Modern Computer Vision Spring 2026

Amit Shmidov | Tutorial 1

This presentation largely rely on DL4CV@Weizmann 23’

PyTorch Basics

What is PyTorch?

- Open-source deep learning framework by Meta (Facebook AI Research)
- Tensor computation with GPU acceleration
- Pythonic and intuitive — works like NumPy but with GPU support
- Rich ecosystem: torchvision, torchaudio, torchtext, HuggingFace
- De facto standard for research and increasingly for production

Tensors

A tensor is a multi-dimensional array

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

```
import torch
a = torch.tensor([[1, 2, 3], [4, 5, 6]])
```

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

```
import torch
import numpy as np

a_np = np.array([[1, 2, 3], [4, 5, 6]])
a = torch.from_numpy(a_np)
```

Tensors | creation

$$a = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

```
import torch
a = torch.zeros(size=(2, 3))
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

$$a = \begin{pmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{pmatrix}$$

```
import torch
a = torch.ones(size=(3, 2))
```

```
tensor([[1., 1.],
        [1., 1.],
        [1., 1.]])
```

Uniformly random a

```
import torch
a = torch.rand(size=(1, 4))
```

```
tensor([[0.3062, 0.1307, 0.8366, 0.1338]])
```

Identity matrix

```
import torch
a = torch.eye(5)
```

```
tensor([[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 1.]])
```

Tensors | attributes

Key attributes of a tensor:

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

shape:

```
a.shape
```

```
a.size()
```

```
torch.Size([2, 3])
```

dtype:

```
a.dtype
```

```
torch.int64
```

device:

```
a.device
```

```
device(type='cpu')
```

during creation:

```
a = torch.tensor([[1, 2, 3], [4, 5, 6]],
                 dtype=torch.float32, device="cpu")
```

common dtypes:

```
torch.float32  # default float
torch.float64
torch.int32  # default int
torch.int8
torch.bool
```

change attributes:

```
a.to(dtype=torch.float64)
a.to(device="cuda")
```

Tensors | device management

Do we have GPU resource available*?

```
torch.cuda.is_available()
```

How many devices do we have?

```
torch.cuda.device_count()
```

Moving to a specific GPU:

```
a.to(device="cuda:1")
```

*Non-NVIDIA GPUs could be available even when False.

Tensors | arithmetic operations

$$a = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \qquad b = \begin{pmatrix} 4 & 3 & 2 & 1 \\ 8 & 7 & 6 & 5 \end{pmatrix}$$

```
a = torch.tensor([[1, 2, 3, 4], [5, 6, 7, 8]])
b = torch.tensor([[4, 3, 2, 1], [8, 7, 6, 5]])
```

`a + b`

```
tensor([[ 5,  5,  5,  5],
        [13, 13, 13, 13]])
```

`a / b`

```
tensor([[0.2500, 0.6667, 1.5000, 4.0000],
        [0.6250, 0.8571, 1.1667, 1.6000]])
```

`a - b`

```
tensor([[-3, -1,  1,  3],
        [-3, -1,  1,  3]])
```

`a * b`

```
tensor([[ 4,  6,  6,  4],
        [40, 42, 42, 40]])
```

`a ** b`

```
tensor([[     1,      8,      9,      4],
        [390625, 279936, 117649,  32768]])
```

Tensors | Boolean element-wise operations

$$a = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \qquad b = \begin{pmatrix} 4 & 3 & 2 & 1 \\ 8 & 7 & 6 & 5 \end{pmatrix}$$

```
a == b
```

```
tensor([[False, False, False, False],
        [False, False, False, False]])
```

```
a < b
```

```
tensor([[ True,  True, False, False],
        [ True,  True, False, False]])
```

```
a != b
```

```
tensor([[True, True, True, True],
        [True, True, True, True]])
```

```
a > b
```

```
tensor([[False, False,  True,  True],
        [False, False,  True,  True]])
```

Tensors | indexing and slicing

$$a = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \qquad b = \begin{pmatrix} 4 & 3 & 2 & 1 \\ 8 & 7 & 6 & 5 \end{pmatrix}$$

```
a[0]
```

```
tensor([1, 2, 3, 4])
```

```
a[0, :]
```

```
tensor([1, 2, 3, 4])
```

```
a[0, 0]
```

```
tensor(1)
```

```
a[1:2, 3]
```

```
tensor([8])
```

```
a[:, 2]
```

```
tensor([3, 7])
```

```
a[0, :].shape
```

```
torch.Size([4])
```

```
a[:, 0].shape
```

```
torch.Size([2])
```

Tensors | concatenation and stacking

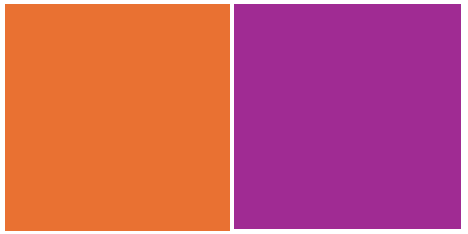
Concatenation:

```
torch.cat((a, b))
```

```
tensor([[1, 2, 3, 4],
        [5, 6, 7, 8],
        [4, 3, 2, 1],
        [8, 7, 6, 5]])
```

```
torch.cat((a, b), dim=1)
```

```
tensor([[1, 2, 3, 4, 4, 3, 2, 1],
        [5, 6, 7, 8, 8, 7, 6, 5]])
```



Stacking:

```
torch.stack((a, b))
```

```
tensor([[[1, 2, 3, 4],
         [5, 6, 7, 8]],

        [[4, 3, 2, 1],
         [8, 7, 6, 5]]])
```

```
torch.stack((a, b)).shape
```

```
torch.Size([2, 2, 4])
```

Tensors | in-place operations

$$a = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

$$b = \begin{pmatrix} 4 & 3 & 2 & 1 \\ 8 & 7 & 6 & 5 \end{pmatrix}$$

```
a.add_(b)
a
```

```
tensor([[ 5,  5,  5,  5],
        [13, 13, 13, 13]])
```

```
a.clamp_(2, 5)
a
```

```
tensor([[2, 2, 3, 4],
        [5, 5, 5, 5]])
```

```
a.mul_(b)
a
```

```
tensor([[ 4,  6,  6,  4],
        [40, 42, 42, 40]])
```

Tensors | reshaping

$a = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$

$b = \begin{pmatrix} 4 & 3 & 2 & 1 \\ 8 & 7 & 6 & 5 \end{pmatrix}$

Contiguity: elements are stored in a single sequential memory block, can use **.contiguous()** if needed

```
torch.stack((a, b), dim=0).permute(1, 0, 2)
```

```
tensor([[[1, 2, 3, 4],
         [4, 3, 2, 1]],

        [[5, 6, 7, 8],
         [8, 7, 6, 5]]])
```

```
a.view((1, 8))
```

Reshape without copying data

```
tensor([[1, 2, 3, 4, 5, 6, 7, 8]])
```

```
a.reshape((1, 8))
```

Reshape even when tensor is non-contiguous

```
tensor([[1, 2, 3, 4, 5, 6, 7, 8]])
```

```
torch.squeeze(torch.ones((5, 1)))
```

```
tensor([1., 1., 1., 1., 1.])
```

```
a.transpose(0, 1)
```

```
tensor([[1, 5],
        [2, 6],
        [3, 7],
        [4, 8]])
```

```
a.transpose(0, 1).is_contiguous()
```

```
False
```

```
torch.unsqueeze(torch.ones((5, )), dim=0)
```

```
tensor([[1., 1., 1., 1., 1.]])
```

Tensors | vector operations

$$a = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

$$b = \begin{pmatrix} 4 & 3 & 2 & 1 \\ 8 & 7 & 6 & 5 \end{pmatrix}$$

```
torch.dot(a[0, :], b[1, :])
```

```
tensor(60)
```

Only 1D

```
torch.inner(a[0, :], b[1, :])
```

```
tensor(60)
```

Same to .dot on 1D

```
torch.inner(a, b)
```

```
tensor([[ 20,  60],
        [ 60, 164]])
```

Above 1D sums per row

Different methods for matrix multiplication:

```
torch.matmul(a, b.T)
```

```
torch.mm(a, b.T)
```

```
a @ b.T
```

```
tensor([[ 20,  60],
        [ 60, 164]])
```

```
mat = torch.rand(size=(N, M))  # N x M
vec = torch.rand(size=(M, ))   # M x 1
out = torch.matmul(mat, vec)   # N x 1

mat1 = torch.rand(size=(N, M))  # N x M
mat2 = torch.rand(size=(M, K))  # M x K
out = torch.matmul(mat1, mat2)  # N x K

mat1 = torch.rand(size=(B, N, M))  # B x N x M
mat2 = torch.rand(size=(B, M, K))  # B x M x K
out = torch.matmul(mat1, mat2)  # B x N x K
```

Tensors | useful functions

$$a = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

$$b = \begin{pmatrix} 4 & 3 & 2 & 1 \\ 8 & 7 & 6 & 5 \end{pmatrix}$$

```
torch.where(condition, if_true, if_false)
torch.where(a > 3, a, torch.zeros_like(a))
```

```
tensor([[0, 0, 0, 4],
        [5, 6, 7, 8]])
```

```
a.flip([0])
```

```
tensor([[5, 6, 7, 8],
        [1, 2, 3, 4]])
```

```
a.flip([0, 1])
```

```
tensor([[8, 7, 6, 5],
        [4, 3, 2, 1]])
```

```
a.unique()
torch.unique(a)
```

```
tensor([1, 2, 3, 4, 5, 6, 7, 8])
```

```
c = torch.rand((3, 3))
```

```
tensor([[0.9903, 0.2482, 0.0167],
        [0.2060, 0.2074, 0.4798],
        [0.3014, 0.9386, 0.1152]])
```

```
torch.sort(c)
```

Default to ascending sort per row

```
torch.return_types.sort(
values=tensor([[0.0167, 0.2482, 0.9903],
        [0.2060, 0.2074, 0.4798],
        [0.1152, 0.3014, 0.9386]]),
indices=tensor([[2, 1, 0],
        [0, 1, 2],
        [2, 0, 1]]))
```

```
torch.sort(c, dim=0)
```

```
torch.sort(c, dim=0, descending=True)
```

Tensors | broadcasting

$$a = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

$$b = \begin{pmatrix} 4 & 3 & 2 & 1 \\ 8 & 7 & 6 & 5 \end{pmatrix}$$

Allows operations on tensors with different shapes without explicit copying

Rules:

1. Dimensions are compatible if equal, or one of them is 1
2. Missing dimensions are treated as size 1

```
a + torch.ones(4)
```

(2, 4) + (4,)

```
a + torch.ones((2, 1))
```

(2, 4) + (2, 1)

```
tensor([[2., 3., 4., 5.],
        [6., 7., 8., 9.]])
```

```
torch.ones((3, 1, 2)) @ a
```

(3, 1, 2) @ (2, 4)

```
tensor([[[ 6.,  8., 10., 12.]],

        [[ 6.,  8., 10., 12.]],

        [[ 6.,  8., 10., 12.]]])
```

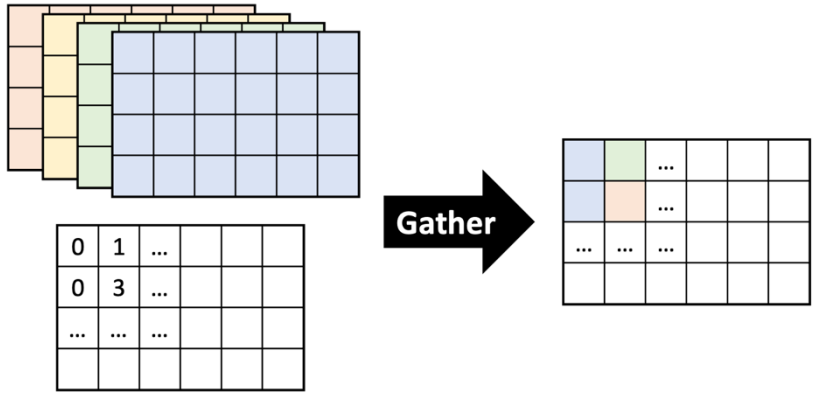
Tensors | Einstein summation

1. Each dimension in each operand is assigned a letter
2. Multiplication is done over dimensions with the same letter
3. Summation is then done over dimensions with letters that are not in the output

```
M, N, K, L, P = 4, 6, 5, 7, 8
a = torch.rand(size=(M, N, K, L, P))
b = torch.rand(size=(L, K, P))
print(torch.einsum("mnklp, lkp -> lp", a, b).shape)
print(torch.einsum("mnklp, lkp -> mk", a, b).shape)
```

```
torch.Size([7, 8])
torch.Size([4, 5])
```

Tensors | gather




```
a = torch.ones((3, 3))
b = 2 * torch.ones((3, 3))
c = 3 * torch.ones((3, 3))

torch.gather(torch.stack((a, b, c), dim=0), 0, torch.tensor([[[0, 1, 0],
                                                             [2, 0, 1],
                                                             [0, 1, 2]]]))
```

```
tensor([[[1., 2., 1.],
         [3., 1., 2.],
         [1., 2., 3.]]])
```

A stack of same-dimensions original tensors

A dimension along the stack

A tensor of indices

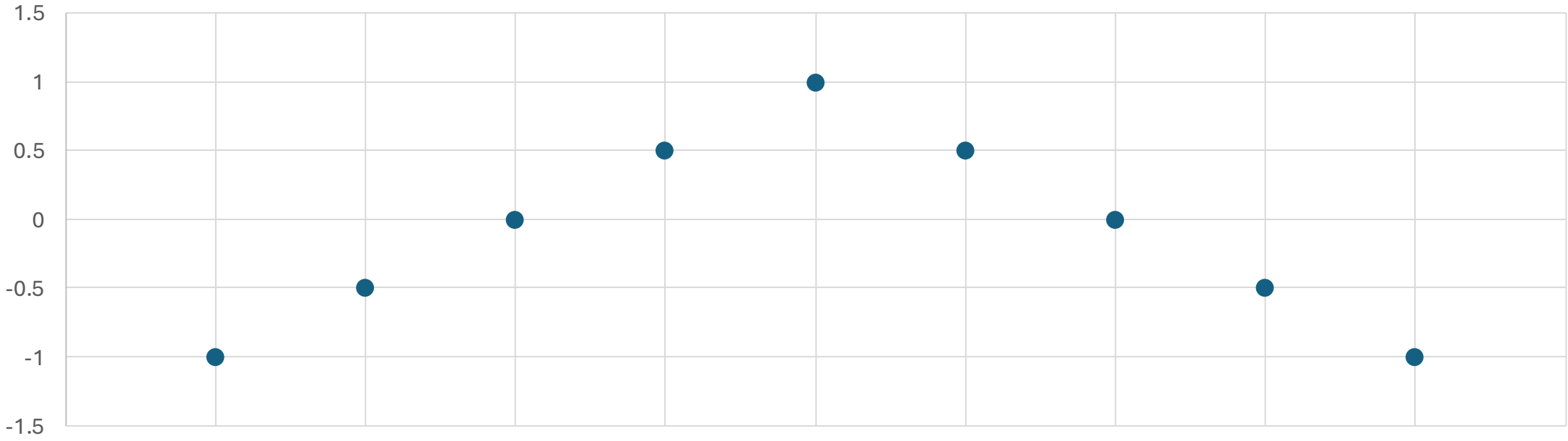
Vectorization & GPU

Golden rules:

1. GPUs allow for massive parallelism for tensor operations, but are usually bounded by memory
2. Always use built-in vectorized operations instead of Python loops
3. Modern deep learning pipelines are huge and complex – be careful with **time** and **space**

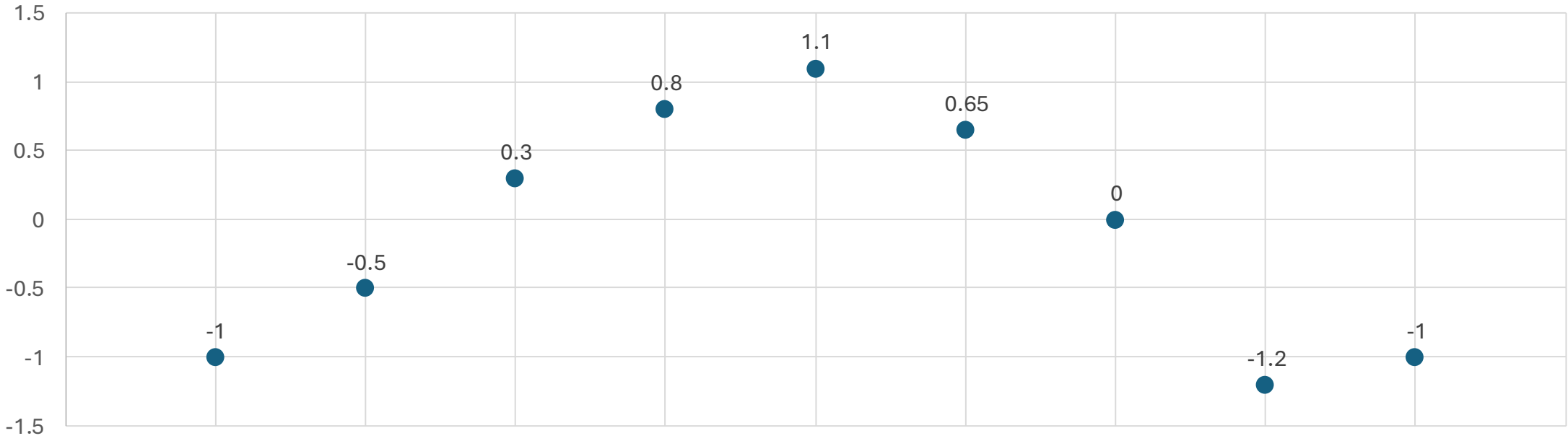
Convolution (1D)

Assume we have a 1D dataset that looks like $x = [-1, -0.5, 0, 0.5, 1, 0.5, 0, -0.5, -1]$:



Convolution (1D)

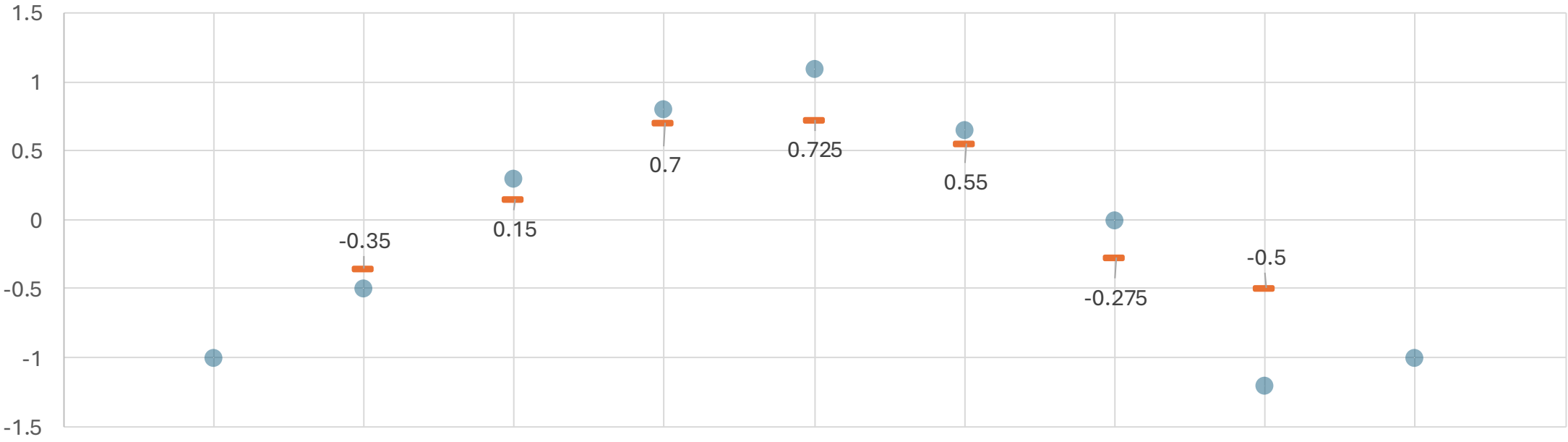
In real life, we sometimes have noise:



Convolution (1D)

We want to smooth our dataset with the following filter ("local function") w:

w: for every element, return the average of its two neighbors



Convolution (1D)

What is the contribution of $x[k]$ to $y[n]$?

If $n-k=1$ or $n-k=-1$ then 0.5, otherwise 0.

We can say that:

$$y[n]=\sum_{k} x[k]\cdot w[n-k]$$

With $w[-1]=0.5, w[0]=0, w[1]=0.5$ (and $w[k]=0\ \forall k\notin[-1,0,1]$)

Convolution (1D)

What is the contribution of $x[k]$ to $y[n]$?

If $n-k=1$ or $n-k=-1$ then 0.5, otherwise 0.

We can say that:

$$y[n]=\sum_{k} x[k]\cdot w[n-k]$$

With $w[-1]=0.5, w[0]=0, w[1]=0.5$ (and $w[k]=0\ \forall k\notin[-1,0,1]$)

Convolution (1D)

We can also ask what would be the result $y[n]$ for every $x[n]$:

$$y[n] = w[-1] \cdot x[n-1] + w[0] \cdot x[n] + w[1] \cdot x[n+1]$$

$$y[n] = \sum_{k} w[k] \cdot x[n+k]$$

Convolution (1D) | cross-correlation

We can also ask what would be the result $y[n]$ for every $x[n]$:

$$y[n] = w[-1] \cdot x[n-1] + w[0] \cdot x[n] + w[1] \cdot x[n+1]$$

$$y[n] = \sum_k w[k] \cdot x[n+k]$$

This is a **cross-correlation**.

Convolution (1D) | cross-correlation

We can also ask what would be the result $y[n]$ for every $x[n]$:

$$y[n] = w[-1] \cdot x[n-1] + w[0] \cdot x[n] + w[1] \cdot x[n+1]$$

$$y[n] = \sum_k w[k] \cdot x[n+k]$$

We call this **cross-correlation**.

With $l = n + k$:

$$y[n] = \sum_l x[l] \cdot w[l-n] = \sum_l x[l] \cdot w[-(n-l)]$$

Cross-correlation and convolution are the same operators with a flipped kernel!

Convolution (1D) | PyTorch

```
from torch.nn import functional as F

x = torch.tensor([[[-1, -0.5, 0, 0.5, 1, 0.5, 0, -0.5, -1]]])  # 1 x 1 x 9
w = torch.tensor([[[-1., 0., 1.]]])                            # 1 x 1 x 3
F.conv1d(x, w)
```

```
tensor([[[ 1.,  1.,  1.,  0., -1., -1., -1.]]])
```

```
F.conv1d(x, w.flip(-1))
```

```
tensor([[[-1., -1., -1.,  0.,  1.,  1.,  1.]]])
```

There is also a Module version (`nn.Conv1d(in_channels, out_channels, kernel_size, ...)`) – will be relevant for DNNs

```
x = torch.tensor([[[-1, -0.5, 0, 0.5, 1, 0.5, 0, -0.5, -1]]])
w = torch.tensor([[[-1., 0., 1.]]])
```

```
tensor([[[ 1.,  1.,  1.,  0., -1., -1., -1.]]])
```

Convolution (1D) | important parameters

- Noticed that $y[n]$ has less elements than $x[n]$? What can we do with $x[1]$ or $x[N]$?
  - We can expand $x[n]$ to both directions

This is called **padding** (default = 0, use "same" for equal input and output)

```
F.conv1d(x, w, padding=1)
```

```
tensor([[[-0.5000,  1.0000,  1.0000,  1.0000,  0.0000, -1.0000, -1.0000,
          -1.0000,  0.5000]]])
```

```
F.conv1d(x, w, padding=3)
```

```
tensor([[[ 0.0000, -1.0000, -0.5000,  1.0000,  1.0000,  1.0000,  0.0000,
          -1.0000, -1.0000, -1.0000,  0.5000,  1.0000,  0.0000]]])
```

```
F.conv1d(x, w, padding="same")
```

```
tensor([[[-0.5000,  1.0000,  1.0000,  1.0000,  0.0000, -1.0000, -1.0000,
          -1.0000,  0.5000]]])
```

```
x = torch.tensor([[[-1, -0.5, 0, 0.5, 1, 0.5, 0, -0.5, -1]]])
w = torch.tensor([[[-1., 0., 1.]]])
```

```
tensor([[[ 1.,  1.,  1.,  0., -1., -1., -1.]]])
```

Convolution (1D) | important parameters

- What if we want to average $x[n]$ over greater ranges?
  - Use the same $w[k]$ on farther elements of x with $x[d \cdot k]$ instead of $x[k]$:

$$y[n] = \sum_k w[k] \cdot x[n - d \cdot k]$$

This is called **dilation** (default = 1)

```
F.conv1d(x, w, dilation=2)
```

```
tensor([[[ 2.,  1.,  0., -1., -2.]]])
```

```
F.conv1d(x, w, padding=2, dilation=2)
```

```
tensor([[[ 0.0000,  0.5000,  2.0000,  1.0000,  0.0000, -1.0000, -2.0000,
          -0.5000,  0.0000]]])
```

```
x = torch.tensor([[[-1, -0.5, 0, 0.5, 1, 0.5, 0, -0.5, -1]]])
w = torch.tensor([[[-1., 0., 1.]]])
```

```
tensor([[[ 1.,  1.,  1.,  0., -1., -1., -1.]]])
```

Convolution (1D) | important parameters

- What if want to skip elements in x?
  - We can compute $y[n]$ not for $x[n]$, but for $x[s \cdot n]$: $y[n] = \sum_k w[k] \cdot x[s \cdot n - k]$
  - This is called **stride** (default = 1)

```
F.conv1d(x, w, stride=3)
```

```
tensor([[[ 1.,  0., -1.]]])
```

```
F.conv1d(x, w, dilation=2, stride=2)
```

```
tensor([[[ 2.,  0., -2.]]])
```

Convolution 1D | shapes

Input: $N \times C_{in} \times L_{in}$

N – number of samples (e.g., number of input audio files)

C_{in} – number of features per sample (e.g., number of tracks per audio file)

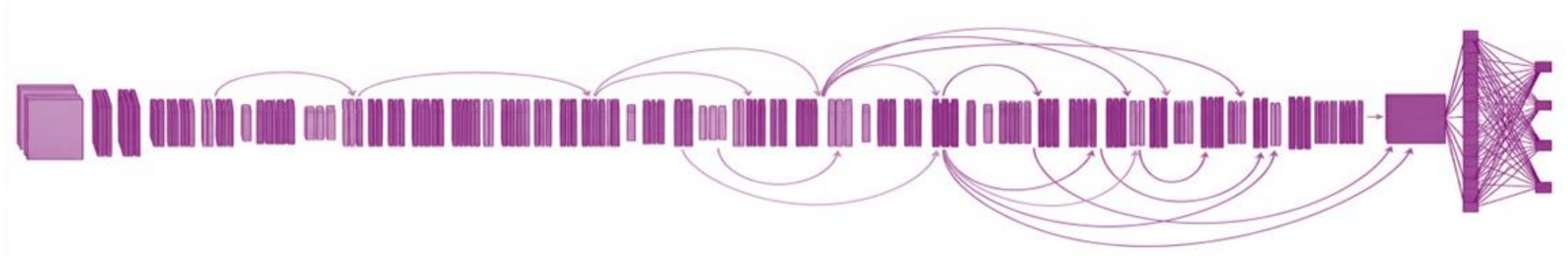
L_{in} – number of elements per sample (e.g., length of audio file)

Kernel: $C_{out} \times C_{in} \times L_{kernel}$

Output will be: $N \times C_{out} \times L_{out}$

$$L_{out} = \left\lfloor \frac{L_{in} + 2 \cdot \text{padding} - \text{dilation} \cdot (L_{kernel} - 1) - 1}{\text{stride}} + 1 \right\rfloor$$

Images in PyTorch



Images as Tensors

- Images in PyTorch: tensors of shape **(C, H, W)**
  - C = channels (3 for RGB, 1 for grayscale)
  - H = height, W = width
- Two common representations:
  - Float [0.0, 1.0] — used for computation and neural networks
  - Uint8 [0, 255] — standard image format, used for storage

```
import torchvision
img = torchvision.io.read_image("image.jpeg")
img.shape
```

```
torch.Size([3, 1512, 2016])
```

```
img[0, 400:405, 1010:1013]
```

```
tensor([[233, 233, 233],
        [233, 233, 233],
        [233, 233, 233],
        [233, 233, 233],
        [233, 233, 233]], dtype=torch.uint8)
```

```
img[0, 400:405, 1010:1013].float() / 255
```

```
tensor([[0.9137, 0.9137, 0.9137],
        [0.9137, 0.9137, 0.9137],
        [0.9137, 0.9137, 0.9137],
        [0.9137, 0.9137, 0.9137],
        [0.9137, 0.9137, 0.9137]])
```

Images as Tensors

```
for i in range(3):
    channel_img = torch.zeros_like(img)
    channel_img[i, :, :] = img[i, :, :]
    torchvision.io.write_jpeg(channel_img, f"{['red', 'green', 'blue'][i]}_image.jpeg")
```

```
gray_img = torchvision.transforms.functional.rgb_to_grayscale(img)
torchvision.io.write_jpeg(gray_img, "gray_image.jpeg")
```

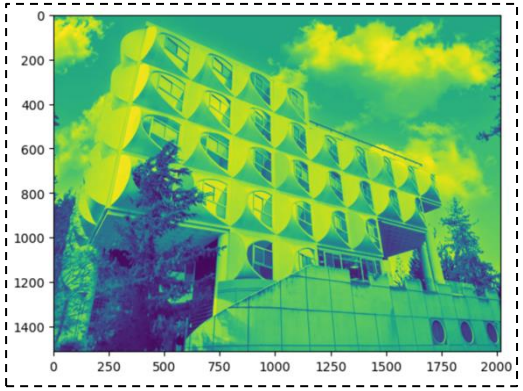
Visualization with Matplotlib

The go-to for image display (and plotting in general)

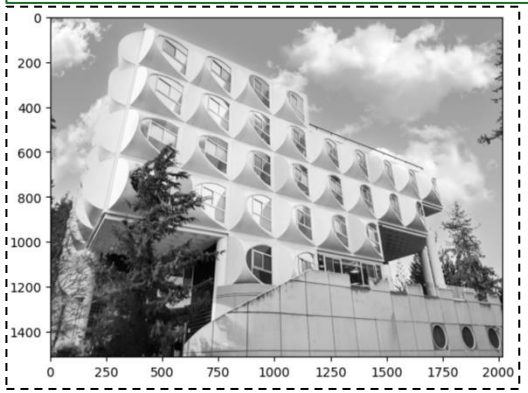
```
plt.figure()
plt.imshow(img.permute(1, 2, 0))
plt.show()
```

Channel order matters – PyTorch (C, H, W) but Matplotlib (H, W, C)


```
plt.figure()
plt.imshow(gray_img.permute(1, 2, 0))
plt.show()
```




```
plt.figure()
plt.imshow(gray_img.permute(1, 2, 0), cmap="gray")
plt.show()
```



Visualization with Matplotlib

Matplotlib automatically adjusts gray level range – use **vmin** and **vmax** to show true level values

```
fig, ax = plt.subplots(1, 2)
ax[0].imshow(gray_img.permute(1, 2, 0) / 1000, cmap="gray")
ax[1].imshow(gray_img.permute(1, 2, 0) / 1000, cmap="gray", vmin=0, vmax=1)
plt.tight_layout()
plt.show()
```

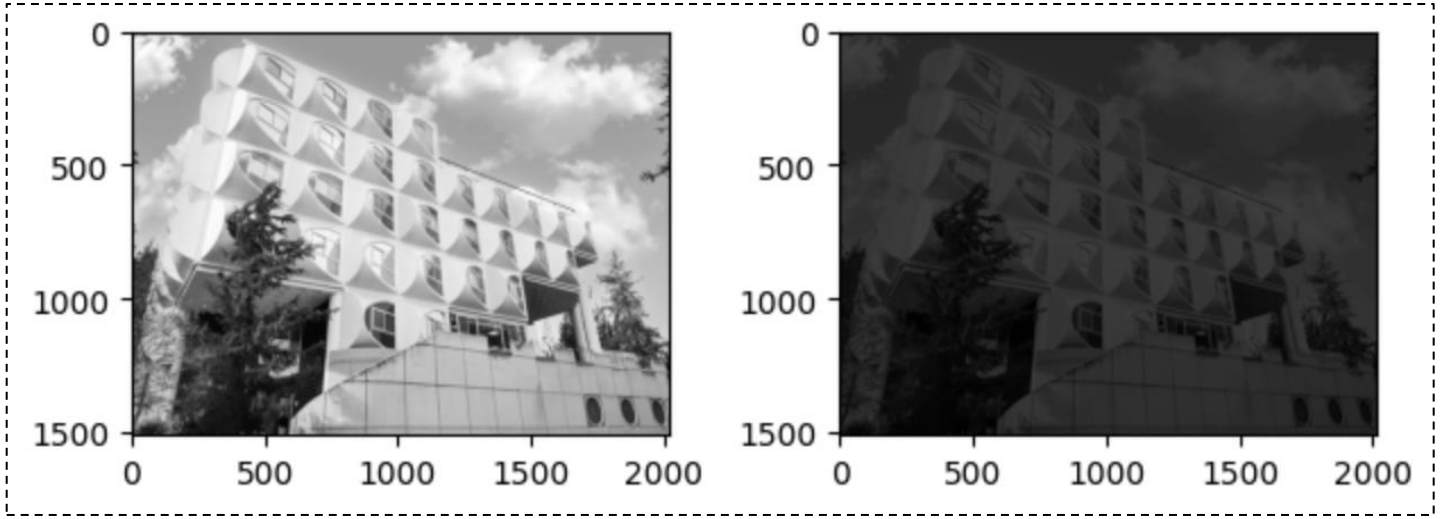
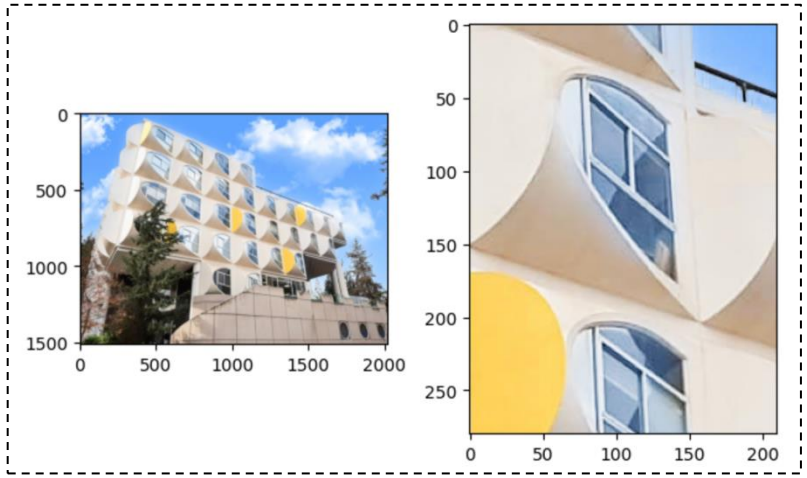


Image slicing & manipulation

Crop:

```
img_perm = img.permute(1, 2, 0)
fig, ax = plt.subplots(1, 2)
ax[0].imshow(img_perm)
ax[1].imshow(img_perm[450:730, 1000:1210, :])
plt.tight_layout()
plt.show()
```



Flip:

```
img_perm = img.permute(1, 2, 0)
fig, ax = plt.subplots(1, 3, figsize=(20, 5))
ax[0].imshow(img_perm)
ax[1].imshow(img_perm.flip(0))
ax[2].imshow(img_perm.flip(1))
plt.tight_layout()
plt.show()
```

Image slicing & manipulation

```python
from copy import deepcopy

img_perm = img.permute(1, 2, 0)
fig, ax = plt.subplots(1, 3, figsize=(20, 5))
ax[0].imshow(img_perm)

img_perm_replaced = deepcopy(img_perm)
img_perm_replaced[:, :, 1] = 0
ax[1].imshow(img_perm_replaced)

img_perm_black = deepcopy(img_perm)
img_perm_black[300:590, 1200:1470, :] = 0
ax[2].imshow(img_perm_black)
plt.tight_layout()
plt.show()
```

These are just examples –
all standard tensor ops apply on images!

Image slicing & manipulation

Interpolate

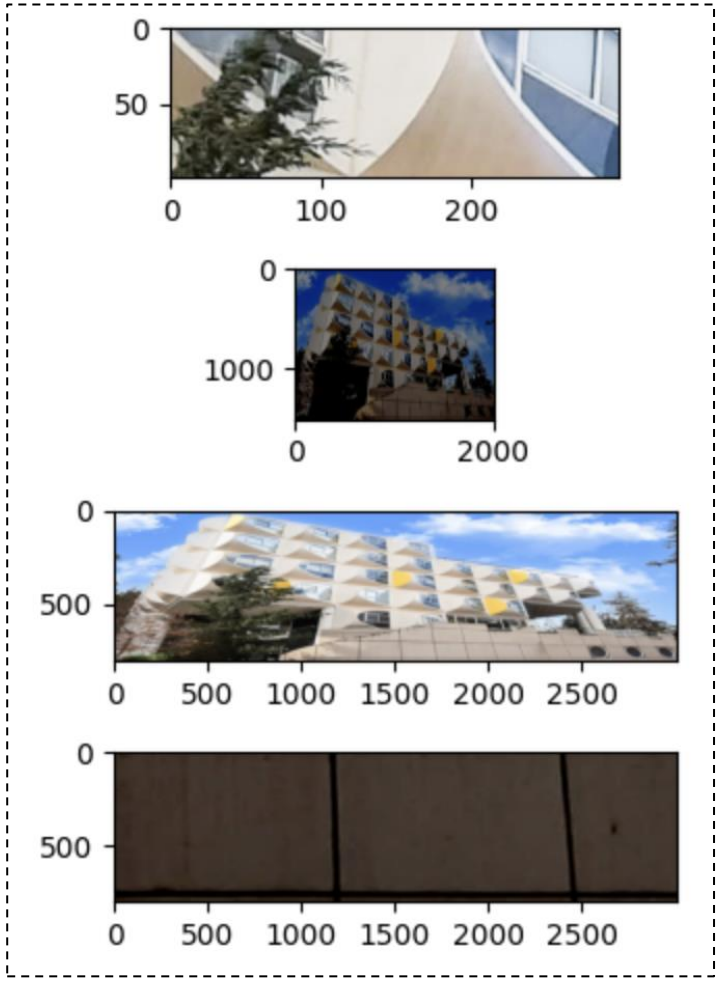
- Resize images
- Input shape: (B, C, H, W)

```
fig, ax = plt.subplots(1, 3, figsize=(20, 5))
inter_img = F.interpolate(img.unsqueeze(0), scale_factor=(3, 1)).squeeze()
ax[0].imshow(inter_img.permute(1, 2, 0))

inter_img = F.interpolate(img.unsqueeze(0), scale_factor=(1, 3)).squeeze()
ax[1].imshow(inter_img.permute(1, 2, 0))

inter_img = F.interpolate(img.unsqueeze(0), size=(50, 50)).squeeze()
ax[2].imshow(inter_img.permute(1, 2, 0))

plt.tight_layout()
plt.show()
```

Image slicing & manipulation

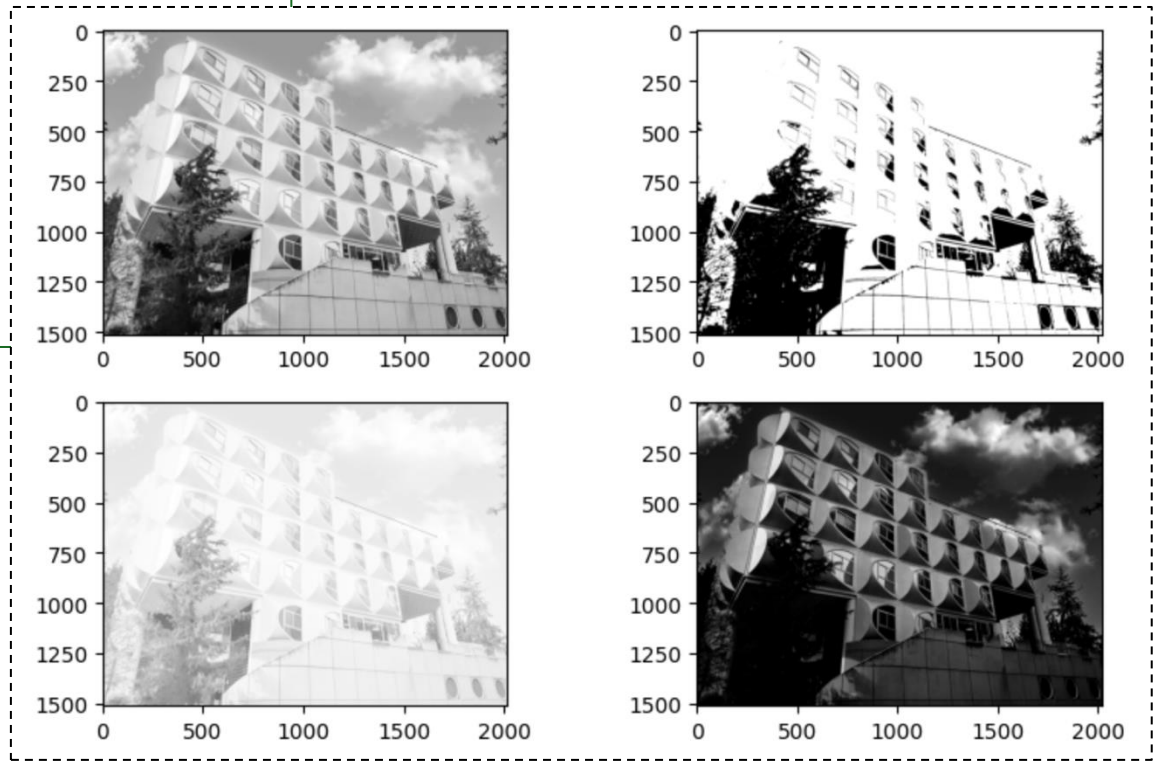
torchvision.transforms

```
from torchvision import transforms
float_img = img.float() / 255
fig, ax = plt.subplots(4, 1)
trans = transforms.RandomCrop((100, 300))
ax[0].imshow(trans(float_img).permute(1, 2, 0))

trans = transforms.Normalize(mean=0.5, std=1)
ax[1].imshow(trans(float_img).permute(1, 2, 0), vmin=0, vmax=1)

trans = transforms.Resize((800, 3000))
ax[2].imshow(trans(float_img).permute(1, 2, 0))

trans = transforms.Compose([transforms.RandomCrop((100, 300)),
                            transforms.Normalize(mean=0.5, std=1),
                            transforms.Resize((800, 3000))])
ax[3].imshow(trans(float_img).permute(1, 2, 0))

plt.tight_layout()
plt.show()
```



Point operations

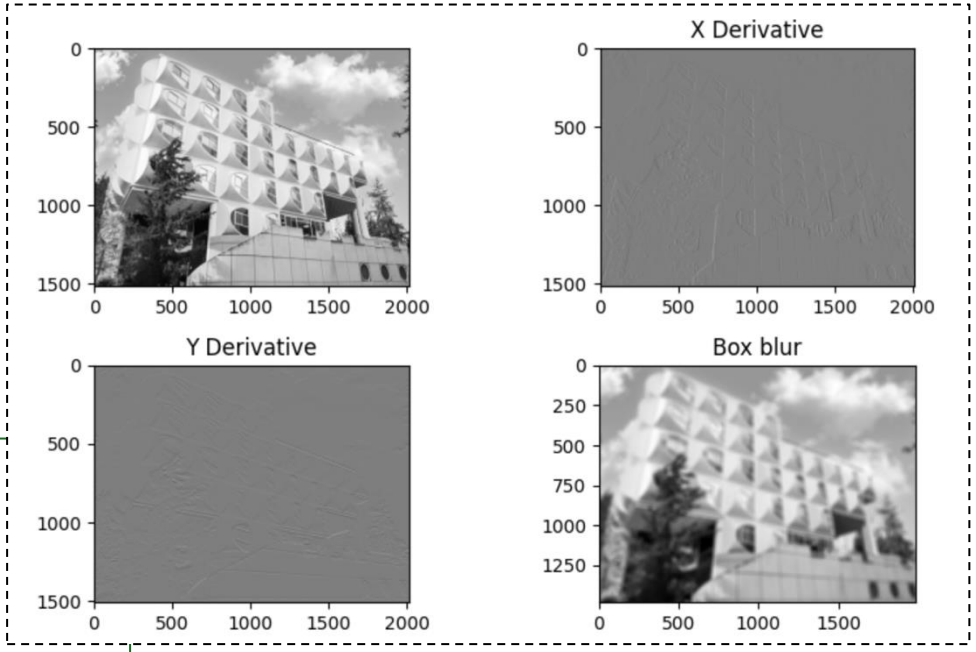
```
gray_img = torchvision.transforms.functional.rgb_to_grayscale(img).float() / 255
fig, ax = plt.subplots(2, 2, figsize=(8, 5))
ax[0, 0].imshow(gray_img[0], cmap="gray")
ax[0, 1].imshow(torch.where(gray_img > 0.5, 1, 0)[0], cmap="gray")

# Gamma correction
ax[1, 0].imshow(gray_img.pow(0.2)[0], cmap="gray")
ax[1, 1].imshow(gray_img.pow(5)[0], cmap="gray")

plt.tight_layout()
plt.show()
```



Convolution 2D

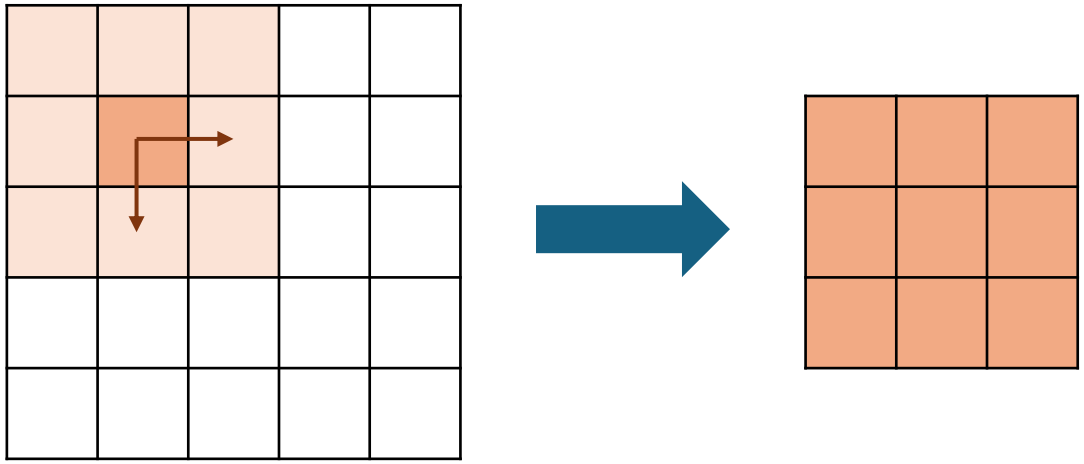
conv2d() instead of **conv1d()**

- Input: (B, C_in, H, W), Kernel: (C_out, C_in, kH, kW)
- For true convolution: flip kernel with **.flip([-2, -1])**

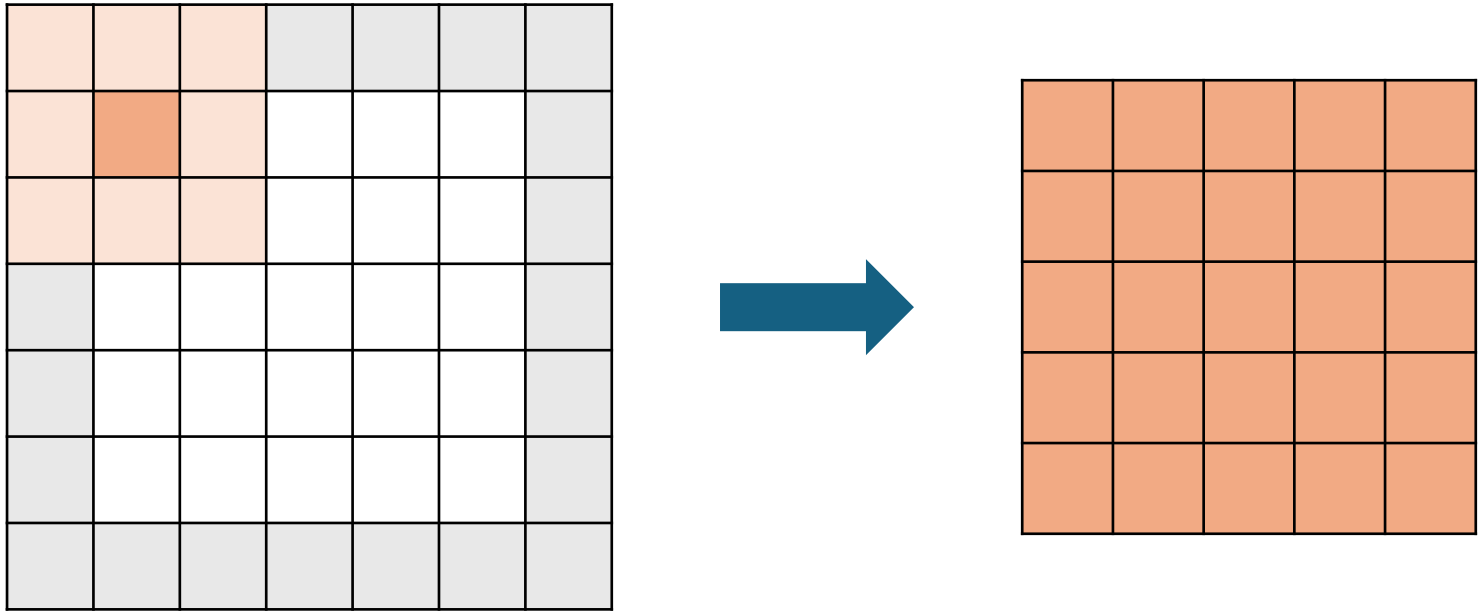



```
gray_img = torchvision.transforms.functional.rgb_to_grayscale(img).float() / 255
fig, ax = plt.subplots(2, 2, figsize=(8, 5))
ax[0, 0].imshow(gray_img[0], cmap="gray")

x_deriv_img = F.conv2d(gray_img.unsqueeze(0), torch.tensor([[[[-1., 0, 1.]]]]))
ax[0, 1].imshow(x_deriv_img.squeeze(), cmap="gray")
ax[0, 1].set_title("X Derivative")

y_deriv_img = F.conv2d(gray_img.unsqueeze(0), torch.tensor([[[[-1.], [0], [1.]]]]))
ax[1, 0].imshow(y_deriv_img.squeeze(), cmap="gray")
ax[1, 0].set_title("Y Derivative")

box_blur_img = F.conv2d(gray_img.unsqueeze(0), torch.ones((1, 1, 30, 30)) / (30 ** 2))
ax[1, 1].imshow(box_blur_img.squeeze(), cmap="gray")
ax[1, 1].set_title("Box blur")

plt.tight_layout()
plt.show()
```

Convolution 2D

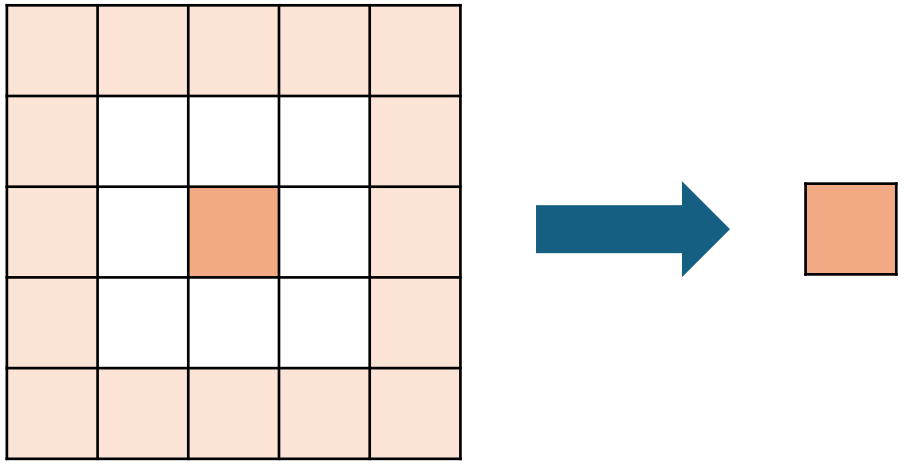


Convolution 2D | padding



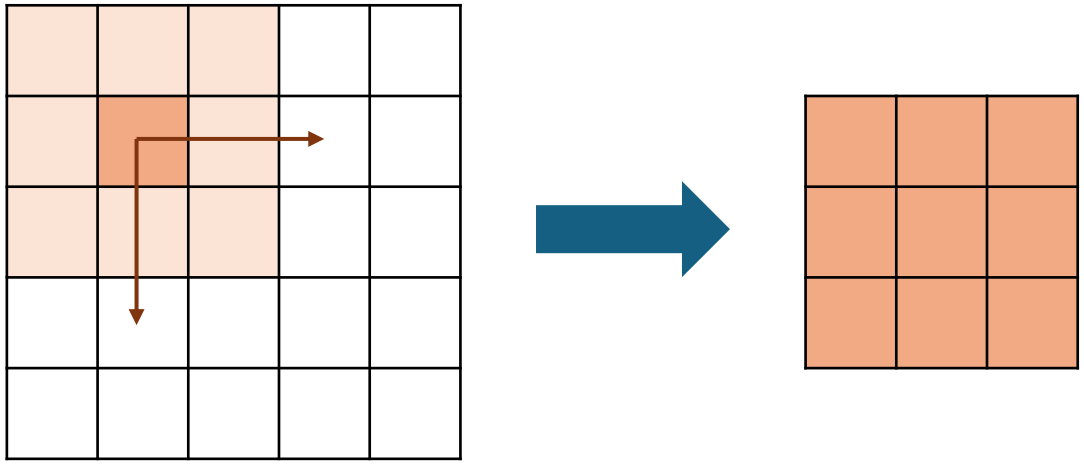
Padding = (1, 1)

Convolution 2D | dilation



Dilation = 2

Convolution 2D | stride



Stride = 2

Batch dimension

Most PyTorch operations expect a batch dimension:

- Single image: (C, H, W) → Batched: **(B, C, H, W)**

Why batches?

- Process multiple images in parallel
- Required by F.conv2d, F.interpolate, neural network layers
- Better GPU utilization — hardware is designed for batched operations

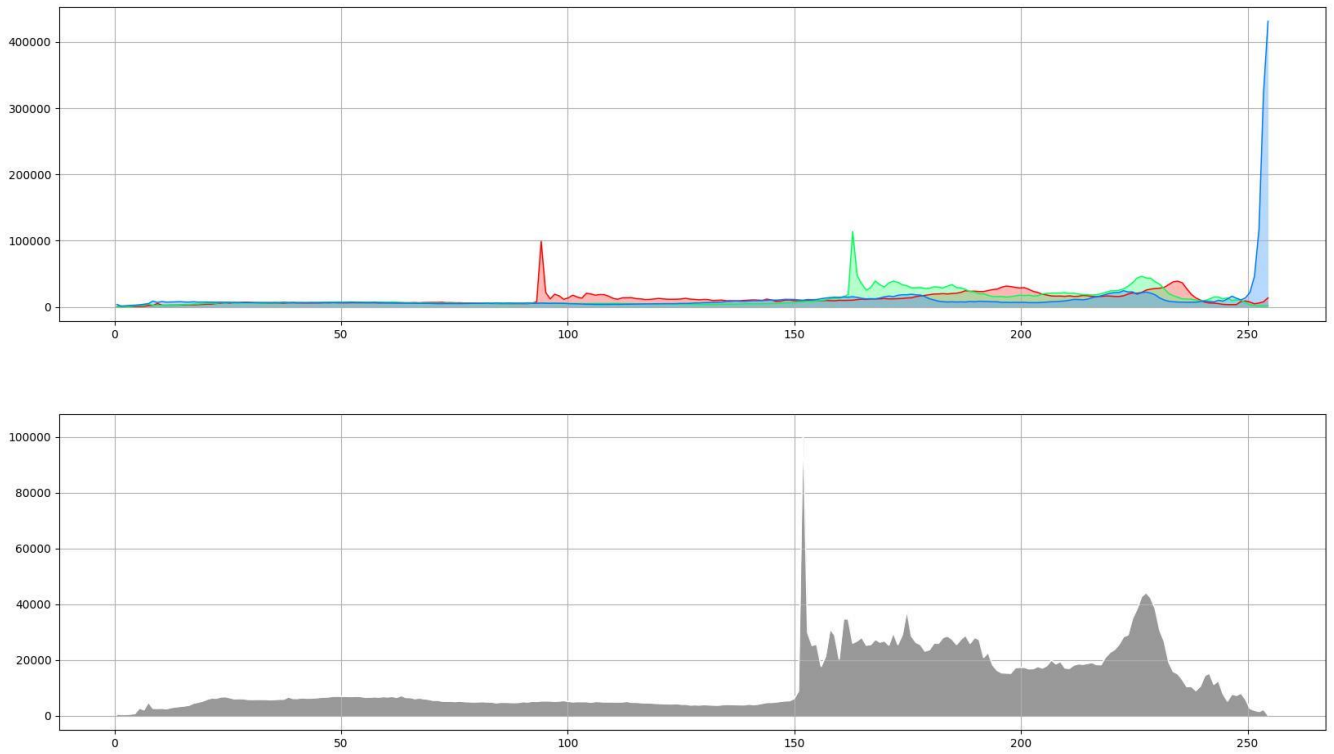
- Adding: **img.unsqueeze(0)** → (1, C, H, W)
- Removing: **output.squeeze(0)** → (C, H, W)
- Stacking: **torch.stack([img1, img2, ...], dim=0)**

Histogram

Histogram: the number of appearances of each element in a set

In images: the number of pixels with (or distribution of) color/gray levels

```
fig = plt.figure(figsize=(20, 5))
plt.grid()
c = img.shape[0]
if c == 1:
        channel_names  = ["Intensity"]
        channel_colors = ["#e0e0e0"]
        channel_edge   = ["#ffffff"]
elif c == 3:
    channel_names  = ["Red", "Green", "Blue"]
    channel_colors = ["#ff4d4d", "#4dff88", "#4d9eff"]
    channel_edge   = ["#ff0000", "#00ff55", "#0077ff"]
for i in range(c):
    pixels = img[i, :, :].float()
    range_min = 0
    range_max = 255
    counts, edges = torch.histogram(pixels, bins=256,
                                    range=(range_min, range_max))
    centers = (edges[:-1] + edges[1:]) / 2
    plt.fill_between(
        centers, counts,
        alpha=0.40,
        color=channel_colors[i],
        linewidth=0,
    )
    plt.plot(
        centers, counts,
        color=channel_edge[i],
        linewidth=1.1,
        label=channel_names[i],
    )

fig.savefig("color_hist.jpeg")
```



Histogram equalization

Step 1 – Compute histogram:

```
hist = torch.histc(gray_img, bins=256, min=0, max=1)
hist[:10]
```

```
tensor([ 454.,  347.,  363.,  507.,  686., 2663., 2053., 4771., 2554., 2543.])
```

Step 2 – Cumulative sum (CDF):

```
cdf = hist.cumsum(dim=0)
cdf[:10]
```

```
tensor([  454.,   801.,  1164.,  1671.,  2357.,  5020.,  7073., 11844., 14398.,
        16941.])
```

Step 3 – Normalize to [0, 1]:

```
cdf_normalized = (cdf - cdf.min()) / (cdf.max() - cdf.min())
cdf_normalized[:10]
```

```
tensor([0.0000, 0.0001, 0.0002, 0.0004, 0.0006, 0.0015, 0.0022, 0.0037, 0.0046,
        0.0054])
```

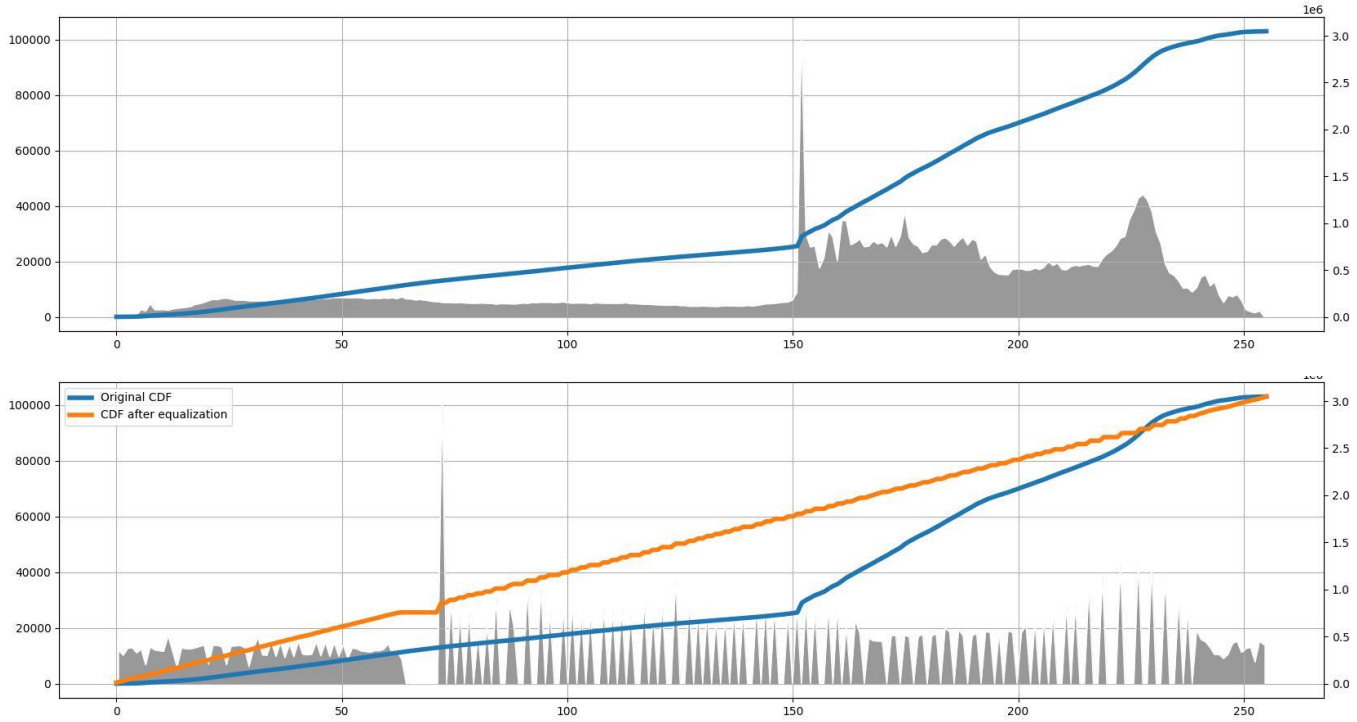
Histogram equalization

Step 4 – Map pixel values:

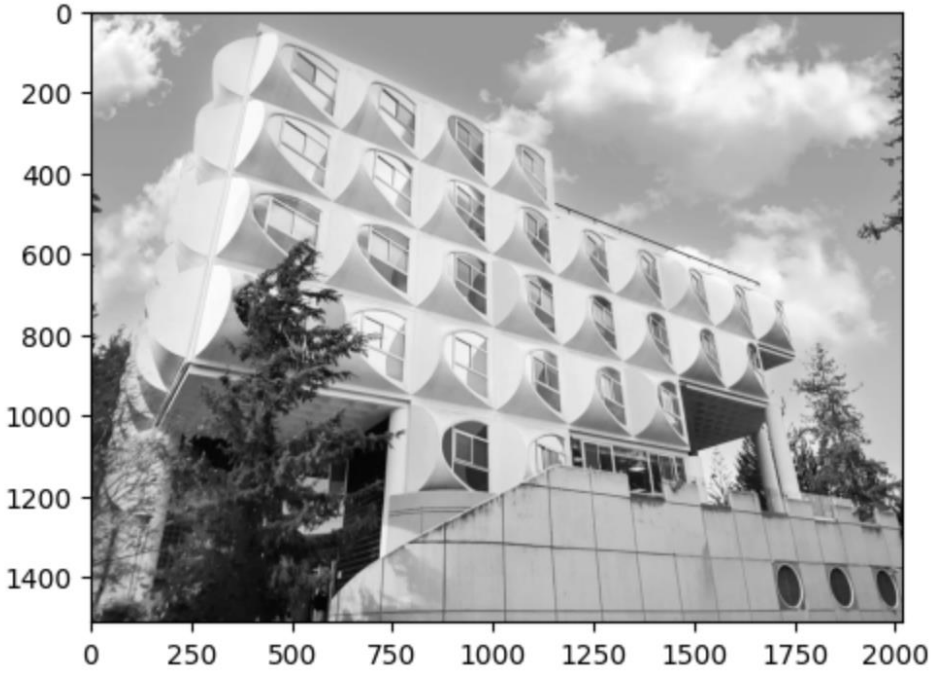
```
indices = (gray_img * 255).long()
equalized = cdf_normalized[indices]
```

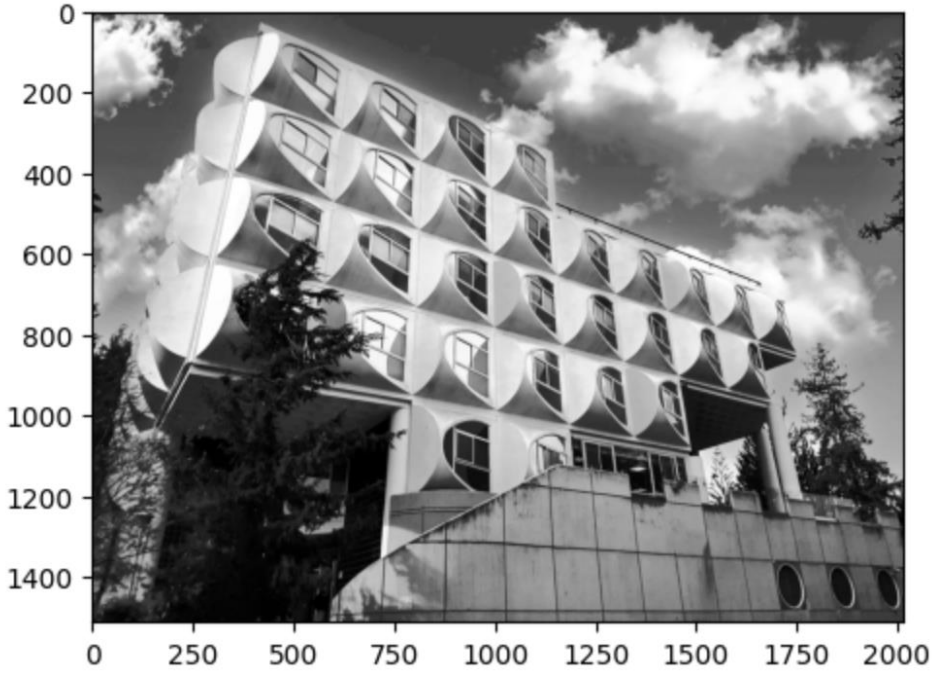
Original histogram
Original CDF

Equalized histogram
Equalized CDF



Histogram equalization





Histogram equalization