Lesson 5 - Convolution

Convolution, Stride, Padding, and Multi-Channel Operations

Andrea Giardina
contact@andreagiardina.com
https://www.linkedin.com/in/agiardina

November 5, 2025

Today's topics

What you will learn

- What a 2D convolution is and why CNNs use it.
- How stride and padding change output size and information content.
- How multi-channel (RGB) convolution works and why each filter spans all channels.
- How to implement simple versions of these operations in NumPy.

Why Convolutional Networks?

Problem. If we feed an image to a fully connected network, we must flatten the 2D structure into a long 1D vector. This has two consequences:

1. The model forgets spatial locality. Two adjacent pixels become two distant entries in a vector with no notion of geometry.
2. The parameter count explodes with input size. A 32x32x3 image has 3072 inputs; a layer with 1000 hidden units needs 3,072,000 weights plus biases.

Understanding 2D Convolution on a Grayscale Image

Concept

Convolution combines a small matrix called the **kernel** with a local region of the input image. At each position, the kernel values are multiplied element-wise with the underlying pixel intensities, and the results are summed to produce one output pixel.

Key idea. Convolutions keep the 2D structure and apply the same small kernel at every location. This introduces two inductive biases:

- *Locality:* nearby pixels are processed together.
- *Weight sharing:* the same kernel is reused across the entire image.

How the Kernel Moves Across the Image (Base Case)

Idea

In the basic convolution operation, the kernel slides **one pixel at a time** over all possible positions where it still fits completely inside the image.

- The top-left corner of the kernel starts at the top-left of the image.
- The kernel then moves **one step to the right** at each iteration.
- When it reaches the end of the row, it moves **one step down** and starts again.
- Each valid position produces **one output pixel**.

Example

For an image of size 5×5 and a kernel of size 2×2, the kernel can move across $(5 - 2 + 1) \times (4 - 2 + 1) = 12$ positions.

Pencil & Paper: first convolution by hand

Given the input and kernel below, compute the valid 2x2 output using stride=1 and no padding.

$$I = \begin{bmatrix} 1 & 2 & 0 & 1 \\ 3 & 1 & 2 & 2 \\ 0 & 1 & 3 & 1 \\ 2 & 2 & 1 & 0 \end{bmatrix}, \quad K = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

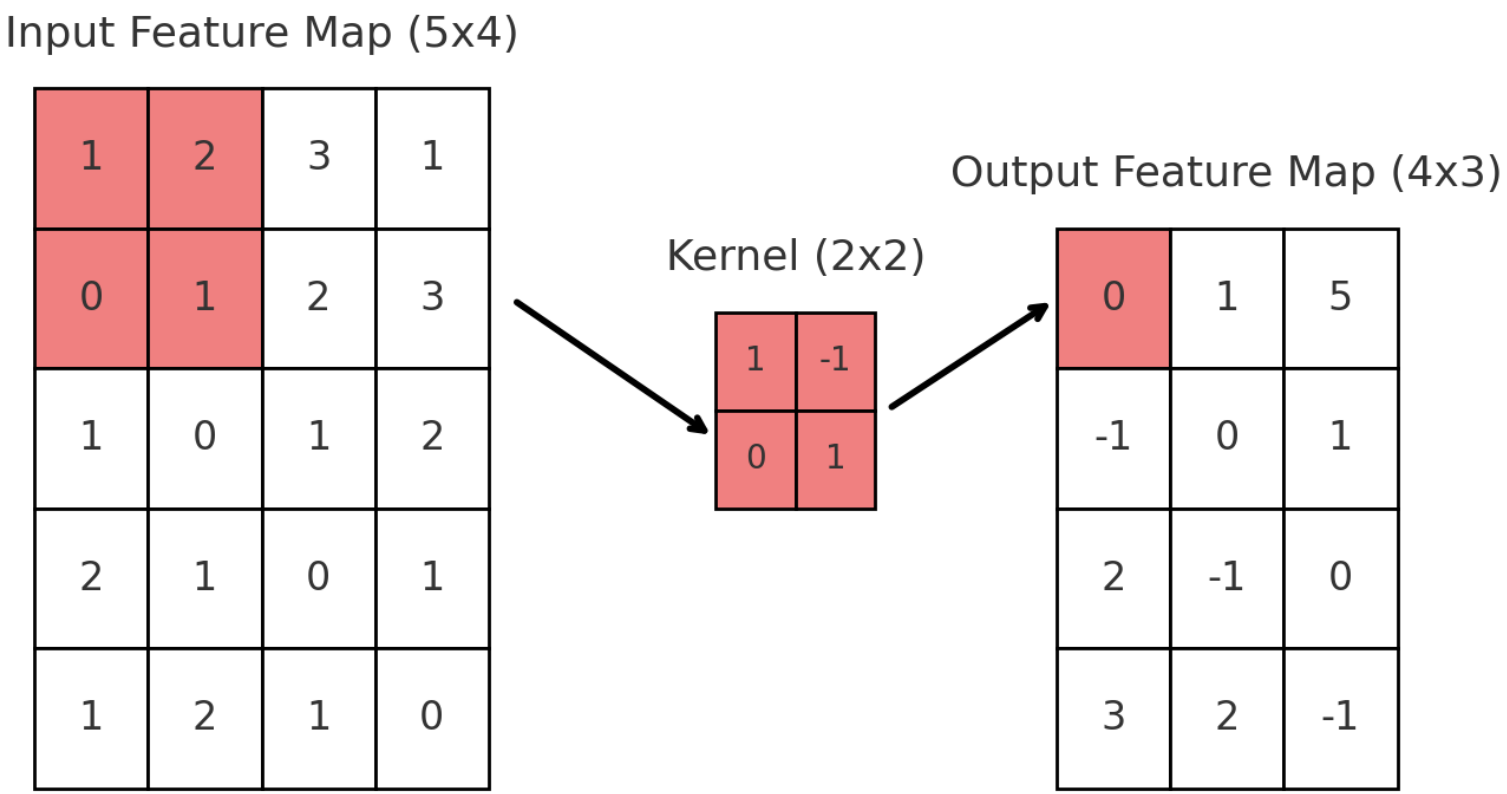
Hints: each output element is a sum of elementwise products between the 3x3 patch of I and K.

Solution

The valid output size is 2x2. The four values are:

$$\begin{bmatrix} -1 & 0 \\ -1 & 1 \end{bmatrix}$$

Convolution visual example



Question Time

1. Output size formula (base case)

Given an input image of size $W_{\text{in}} \times H_{\text{in}}$ and a kernel of size $K_w \times K_h$, **no padding, no stride.**

Question: What is the correct mathematical formula to compute the output size $(W_{\text{out}}, H_{\text{out}})$ in this base case?

Hint: The kernel must fit completely inside the image.

Question Time

1. Output size formula (base case)

Given an input image of size $W_{\text{in}} \times H_{\text{in}}$ and a kernel of size $K_w \times K_h$, **no padding, no stride.**

Question: What is the correct mathematical formula to compute the output size $(W_{\text{out}}, H_{\text{out}})$ in this base case?

Hint: The kernel must fit completely inside the image.

2. Pixel computation formula

For a pixel at position (i, j) in the output, **Question:** What is the correct expression for $O(i, j)$ in terms of the input image I and the kernel K?

Hint: Each output pixel is obtained by multiplying the overlapping region of I and K, and summing all the products.

Question Time – Solutions

1. Output size formula (base case)

For an input image of size $W_{\text{in}} \times H_{\text{in}}$ and a kernel of size $K_w \times K_h$ (no padding, no stride):

$$\boxed{\begin{aligned} W_{\text{out}} &= W_{\text{in}} - K_w + 1 \\ H_{\text{out}} &= H_{\text{in}} - K_h + 1 \end{aligned}}$$

- The kernel must fit entirely inside the input image.
- Each valid kernel position produces one output pixel.

Question Time – Solutions

1. Output size formula (base case)

For an input image of size $W_{\text{in}} \times H_{\text{in}}$ and a kernel of size $K_w \times K_h$ (no padding, no stride):

$$\boxed{\begin{aligned} W_{\text{out}} &= W_{\text{in}} - K_w + 1 \\ H_{\text{out}} &= H_{\text{in}} - K_h + 1 \end{aligned}}$$

- The kernel must fit entirely inside the input image.
- Each valid kernel position produces one output pixel.

2. Pixel computation formula

For each output coordinate (i, j), where $i \in [0, W_{\text{out}} - 1]$ and $j \in [0, H_{\text{out}} - 1]$:

$$\boxed{O(i,j) = \sum_{m=0}^{K_w-1} \sum_{n=0}^{K_h-1} I(i+m, j+n) \cdot K(m,n)}$$

Worked example: computing the first two outputs

We use the same I and K from pencil & paper exercise. For the top-left output:

$$\begin{aligned} y_{0,0} &= \sum_{m=0}^{2} \sum_{n=0}^{2} I_{m,n}\, K_{m,n} \\ &= 1 \cdot 1 + 2 \cdot 0 + 0 \cdot (-1) + 3 \cdot 1 + 1 \cdot 0 + 2 \cdot (-1) + 0 \cdot 1 + 1 \cdot 0 + 3 \cdot (-1) \\ &= -1 \end{aligned}$$

For the top-right output:

$$y_{0,1} = \sum_{m=0}^{2} \sum_{n=0}^{2} I_{m,n+1}\, K_{m,n} = 0$$

The full valid output for this example is

$$\begin{bmatrix} -1 & 0 \\ -1 & 1 \end{bmatrix}$$

Lab Time: the conv2d function

Implement a naive conv2d function with the following signature:

```
def conv2d(image, kernel):
    ...
    return out
```

NumPy implementation: naive 2D correlation

```
def conv2d(image, kernel):
    H, W = image.shape
    h, w = kernel.shape
    out = np.zeros((H - h + 1, W - w + 1))
    for i in range(out.shape[0]):
        for j in range(out.shape[1]):
            region = image[i:i+h, j:j+w]
            out[i, j] = np.sum(region * kernel)
    return out
```

Summary: How 2D Convolution Works (Base Case)

Step 1 — Setup

- The input is a 2D grayscale image of size $W_{\text{in}} \times H_{\text{in}}$.
- The kernel (or filter) is a small matrix of size $K_w \times K_h$.
- No padding and no stride are used.

Summary: How 2D Convolution Works (Base Case)

Step 1 — Setup

- The input is a 2D grayscale image of size $W_{\text{in}} \times H_{\text{in}}$.
- The kernel (or filter) is a small matrix of size $K_w \times K_h$.
- No padding and no stride are used.

Step 2 — Kernel movement

- The kernel slides one pixel at a time across all valid positions.
- It can only move where it fully fits inside the image.
- Each valid position corresponds to one pixel in the output.

Summary: How 2D Convolution Works (Base Case)

Step 3 — Computation at each position

- For each output coordinate (i, j):

$$O(i,j) = \sum_{m=0}^{K_w-1} \sum_{n=0}^{K_h-1} I(i+m, j+n) \cdot K(m,n)$$

- The kernel values $K(m,n)$ act as weights that emphasize certain patterns.
- The resulting value $O(i,j)$ represents how strongly the local image region matches the kernel.

Summary: How 2D Convolution Works (Base Case)

Step 3 — Computation at each position

- For each output coordinate (i, j):

$$O(i,j) = \sum_{m=0}^{K_w-1} \sum_{n=0}^{K_h-1} I(i+m, j+n) \cdot K(m,n)$$

- The kernel values $K(m,n)$ act as weights that emphasize certain patterns.
- The resulting value $O(i,j)$ represents how strongly the local image region matches the kernel.

Step 4 — Output size

$$\boxed{W_{\text{out}} = W_{\text{in}} - K_w + 1, \quad H_{\text{out}} = H_{\text{in}} - K_h + 1}$$

The output image is smaller because the kernel must stay entirely within the input.

How to Preserve the Image Size: Introducing Padding

The problem

In the base case, each convolution reduces the image size:

$$W_{\text{out}} = W_{\text{in}} - K_w + 1, \quad H_{\text{out}} = H_{\text{in}} - K_h + 1$$

This happens because the kernel cannot move beyond the image borders.

How to Preserve the Image Size: Introducing Padding

The problem

In the base case, each convolution reduces the image size:

$$W_{\text{out}} = W_{\text{in}} - K_w + 1, \quad H_{\text{out}} = H_{\text{in}} - K_h + 1$$

This happens because the kernel cannot move beyond the image borders.

The idea of padding

To keep the output the same size as the input, we artificially extend the image by adding a border of zeros around it.

- These added zeros are called **padding**.
- The kernel can now slide over the original borders as well.
- Each output pixel still represents a local weighted sum, but the borders now include zeros.

How to Preserve the Image Size: Introducing Padding

Question

How many pixels do we have to add on each side to preserve the original image size?

How to Preserve the Image Size: Introducing Padding

Question

How many pixels do we have to add on each side to preserve the original image size?

Mathematical effect

With padding P_w and P_h pixels added on each side:

$$\boxed{\begin{aligned} W_{\text{out}} &= W_{\text{in}} - K_w + 1 + 2P_w \\ H_{\text{out}} &= H_{\text{in}} - K_h + 1 + 2P_h \end{aligned}}$$

Choosing $P_w = \frac{K_w - 1}{2}$ and $P_h = \frac{K_h - 1}{2}$ preserves the original image size (**"same" convolution**). An odd-sized kernel is preferred because it allows symmetric padding, which preserves the original image size without shifting the output.

There Are Different Types of Padding

Motivation

Padding determines what values are placed around the image borders before applying the convolution.

- In the base case we used **zero-padding**, adding zeros around the image.
- However, zero is not always the best choice:
  - It can create dark borders or artificial edges.
  - It may not represent the true content near the boundaries.
- Depending on the problem, we may want to:
  - Keep the edge value constant.
  - Reflect or mirror the content near the borders.
  - Fill with a constant or periodic pattern.

There Are Different Types of Padding

Motivation

Padding determines what values are placed around the image borders before applying the convolution.

- In the base case we used **zero-padding**, adding zeros around the image.
- However, zero is not always the best choice:
  - It can create dark borders or artificial edges.
  - It may not represent the true content near the boundaries.
- Depending on the problem, we may want to:
  - Keep the edge value constant.
  - Reflect or mirror the content near the borders.
  - Fill with a constant or periodic pattern.

Idea

Different **padding strategies** control how the kernel behaves at the borders, balancing realism, continuity, and numerical stability.

Different Types of Padding and When to Use Them

Concept

Padding defines what values are added around the image borders so that the kernel can slide over edge pixels as well. The choice of padding affects both the appearance of the borders and the numerical stability of the operation.

Padding type	**Description**	**When to use it**
Zero padding	Fill with zeros around the image.	Default choice in CNNs; neutral borders.
Replicate padding	Repeat the edge pixel values.	Smooth images, avoid border fading.
Reflect padding	Mirror the image at the borders.	Natural textures, reduce discontinuities.
Constant padding	Fill with a chosen constant value c.	Synthetic data or mean-preserving cases.
Circular padding	Wrap pixels from the opposite edge.	Periodic signals or repeating textures.

Different Types of Padding and When to Use Them

Concept

Padding defines what values are added around the image borders so that the kernel can slide over edge pixels as well. The choice of padding affects both the appearance of the borders and the numerical stability of the operation.

Padding type	Description	When to use it
Zero padding	Fill with zeros around the image.	Default choice in CNNs; neutral borders.
Replicate padding	Repeat the edge pixel values.	Smooth images, avoid border fading.
Reflect padding	Mirror the image at the borders.	Natural textures, reduce discontinuities.
Constant padding	Fill with a chosen constant value c.	Synthetic data or mean-preserving cases.
Circular padding	Wrap pixels from the opposite edge.	Periodic signals or repeating textures.

Key takeaway

Zero-padding is the most common in deep learning, but reflect or replicate padding often produce smoother results for image processing and classical computer vision tasks.

Introducing Stride

Concept

The **stride** defines how far the kernel moves each time it slides across the image.

- In the basic case, the stride $S = 1$: the kernel moves **one pixel at a time**, both horizontally and vertically.
- If we increase the stride (e.g., $S = 2$), the kernel **jumps two pixels** each time it moves.
- A larger stride means:
  - fewer kernel positions,
  - a smaller output size,
  - faster computation but lower spatial resolution.

Stride

Question time

How the stride change W_{out} and H_{out}?

Stride

Question time

How the stride change W_{out} and H_{out}?

Mathematical effect

With stride S, padding P, and kernel size K:

$$W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} - K + 2P}{S} \right\rfloor + 1$$

$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} - K + 2P}{S} \right\rfloor + 1$$

Why We Use the Floor Function in the Stride Formula

The problem

When the kernel moves with stride S, it may not perfectly fit across the entire image. The last step could partially fall **outside the valid area**.

Why We Use the Floor Function in the Stride Formula

The problem

When the kernel moves with stride S, it may not perfectly fit across the entire image. The last step could partially fall **outside the valid area**.

Example

$$W_{\text{in}} = 8, \quad K = 3, \quad P = 0, \quad S = 2, \quad W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} - K + 2P}{S} \right\rfloor + 1$$

$$W_{\text{out}} = \frac{8-3}{2} + 1 = 3.5 \Rightarrow \lfloor 3.5 \rfloor = 3$$

Only 3 valid kernel positions exist: starting at pixel 0, 2, and 4. The fourth step (start = 6) would go out of bounds.

Why We Use the Floor Function in the Stride Formula

The problem

When the kernel moves with stride S, it may not perfectly fit across the entire image. The last step could partially fall **outside the valid area**.

Example

$$W_{\text{in}} = 8, \quad K = 3, \quad P = 0, \quad S = 2, \quad W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} - K + 2P}{S} \right\rfloor + 1$$

$$W_{\text{out}} = \frac{8-3}{2} + 1 = 3.5 \Rightarrow \lfloor 3.5 \rfloor = 3$$

Only 3 valid kernel positions exist: starting at pixel 0, 2, and 4. The fourth step (start = 6) would go out of bounds.

Key takeaway

The floor operation guarantees an integer output size and prevents the kernel from extending beyond the image borders.

Lab Time: the conv2d_sp function

Implement a naive conv2d_sp with support for stride and constant padding. The function should have the following signature:

```
def conv2d_sp(image, kernel, stride=1, padding=0):
    ...
    return out
```

NumPy naive implementation: stride and padding

```
def conv2d_sp(image, kernel, stride=1, padding=0):
    if padding > 0:
        image = np.pad(image, ((padding,padding),(padding,padding)),
                       mode='constant')
    H, W = image.shape
    h, w = kernel.shape
    out_H = (H - h)//stride + 1
    out_W = (W - w)//stride + 1
    out = np.zeros((out_H, out_W))
    for i in range(out_H):
        for j in range(out_W):
            region = image[i*stride:i*stride+h, j*stride:j*stride+w]
            out[i, j] = np.sum(region * kernel)
    return out
```

Interpreting filters

- Small 3x3 or 5x5 filters can detect edges, corners, and simple textures.
- A vertical edge detector has large positive weights on the left column, large negative weights on the right column, and near-zero center weights.
- A blur filter averages neighbors; a sharpen filter adds the high-frequency details extracted by subtracting a blurred version from the original.

Sobel Filters: Edge Detection

- Sobel filters are classical **edge detection operators**.
- They estimate the **intensity gradient** of an image.
- Two 3×3 kernels are used:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

- G_x detects **vertical edges**.
- G_y detects **horizontal edges**.

Lab time: apply and compare filters

Tasks:

- Load a grayscale image.
- Apply blur, sharpen, Sobel X, Sobel Y filters.

Filters in Classical Vision

- Before deep learning, filters like Sobel were used to extract edges and shapes.
- These filters were **manually designed**, not learned.
- They acted as fixed feature extractors before a classifier (e.g., SVM, k-NN).
- Limitation: they do not adapt to data.

In Deep Learning: Learned Filters

- In modern CNNs, filters are **learned automatically** through backpropagation.
- Early convolutional layers learn patterns similar to Sobel filters:
  - Edge detectors
  - Orientation-sensitive filters
  - Color or contrast gradients
- Intermediate layers learn textures and corners.
- Deep layers learn abstract semantic concepts (e.g., eyes, wheels, faces).

Sobel-like Patterns in CNNs

- When visualizing the first layer of a CNN (e.g., trained on ImageNet):
  - We often find filters resembling Sobel operators.
  - These emerge naturally from gradient-based learning.
- Thus, CNNs **rediscover Sobel-like filters** as optimal primitives for building higher-level representations.

Key Idea

Early CNN filters $\approx$ learned Sobel filters.

Summary

- Sobel filters compute image gradients $\Rightarrow$ detect edges.
- Classical: hand-crafted, non-adaptive.
- Deep Learning: filters are learned automatically.
- First CNN layers often learn edge detectors similar to Sobel filters.

From Grayscale to RGB Images

Grayscale vs. RGB

- A **grayscale image** has only one channel: intensity values.
- An **RGB image** has three channels:

$$I = [I_R, I_G, I_B]$$

representing Red, Green, and Blue components.

- Each channel is a separate 2D matrix of the same size.

Example

For an image of size $H \times W$:

$$\text{Grayscale: } I \in \mathbb{R}^{H \times W} \qquad \text{RGB: } I \in \mathbb{R}^{H \times W \times 3}$$

2D Convolution on RGB Images

Key difference from grayscale

In grayscale convolution, the kernel slides over a single 2D matrix. For RGB, the kernel must process all **three channels simultaneously**.

2D Convolution on RGB Images

Key difference from grayscale

In grayscale convolution, the kernel slides over a single 2D matrix. For RGB, the kernel must process all **three channels simultaneously**.

- Each kernel has three 2D matrices — one per channel:

$$K = [K_R, K_G, K_B]$$

- Each kernel position computes a weighted sum across all channels:

$$O(i,j) = \sum_{c \in \{R,G,B\}} \sum_{m,n} I_c(i+m, j+n) \cdot K_c(m,n)$$

- The result $O(i,j)$ is a single scalar $\rightarrow$ one output channel.

2D Convolution on RGB Images

Key difference from grayscale

In grayscale convolution, the kernel slides over a single 2D matrix. For RGB, the kernel must process all **three channels simultaneously**.

- Each kernel has three 2D matrices — one per channel:

$$K = [K_R, K_G, K_B]$$

- Each kernel position computes a weighted sum across all channels:

$$O(i,j) = \sum_{c \in \{R,G,B\}} \sum_{m,n} I_c(i+m, j+n) \cdot K_c(m,n)$$

- The result $O(i,j)$ is a single scalar $\rightarrow$ one output channel.

Shape

If I is $(H, W, 3)$ and K is $(K_h, K_w, 3)$ $\rightarrow$ the output of one kernel is $(H_{\text{out}}, W_{\text{out}}, 1)$.

Compact vs. Explicit Formula for RGB Convolution

Compact form (standard notation)

$$O(i,j) = \sum_{c \in \{R,G,B\}} \sum_{m,n} I_c(i+m, j+n) \cdot K_c(m,n)$$

- $\sum_{m,n}$ means a 2D spatial sum (height $\times$ width).
- $\sum_c$ adds the contributions from all color channels.

Compact vs. Explicit Formula for RGB Convolution

Compact form (standard notation)

$$O(i,j) = \sum_{c \in \{R,G,B\}} \sum_{m,n} I_c(i+m, j+n) \cdot K_c(m,n)$$

- $\sum_{m,n}$ means a 2D spatial sum (height $\times$ width).
- $\sum_c$ adds the contributions from all color channels.

Fully explicit form

$$O(i,j) = \sum_{c \in \{R,G,B\}} \sum_{m=0}^{K_h-1} \sum_{n=0}^{K_w-1} I_c(i+m, j+n) \cdot K_c(m,n)$$

- Three nested sums: over height (m), width (n), and channels (c).
- Each term multiplies corresponding pixels and kernel weights.
- The result is one scalar value $O(i,j)$ in the output feature map.

Lab Time: the conv2d_rgb function

Implement a naive conv2d_rgb with support for rgb channels, stride and constant padding. The function should have the following signature:

```
def conv2d_rgb(image, kernel, stride=1, padding=0):
    ...
    return out
```

Hint 1: small changes to conv2d_sp are required to transform conv2d_sp into conv2d_rgb

Hint 2: the kernel should have shape $(h, w, 3)$

NumPy naive implementation

```
def conv2d_rgb(image, kernel, stride=1, padding=0):
    if padding > 0:
        image = np.pad(image, ((padding, padding),
                               (padding, padding), (0, 0)),
                       mode='constant')

    s = stride
    H, W, C = image.shape
    h, w, _ = kernel.shape
    out_H = (H - h) // s + 1
    out_W = (W - w) // s + 1
    out = np.zeros((out_H, out_W))

    for i in range(out_H):
        for j in range(out_W):
            region = image[i*s:i*s+h, j*s:j*s+w, :]
            out[i, j] = np.sum(region * kernel)

    return out
```

Multiple Filters and Feature Maps

Concept

In CNNs, we use multiple filters, each with its own set of 3-channel weights.

- Each filter learns to detect a specific pattern (e.g., edges, colors, textures).
- Each produces a separate output matrix called a **feature map**.
- Stacking all feature maps gives the output volume of the convolutional layer.

Multiple Filters and Feature Maps

Concept

In CNNs, we use multiple filters, each with its own set of 3-channel weights.

- Each filter learns to detect a specific pattern (e.g., edges, colors, textures).
- Each produces a separate output matrix called a **feature map**.
- Stacking all feature maps gives the output volume of the convolutional layer.

Shape of the output tensor

If the input is $(H, W, 3)$ and we use N_f filters, each $(K_h, K_w, 3)$:

$$\text{Output shape: } (H_{\text{out}}, W_{\text{out}}, N_f)$$

Multiple Filters and Feature Maps

Concept

In CNNs, we use multiple filters, each with its own set of 3-channel weights.

- Each filter learns to detect a specific pattern (e.g., edges, colors, textures).
- Each produces a separate output matrix called a **feature map**.
- Stacking all feature maps gives the output volume of the convolutional layer.

Shape of the output tensor

If the input is $(H, W, 3)$ and we use N_f filters, each $(K_h, K_w, 3)$:

$$\text{Output shape: } (H_{\text{out}}, W_{\text{out}}, N_f)$$

Lab Time: the conv2d_rgb_bank function

Extend the convd2d_rgb to support multiple filters. The kernel should have shape $(K, h, w, 3)$, where K is the number of filters.

```
def conv2d_rgb_bank(image, kernel, stride=1, padding=0):
    ...
    .... out[i, j, k] = ...
    return out
```

Hint: one more for loop is required

NumPy naive implementation

```
def conv2d_rgb_bank(image, kernels, stride=1, padding=0):
    if padding > 0:
        image = np.pad(image, ((padding, padding),
                               (padding, padding), (0, 0)),
                       mode='constant')

    s = stride
    H, W, C = image.shape
    K, h, w, Ck = kernels.shape

    out_H = (H - h) // s + 1
    out_W = (W - w) // s + 1
    out = np.zeros((out_H, out_W, K), dtype=np.float64)

    for k in range(K):
        kernel = kernels[k]  # (h, w, 3)
        for i in range(out_H):
            for j in range(out_W):
                region = image[i*s:i*s+h, j*s:j*s+w, :]  # (h, w, 3)
                out[i, j, k] = np.sum(region * kernel)
    return out
```

Summary: RGB Convolution

- RGB images have three input channels: Red, Green, Blue.
- Each convolutional filter has **three corresponding kernels**.
- The outputs from all channels are summed to produce one feature map.
- Using multiple filters $\rightarrow$ multiple output channels (feature maps).

Summary: RGB Convolution

- RGB images have three input channels: Red, Green, Blue.
- Each convolutional filter has **three corresponding kernels**.
- The outputs from all channels are summed to produce one feature map.
- Using multiple filters $\rightarrow$ multiple output channels (feature maps).

Shapes at a glance

$$I \in \mathbb{R}^{H \times W \times 3}$$

$$K \in \mathbb{R}^{K_h \times K_w \times 3}$$

$$O \in \mathbb{R}^{H_{\text{out}} \times W_{\text{out}} \times N_f}$$

Summary: RGB Convolution

- RGB images have three input channels: Red, Green, Blue.
- Each convolutional filter has **three corresponding kernels**.
- The outputs from all channels are summed to produce one feature map.
- Using multiple filters $\rightarrow$ multiple output channels (feature maps).

Shapes at a glance

$$I \in \mathbb{R}^{H \times W \times 3}$$

$$K \in \mathbb{R}^{K_h \times K_w \times 3}$$

$$O \in \mathbb{R}^{H_{\text{out}} \times W_{\text{out}} \times N_f}$$

Intuition

The convolution learns to combine color information across channels to detect meaningful features — not just edges, but also color patterns.

Pencil & Paper: parameter counting challenge

Question

Compute and compare:

- MLP with input 32x32x3 and 10 outputs.
- CNN layer with 6 filters of size 5x5x3.

Include biases.

Pencil & Paper: parameter counting challenge

Question

Compute and compare:

- MLP with input 32x32x3 and 10 outputs.
- CNN layer with 6 filters of size 5x5x3.

Include biases.

Solution

MLP: 3072 x 10 weights + 10 biases = 30,730 parameters.
CNN layer: 6 x (5 x 5 x 3) weights + 6 biases = 456 parameters.
The difference comes from local connectivity and weight sharing.

Thanks!