Deep Learning from First Principles

Lesson 6 - Convolution and Pooling with im2col

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

Learning goals

- Understand how the `im2col` trick converts convolution into a matrix multiplication (GEMM).
- Separate responsibilities between `Conv2d` (semantics) and `im2col` (geometry).
- Build forward-only `Conv2d` step-by-step without blocking a future backward.
- Handle padding, stride, multi-channel inputs, multiple filters (feature maps), and batch.

What is GEMM?

Definition

GEMM stands for **GEneral Matrix–Matrix Multiplication**. It is the fundamental linear algebra operation used throughout deep learning frameworks.

Mathematical Form

$$C = \alpha AB + \beta C$$

- A: matrix of shape $(m \times k)$
- B: matrix of shape $(k \times n)$
- C: output matrix $(m \times n)$
- α, β: scalar coefficients

What is GEMM?

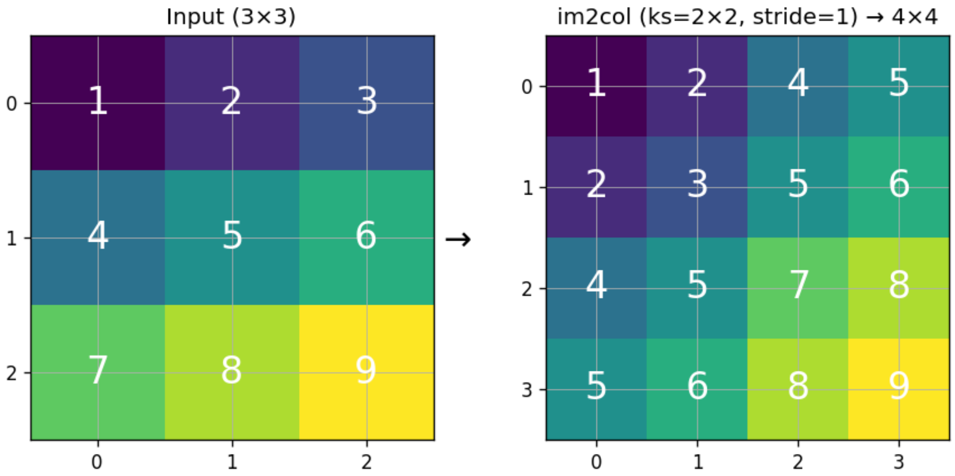
Why it matters

- Every **fully-connected layer** can be written as a GEMM.
- Using the **im2col** trick, a **convolution** becomes a GEMM too.
- Highly optimized GEMM kernels (BLAS, cuBLAS, MKL) are the backbone of deep learning performance.

Big picture: convolution as GEMM

- Convolution can be rearranged as: $\text{out\_cols} = W_{\text{col}} \times X_{\text{col}} + b$.
- `im2col` builds X_{col} by flattening each sliding window (patch) into a column.
- Kernels (weights) are reshaped into W_{col} so a single matrix multiply yields all outputs.
- Reshape out_cols back to $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$.

im2col visual example



Exercise: Implement `im2col` (grayscale, no padding, stride=1)

Goal. Write a minimal function `im2col(x, kH, kW)` for a single-channel image.

Specification

- Input: $x \in \mathbb{R}^{H \times W}$ (NumPy 2D array), kernel size (kH, kW).
- No padding, stride $= 1$.
- Output: $X_{\text{col}} \in \mathbb{R}^{(kH \cdot kW) \times (H_{\text{out}} \cdot W_{\text{out}})}$,

$$H_{\text{out}} = H - kH + 1, \quad W_{\text{out}} = W - kW + 1.$$

- Each column is one flattened $kH \times kW$ patch (row-major) taken from x.
- Save the 'im2col' function in the 'functional.py' python file

Function to implement:

```
def im2col(x: np.ndarray, kH: int, kW: int) -> np.ndarray:
    ...
```

Acceptance checks

- Shape matches: $(kH * kW,\ H_{\text{out}} * W_{\text{out}})$.
- Columns correspond to sliding windows (top-left moves by 1 pixel).
- Works on a small test (e.g., 3x3 image, 2x2 kernel).

My trivial implementation

```python
import numpy as np

def im2col(x, kH, kW):
    H, W = x.shape
    H_out = H - kH + 1
    W_out = W - kW + 1

    # Each column is one patch flattened (length kH*kW)
    # We create (kH*kW, H_out*W_out)
    X_col = np.empty((kH * kW, H_out * W_out), dtype=x.dtype)

    col = 0
    for i in range(H_out):            # top-left row of the patch
        for j in range(W_out):        # top-left col of the patch
            patch = x[i:i + kH, j:j + kW]
            X_col[:, col] = patch.reshape(-1)  # flatten row-major
            col += 1

    return X_col
```

Introducing the `Conv2d` Class

Motivation

In the previous lesson, we manually applied filters such as blur or sharpen. Those kernels were **hand-designed**.

Introducing the `Conv2d` Class

Motivation

In the previous lesson, we manually applied filters such as blur or sharpen. Those kernels were **hand-designed**.

What changes in a learning model?

- In AI models, the kernel is no longer fixed.
- Its values are **learned automatically** from data through training.
- The convolution operation itself, however, remains the same.

Introducing the `Conv2d` Class

Motivation

In the previous lesson, we manually applied filters such as blur or sharpen. Those kernels were **hand-designed**.

What changes in a learning model?

- In AI models, the kernel is no longer fixed.
- Its values are **learned automatically** from data through training.
- The convolution operation itself, however, remains the same.

Goal of this step

We now want to build a minimal class `Conv2d` that:

- stores the kernel (its size and current weights),
- applies it to an image using the `im2col` trick,
- produces the filtered output.

Lab time: Implement a Minimal `Conv2d`

Task

Implement a minimal `Conv2d` class that applies a single 2D filter to a grayscale image using the `im2col` trick.

- **Scope**: grayscale only; no batch, no channels, no stride, no padding, no bias.
- **Constructor** `__init__(kernel_size)`:
  - Accepts `int` or `(kH, kW)`.
  - Initializes `self.weight` of shape `(kH, kW)` with small random values.
- **Forward** `forward(x)`:
  - Input `x` has shape `(H, W)` with `H >= kH`, `W >= kW`.
  - Build `X_col = im2col(x, kH, kW)` with shape `(kH*kW, H_out*W_out)`.
  - Flatten `weight` to `W_col` with shape `(1, kH*kW)`.
  - Compute `Y = W_col @ X_col` with shape `(1, H_out*W_out)`.
  - Reshape to `(H_out, W_out)`, where `H_out = H - kH + 1`, `W_out = W - kW + 1`.
  - Return the 2D output.

Lab time: Implement a Minimal `Conv2d`

Function Signatures

```
class Conv2d:
  def __init__(self, kernel_size):  ...
  def forward(self, x):  ...
```

Acceptance Checks

- Shapes match: `out.shape == (H - kH + 1, W - kW + 1)`.
- With an all-ones kernel, `out[i,j]` equals the sum of the corresponding `kH x kW` patch.
- Deterministic for fixed `self.weight`.

My trivial implementation

```
import numpy as np
from functional.im2col import im2col

class Conv2d:
    def __init__(self, kernel_size):
        if isinstance(kernel_size, int):
            kernel_size = (kernel_size, kernel_size)
        self.kernel_size = kernel_size
        self.weight = np.random.randn(*kernel_size) * 0.01

    def forward(self, x):
        X_col = im2col(x, self.kernel_size[0], self.kernel_size[1])
        W_col = self.weight.reshape(1, -1)
        out = W_col @ X_col
        H_out = x.shape[0] - self.kernel_size[0] + 1
        W_out = x.shape[1] - self.kernel_size[1] + 1
        out = out.reshape(H_out, W_out)
        return out
```

Extending `im2col` and `Conv2d`: Padding and Stride

Motivation

So far, our convolution has worked only when the kernel fits perfectly inside the image. This meant:

$$H_{\text{out}} = H - k_H + 1, \quad W_{\text{out}} = W - k_W + 1$$

But real convolutional layers need more flexibility.

Extending `im2col` and `Conv2d`: Padding and Stride

Motivation

So far, our convolution has worked only when the kernel fits perfectly inside the image. This meant:

$$H_{\text{out}} = H - k_H + 1, \quad W_{\text{out}} = W - k_W + 1$$

But real convolutional layers need more flexibility.

Two key extensions

Padding: Adds artificial borders (usually zeros) around the image. $\Rightarrow$ Controls the spatial size of the output.

$$H_{\text{out}} = \frac{H + 2p - k_H}{s} + 1$$

Stride: Moves the kernel in steps larger than 1. $\Rightarrow$ Controls how densely we sample the image.

Extending `im2col` and `Conv2d`: Padding and Stride

Goal of this section

We now want to:

1. Extend Conv2d to handle padding, and extend im2col to handle stride.
2. Update the `Conv2d` class to use these parameters.
3. Verify that the output shapes follow the general formula.

Lab time: Add Padding and Stride Support

Goal

Extend your previous implementation so that both `im2col` and `Conv2d` support **padding** and **stride**.

Part 1: Update `im2col`

- Add an optional parameter `stride=1`.
- When extracting patches, move the sliding window in steps of `stride`.
- Update the output size:

$$H_{\text{out}} = \frac{H - k_H}{s} + 1, \quad W_{\text{out}} = \frac{W - k_W}{s} + 1$$

Lab time: Add Padding and Stride Support

Part 2: Update `Conv2d`

- Add parameters `stride` and `padding` in the constructor.
- If `padding > 0`, apply zero-padding using `np.pad`.
- Pass `stride` to the updated `im2col` call.
- Recompute output dimensions according to:

$$H_{\text{out}} = \frac{H + 2p - k_H}{s} + 1, \quad W_{\text{out}} = \frac{W + 2p - k_W}{s} + 1$$

Checks

- With `padding=1, stride=1`, the output should preserve input size.
- With `stride=2`, the output should shrink roughly by half.
- Verify that results match manual convolution for small examples.

My trivial implementation

```python
import numpy as np


def im2col(x, kH, kW, stride=1):
    H, W = x.shape
    H_out = (H - kH) // stride + 1
    W_out = (W - kW) // stride + 1

    X_col = np.empty((kH * kW, H_out * W_out), dtype=x.dtype)

    col = 0
    for i in range(0, H - kH + 1, stride):   # top-left row of the patch
        for j in range(0, W - kW + 1, stride):  # top-left col of the patch
            patch = x[i:i + kH, j:j + kW]
            X_col[:, col] = patch.reshape(-1)
            col += 1

    return X_col
```

My trivial implementation

```python
import numpy as np
from functional.im2col import im2col

class Conv2d:
    def __init__(self, kernel_size, stride=1, padding=0):
        if isinstance(kernel_size, int):
            kernel_size = (kernel_size, kernel_size)
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding
        self.weight = np.random.randn(*kernel_size) * 0.01
```

My trivial implementation

```python
def forward(self, x):
    # apply zero-padding if needed
    if self.padding > 0:
        x = np.pad(x, ((self.padding, self.padding),
                       (self.padding, self.padding)),
                   mode='constant')

    kH, kW = self.kernel_size
    # convert image to columns with stride
    X_col = im2col(x, kH, kW, stride=self.stride)
    W_col = self.weight.reshape(1, -1)

    out = W_col @ X_col
    H_out = (x.shape[0] - kH) // self.stride + 1
    W_out = (x.shape[1] - kW) // self.stride + 1
    out = out.reshape(H_out, W_out)
    return out
```

Extending `im2col` to Support Multiple Channels

From grayscale to multi-channel input

So far, `im2col` assumed a single 2D image (H, W). When the input has multiple channels (C_{in}, H, W), each patch now contains information from all channels.

Extending `im2col` to Support Multiple Channels

From grayscale to multi-channel input

So far, `im2col` assumed a single 2D image (H, W). When the input has multiple channels (C_{in}, H, W), each patch now contains information from all channels.

Key idea

- For each spatial position (i, j), we extract a 3D block:

$$\text{patch}_{i,j} \in \mathbb{R}^{C_{in} \times k_H \times k_W}$$

- This patch is flattened into a single column of length:

$$C_{in} \cdot k_H \cdot k_W$$

- The final output is:

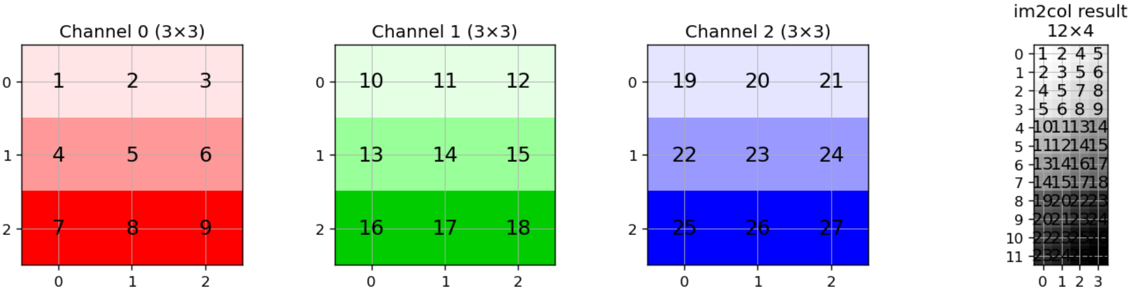
$$X_{\text{col}} \in \mathbb{R}^{(C_{in} \cdot k_H \cdot k_W) \times (H_{\text{out}} \cdot W_{\text{out}})}$$

Extending `im2col` to Support Multiple Channels

Implementation insight

- Loop over each channel, extract its patches, and place them in consecutive rows of `X_col`.
- The rest of the logic (stride, output shape) remains unchanged.

im2col rgb visual example



Lab time: Adding Channel Support to `im2col` and `Conv2d`

Goal: Extend your previous implementations to handle *multi-channel inputs* (e.g. RGB images).

Starting point:

- You already implemented `im2col(x, kH, kW, stride)` for grayscale images ($x \in \mathbb{R}^{H \times W}$).
- You also have a minimal `Conv2d` class that applies a single kernel on a 2D image.

Your task:

1. Modify `im2col` so that it works with inputs of shape (C, H, W). Each patch should now contain all channel values concatenated:
$$X_{\text{col}} \in \mathbb{R}^{(C \cdot k_H \cdot k_W) \times (H_{\text{out}} \cdot W_{\text{out}})}$$
2. Update `Conv2d.forward` to use the new `im2col`, assuming the kernel has shape (C, k_H, k_W).

Hint: Remember to apply zero-padding only to the spatial dimensions, not to the channels.

Lab time: Function Signatures for Multi-Channel Support

Implement the following interfaces:

```
def im2col(x, kH, kW, stride=1):
    """
    Args:
        x: input array of shape (C, H, W)
        kH, kW: kernel height and width
        stride: step size
    Returns:
        X_col: (C * kH * kW, H_out * W_out)
    """
```

```
class Conv2d:
    def __init__(self, in_channels, kernel_size, stride=1, padding=0):
        # weight: (in_channels, kH, kW)

    def forward(self, x):
        # x: (C, H, W)
        # return: (H_out, W_out)
```

My trivial implementation

```
import numpy as np


def im2col(x, kH, kW, stride=1):
    # x: input numpy array di shape (C, H, W)
    C, H, W = x.shape
    H_out = (H - kH) // stride + 1
    W_out = (W - kW) // stride + 1

    X_col = np.empty((C * kH * kW, H_out * W_out), dtype=x.dtype)

    col = 0
    for i in range(0, H - kH + 1, stride):       # top-left row of patch
        for j in range(0, W - kW + 1, stride):   # top-left col of patch
            patch = x[:, i:i + kH, j:j + kW]     # patch di forma (C, kH, kW)
            X_col[:, col] = patch.reshape(-1)
            col += 1

    return X_col
```

My trivial implementation

```
import numpy as np
from functional.im2col import im2col

class Conv2d:
    def __init__(self, in_channels, kernel_size, stride=1, padding=0):
        if isinstance(kernel_size, int):
            kernel_size = (kernel_size, kernel_size)
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding

        kH, kW = self.kernel_size
        self.weight = np.random.randn(in_channels, kH, kW) * 0.01
```

My trivial implementation

```
def forward(self, x):
    C, H, W = x.shape
    kH, kW = self.kernel_size

    if self.padding > 0:
        x = np.pad(x, ((0, 0), (self.padding, self.padding), (self.padding, self.padding)),
                   mode='constant')

    _, H_pad, W_pad = x.shape
    H_out = (H_pad - kH) // self.stride + 1
    W_out = (W_pad - kW) // self.stride + 1

    X_col = im2col(x, kH, kW, stride=self.stride)
    W_col = self.weight.reshape(1, -1)

    out = (W_col @ X_col).reshape(H_out, W_out)
    return out
```

Output Filters in Convolutional Layers

- Until now, we applied a **single filter** (kernel) to the input image.
- In practice, a convolutional layer learns **multiple filters**, each detecting a different feature (e.g., edges, textures, corners, colors).

Definition

Each output channel (feature map) is produced by one distinct filter:

$$\text{Output}[i] = W_i * X, \quad i = 1, \dots, C_{\text{out}}$$

where:

$$W_i \in \mathbb{R}^{C_{\text{in}} \times k_H \times k_W}, \quad X \in \mathbb{R}^{C_{\text{in}} \times H \times W}$$

- The layer thus produces an output tensor:

$$Y \in \mathbb{R}^{C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}}$$

- Each filter responds to different spatial patterns or semantic cues in the input.

Lab time: Supporting Multiple Output Filters

Goal

Extend your `Conv2d` class to support multiple output filters (`out_channels`).

- Each filter should have its own set of weights:

$$W \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times k_H \times k_W}$$

- The forward pass should produce:

$$Y \in \mathbb{R}^{C_{\text{out}} \times H_{\text{out}} \times W_{\text{out}}}$$

Hints

- Flatten both `X` (via `im2col`) and `W` for matrix multiplication:

$$Y_{\text{col}} = W_{\text{col}} \times X_{\text{col}}$$

- `W_col` shape: `(out_channels, in_channels * kH * kW)`
- Reshape output to `(out_channels, H_out, W_out)`

Lab time: Supporting Multiple Output Filters

Signature to complete:

```
class Conv2d:
    def __init__(self, in_channels, out_channels, kernel_size, stride=1, padding=0):
    def forward(self, x):
        """
        x: (C_in, H, W)
        return: (C_out, H_out, W_out)
        """
        C_in, H, W = x.shape
```

My trivial implementation

```python
class Conv2d:
    def __init__(self, in_channels, out_channels, kernel_size, stride=1, padding=0):
        if isinstance(kernel_size, int):
            kernel_size = (kernel_size, kernel_size)
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding

        kH, kW = self.kernel_size
        self.weight = np.random.randn(out_channels, in_channels, kH, kW) * 0.01
```

My trivial implementation

```python
def forward(self, x):
    """
    x: (C_in, H, W)
    return: (C_out, H_out, W_out)
    """
    C_in, H, W = x.shape
    kH, kW = self.kernel_size

    # Padding
    if self.padding > 0:
        x = np.pad(
            x,
            ((0, 0), (self.padding, self.padding), (self.padding, self.padding)),
            mode='constant'
        )
```

My trivial implementation

```python
_, H_pad, W_pad = x.shape
H_out = (H_pad - kH) // self.stride + 1
W_out = (W_pad - kW) // self.stride + 1

X_col = im2col(x, kH, kW, stride=self.stride)
W_col = self.weight.reshape(self.out_channels, -1)

# MatMul: (C_out, H_out*W_out)
out = W_col @ X_col

# Reshape a (C_out, H_out, W_out)
out = out.reshape(self.out_channels, H_out, W_out)
return out
```

Lab time: Extend Conv2d for Batch Input

Goal: Extend the `Conv2d` class so that it can process a batch of images instead of a single one.

Current situation:

- `im2col(x, kH, kW, stride)` works on a single image of shape (C, H, W).
- `Conv2d.forward(x)` also assumes a single image.

Task:

1. Modify `Conv2d.forward` so that it accepts inputs of shape:

$$x \in \mathbb{R}^{(N, C_{\text{in}}, H, W)}$$

   where N is the batch size.
2. Use a simple Python `for`-loop over N to call `im2col` on each image.
3. Concatenate the outputs into a tensor of shape:

$$(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$$

Hints:

- You do *not* need to modify `im2col`.
- Apply spatial padding to the whole batch before the loop.
- Reshape the output properly after the matrix multiplication.

My trivial implementation

```
class Conv2d:
    def __init__(self, in_channels, out_channels, kernel_size, stride=1, padding=0):
        if isinstance(kernel_size, int):
            kernel_size = (kernel_size, kernel_size)
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding

        kH, kW = self.kernel_size
        self.weight = np.random.randn(out_channels, in_channels, kH, kW) * 0.01
```

My trivial implementation

```python
def forward(self, x):
    """
    x: (N, C_in, H, W)
    return: (N, C_out, H_out, W_out)
    """
    N, C_in, H, W = x.shape
    kH, kW = self.kernel_size
    p = self.padding
    s = self.stride

    if p > 0:
        x = np.pad(x, ((0, 0), (0, 0), (p, p), (p, p)), mode='constant')

    _, _, H_pad, W_pad = x.shape
    H_out = (H_pad - kH) // s + 1
    W_out = (W_pad - kW) // s + 1
```

My trivial implementation

```python
W_col = self.weight.reshape(self.out_channels, -1)  # (C_out, C_in*kH*kW)
out = np.empty((N, self.out_channels, H_out, W_out), dtype=x.dtype)

for n in range(N):
    X_col = im2col(x[n], kH, kW, stride=s)          # (C_in*kH*kW, H_out*W_out)
    Y_col = W_col @ X_col                           # (C_out, H_out*W_out)
    out[n] = Y_col.reshape(self.out_channels, H_out, W_out)

return out
```

Motivation for Pooling

- After convolution $+$ nonlinearity, feature maps may still be large in spatial size.
- We want to **reduce spatial resolution** while keeping the most relevant information.
- Pooling provides:
  - **Dimensionality reduction** $\rightarrow$ fewer parameters and faster computation.
  - **Translation invariance** $\rightarrow$ small shifts in the input do not change the pooled feature.
- The most common pooling operation is the **Max Pool**.

$$\text{MaxPool}(X) = \max_{(i,j)\in\text{window}} X_{i,j}$$

How Max Pooling Works

- A sliding window of size $k \times k$ moves across each feature map.
- For each window, we take the **maximum value**.
- This produces a smaller output feature map:

$$H_{out} = \left\lfloor \frac{H_{in} - k}{s} \right\rfloor + 1$$

$$W_{out} = \left\lfloor \frac{W_{in} - k}{s} \right\rfloor + 1$$

- Pooling can be seen as a special convolution:
  - No learnable weights
  - The operation is a reduction (max, avg, etc.)

Example:

$$\text{window} \begin{bmatrix} 1 & 5 \\ 2 & 3 \end{bmatrix} \rightarrow \max = 5$$

Stride in Max Pooling

- The **stride** controls how far the pooling window moves at each step.
- When `stride = 1`, windows **overlap** - each window shares pixels with the next.
- When `stride = kernel_size`, windows are **non-overlapping**.
- This is the most common configuration:

```
MaxPool2d(kernel_size=2, stride=2)
```

  It halves both height and width of the feature map.

Output size formula

$$H_{out} = \left\lfloor \frac{H_{in} - k}{s} \right\rfloor + 1 \qquad W_{out} = \left\lfloor \frac{W_{in} - k}{s} \right\rfloor + 1$$

- Stride $=$ kernel size $\Rightarrow$ strong downsampling, faster computation.
- Smaller stride $\Rightarrow$ overlapping pooling, smoother transitions but higher cost.

Max Pooling Across Channels

Key idea: Max pooling does **not mix information across channels**.

- The operation is applied **independently** to each channel.
- Within each channel, a sliding window selects the maximum value in every spatial region.
- The same pooling window moves over all channels, but the maxima are computed separately.

Max Pooling Across Channels

Key idea: Max pooling does **not mix information across channels**.

- The operation is applied **independently** to each channel.
- Within each channel, a sliding window selects the maximum value in every spatial region.
- The same pooling window moves over all channels, but the maxima are computed separately.

Example

For an input of shape (N, C, H, W), the output has shape $(N, C, H_{\text{out}}, W_{\text{out}})$ because each of the C channels is pooled independently.

Max Pooling Across Channels

Key idea: Max pooling does **not mix information across channels**.

- The operation is applied **independently** to each channel.
- Within each channel, a sliding window selects the maximum value in every spatial region.
- The same pooling window moves over all channels, but the maxima are computed separately.

Example

For an input of shape (N, C, H, W), the output has shape $(N, C, H_{\text{out}}, W_{\text{out}})$ because each of the C channels is pooled independently.

Intuition: Pooling reduces spatial resolution (height and width) but preserves the depth (number of channels).

Lab time: Implement MaxPool2d starting from Conv2d

Goal. Implement a Max Pooling layer by reusing the structure of `Conv2d` (NCHW layout). **Required interface**

```
class MaxPool2d:
    def __init__(self, kernel_size, stride=None, padding=0):
    def forward(self, x):
```

Requirements

- Apply the maximum *independently* on each channel.
- Support `padding=p` and `stride=s`, with default `s = kernel_size` if `None`.
- Output shape formulas (same as in `Conv2d`):

$$H_{\text{out}} = \left\lfloor \frac{H + 2p - k_H}{s} \right\rfloor + 1, \quad W_{\text{out}} = \left\lfloor \frac{W + 2p - k_W}{s} \right\rfloor + 1.$$

- No weights or biases; optionally reuse `im2col` to extract patches and take `max` over each column.

Minimal hints

- Copy the structure of the `Conv2d.forward`: optional padding, compute $H_{\text{out}}, W_{\text{out}}$, loop over batch and channels.
- For each channel: `X_col = im2col(...)` $\to$ `X_col.max(axis=0)` $\to$ reshape to $(H_{\text{out}}, W_{\text{out}})$.

My trivial implementation

```python
import numpy as np
from functional.im2col import im2col

class MaxPool2d:
    def __init__(self, kernel_size, stride=None, padding=0):
        if isinstance(kernel_size, int):
            kernel_size = (kernel_size, kernel_size)
        self.kernel_size = kernel_size
        self.stride = stride if stride is not None else kernel_size
        self.padding = padding

    def forward(self, x):
        """
        x: (N, C, H, W)
        return: (N, C, H_out, W_out)
```

My trivial implementation

```python
N, C, H, W = x.shape
kH, kW = self.kernel_size
p = self.padding
s = self.stride

if p > 0:
    x = np.pad(x, ((0,0),(0,0),(p,p),(p,p)), mode='constant')

_, _, H_pad, W_pad = x.shape
H_out = (H_pad - kH) // s + 1
W_out = (W_pad - kW) // s + 1

out = np.empty((N, C, H_out, W_out), dtype=x.dtype)

for n in range(N):
    for c in range(C):
        X_col = im2col(x[n, c:c+1], kH, kW, stride=s)  # (kH*kW, H_out*W_out)
        out[n, c] = X_col.max(axis=0).reshape(H_out, W_out)

return out
```

Putting all together

```
x = np.array([
    [
        [[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9,]],

        [[11, 12, 13],
        [ 14, 15, 16],
        [ 17, 18, 19]]
    ]
], dtype=float)

conv = Conv2d(in_channels=2, out_channels=2, kernel_size=2, stride=1, padding=0)
blur = np.array([[1, 1],
                [1, 1]])
vertical = np.array([[-1, 1],
                    [-1, 1]])

for c in range(conv.in_channels):
    conv.weight[0, c] = blur
    conv.weight[1, c] = vertical
```

Putting all together

```
pool = MaxPool2d(kernel_size=2, stride=2)

y_conv = conv.forward(x)
y_pool = pool.forward(y_conv)

print("Input (x):")
print(x)
print("\nAfter Conv2d:")
print(y_conv)
print("\nAfter MaxPool2d:")
print(y_pool)
```

Putting all together: output

```
Input (x):
[[[[ 1.  2.  3.]
   [ 4.  5.  6.]
   [ 7.  8.  9.]]

  [[11. 12. 13.]
   [14. 15. 16.]
   [17. 18. 19.]]]]

After Conv2d:
[[[[64. 72.]
   [88. 96.]]

  [[ 4.  4.]
   [ 4.  4.]]]]

After MaxPool2d:
[[[[96.]]

  [[ 4.]]]]
```

Thanks!