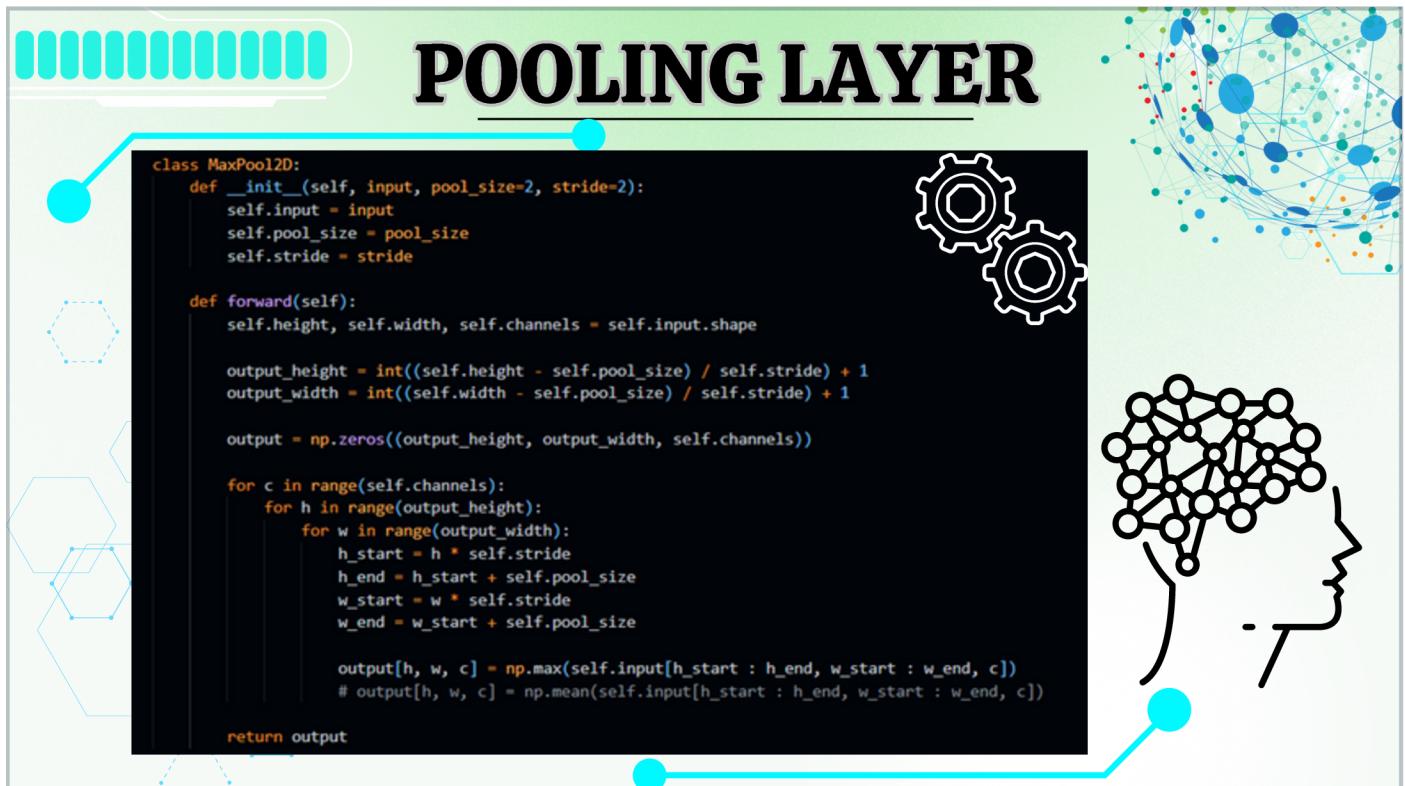
```
class ReLU:  
    def __init__(self, input):  
        self.input = input  
  
    def forward(self):  
        return np.maximum(0, self.input)
```

```
img_gray_con_relu = ReLU(img_gray_conv2d).forward()  
  
fig = plt.figure(figsize=(8, 4))  
for i in range(8):  
    plt.subplot(2, 4, i + 1)  
    plt.imshow(img_gray_con_relu[:, :, i], cmap = 'gray')  
    plt.axis('off')
```



# POOLING LAYER

```
class MaxPool2D:  
    def __init__(self, input, pool_size=2, stride=2):  
        self.input = input  
        self.pool_size = pool_size  
        self.stride = stride  
  
    def forward(self):  
        self.height, self.width, self.channels = self.input.shape  
  
        output_height = int((self.height - self.pool_size) / self.stride) + 1  
        output_width = int((self.width - self.pool_size) / self.stride) + 1  
  
        output = np.zeros((output_height, output_width, self.channels))  
  
        for c in range(self.channels):  
            for h in range(output_height):  
                for w in range(output_width):  
                    h_start = h * self.stride  
                    h_end = h_start + self.pool_size  
                    w_start = w * self.stride  
                    w_end = w_start + self.pool_size  
  
                    output[h, w, c] = np.max(self.input[h_start : h_end, w_start : w_end, c])  
                    # output[h, w, c] = np.mean(self.input[h_start : h_end, w_start : w_end, c])  
  
    return output
```



# POOLING LAYER

```
img_gray_c_r_Max = MaxPool2D(img_gray_con_relu).forward()

fig = plt.figure(figsize=(8, 4))
for i in range(8):
    plt.subplot(2, 4, i + 1)
    plt.imshow(img_gray_c_r_Max[:, :, i], cmap = 'gray')
    plt.axis('off')
```



```
img_gray_c_r_Ave = AvePool2D(img_gray_con_relu).forward()

fig = plt.figure(figsize=(8, 4))
for i in range(8):
    plt.subplot(2, 4, i + 1)
    plt.imshow(img_gray_c_r_Ave[:, :, i], cmap = 'gray')
    plt.axis('off')

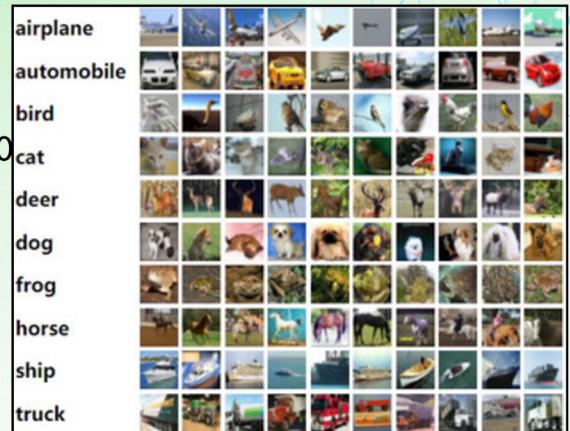
plt.savefig("img_gray_c_r_Ave.jpg")
```



# ỨNG DỤNG MẠNG CNN TRONG BÀI TOÁN PHÂN LOẠI HÌNH ẢNH VỚI DATASET CIFAR – 10

## Dataset CIFAR-10:

- Gồm 60000 ảnh màu kích thước 32 x 32.
- Ảnh được chia đều cho 10 lớp. Mỗi lớp gồm 6000 ảnh.
- 10 lớp gồm: airplane, automobile, bird, cat....
  - Trong 60000 ảnh có:
    - ◆ 50000 ảnh train.
    - ◆ 10000 ảnh test



Thường được sử dụng để đánh giá mô hình CNN, kiểm tra các thuật toán học sâu và học máy, sử dụng trong lĩnh vực thi giác máy tính.

# DATASET MNIST

- Bộ dữ liệu Minst: bao gồm các hình ảnh viết tay từ 0--> 9
- Số lượng mẫu:
  - ★ Tập huấn luyện: 60000
  - ★ Tập kiểm tra: 10000
- Kích thước ảnh: mỗi ảnh là 1 hình vuông grayscale 28 x28.
- Nhãn: từ 0 đến 9.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |





