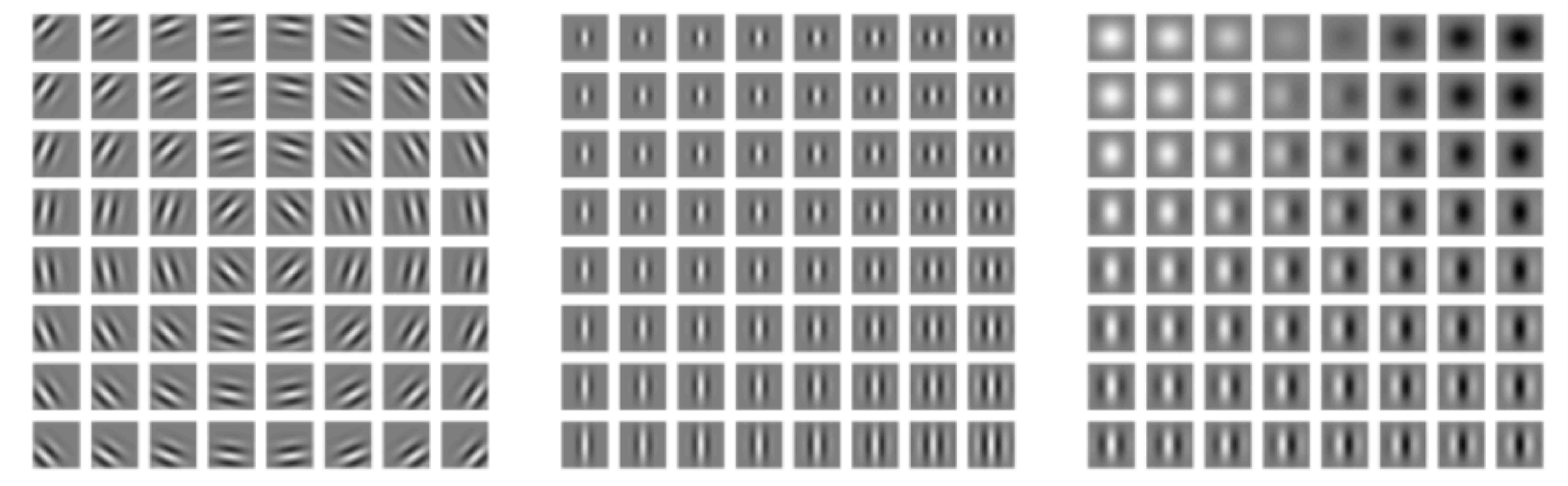
Convolutional Neural Networks (CNN)

David I. Inouye

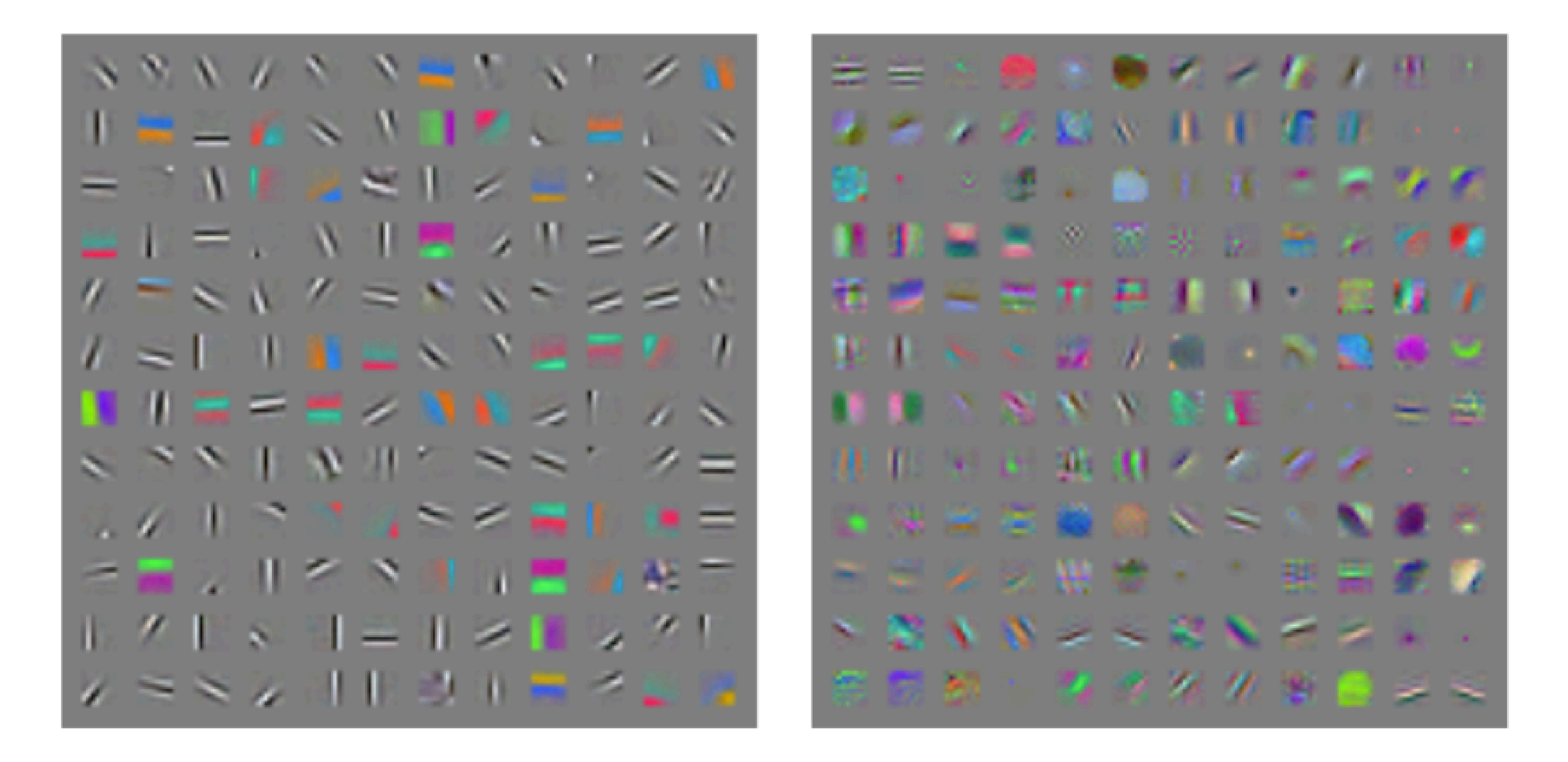
Why Convolution Networks?: Neurocientific Inspiration

Gabor Functions Derived From Neuroscience Experiments Are Simple Convolutional Filters [DL, ch. 9]



Why Convolution Networks?: Neurocientific Inspiration

Convolutional Networks Automatically Learn Filters Similar to Gabor Functions [DL, ch. 9]



Why Convolutional Networks?: Computational Reasons

- Sparse computation (compared to full deep linear networks)
  - Computationally efficient (can be implemented with fast libraries)
  - $O(n \times k)$ instead of $O(n^2)$ for fully connected layer
- Shared parameters (only a small number of shared parameters)
  - Comparison of number of parameters for fully connected vs convolutional layer:
    - Fully connected: $O(n_{in} \times n_{out})$
    - Convolutional: $O(k \times k \times c_{in} \times c_{out})$ where k is kernel size and c is number of channels
  - Fewer parameters $\rightarrow$ less data needed to train
- Translation invariance
  - Convolutional layers can detect features regardless of their position

1D Convolutions Are Similar but Slightly Different Than Signal Processing / Math Convolutions

x	1	2	3	2	5	1
f	1	2				
y	5	8	7	12	7	

Padding or Stride Parameters Alter the Computation and Output Shape

x						
	1	2	3	2	5	1

f		
	1	2

Stride of 2

y			
	5	7	7

1D Convolutions With Padding

x

1	2	3	2	5	1

f

1	2

Zero padding of 1

y

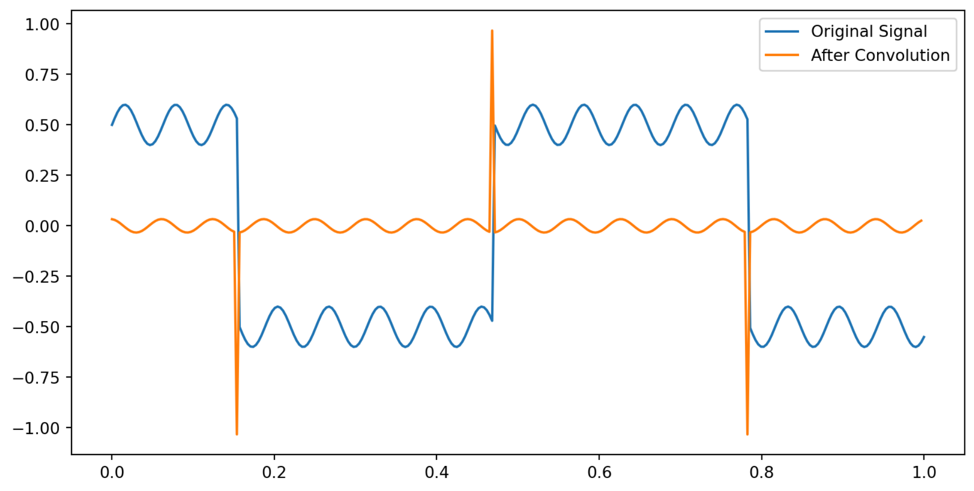
2	5	8	7	12	7	1

1D Demo: 1D convolutions, similar but slightly different than signal processing / math convolutions

[-1, 1] filter/kernel highlights “sharp points” of signal

```python
import torch
import matplotlib.pyplot as plt
%matplotlib inline

t = torch.linspace(0, 1.0, 300)
x = (torch.cos(10*t) > 0.0).float() + 0.1*torch.sin(100*t)-0.
plt.plot(t.numpy(), x.numpy(), label='Original Signal')

from torch.nn import functional as F
filt = torch.tensor([-1, 1.0])
print('Filter')
print(filt)
# Should have shape $(m, c, w)$ where m is minibatch size, c
y = F.conv1d(
    x.reshape(1, 1, len(x)),
    filt.reshape(1, 1, len(filt))
).squeeze_()
plt.plot(
    t.numpy()[:len(y)], y.numpy(),
    label='After Convolution')
plt.legend()
```

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



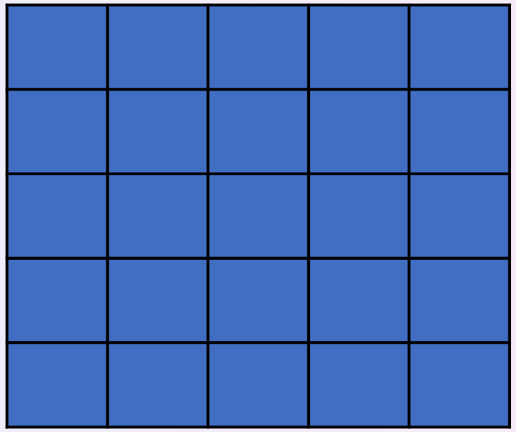
Convolutions are linear operators (i.e., matrix multiplication) with shared parameters

```python
x = torch.randn(10).float().requires_grad_(True)
filt = torch.tensor([-1, 1]).float()
#filt = torch.tensor([1, 2, 3, 4]).float()
y = F.conv1d(x.reshape(1, 1, len(x)), filt.reshape(1, 1, len(

def extract_jacobian(x, y):
    J = torch.zeros((len(y), len(x))).float()
    for i in range(len(y)):
        v = torch.zeros(len(y)).float()
        v[i] = 1
        if x.grad is not None:
            x.grad.zero_()
        y.backward(v, retain_graph=True)
        J[i, :] = x.grad
    return J

A = extract_jacobian(x, y)
print(A)
y2 = torch.matmul(A, x)
print(y)
print(y2)
print(y-y2)
```

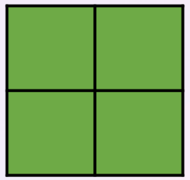
```
tensor([[-1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0., -1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0., -1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0., -1.,  1.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0., -1.,  1.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0., -1.,  1.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0., -1.,  1.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0., -1.,  1.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0., -1.,  1.]])
tensor([-1.8103,  1.4400, -1.4673,  0.5769,  0.7165, -1.4310,
0.7316,  0.0311,
         1.2744], grad_fn=<SqueezeBackward3>)
tensor([-1.8103,  1.4400, -1.4673,  0.5769,  0.7165, -1.4310,
0.7316,  0.0311,
         1.2744], grad_fn=<MvBackward0>)
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0.], grad_fn=
<SubBackward0>)
```

2D Convolutions Are Simple Generalizations to Matrices

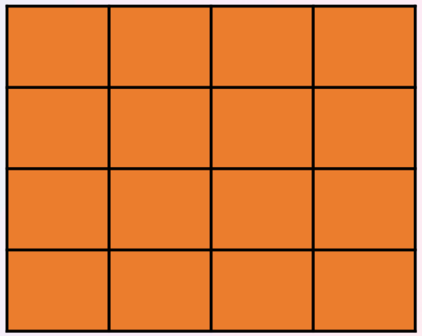
x



f

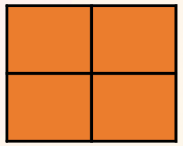


y



Stride of 2

y

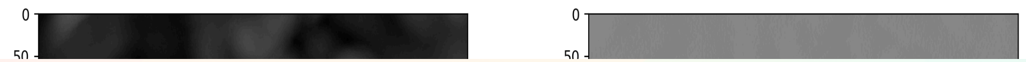


2D convolutions are similar and can be applied to images

Different filters extract different features from the image

```python
import sklearn.datasets
A = torch.tensor(sklearn.datasets.load_sample_image('china.jp
A = torch.tensor(sklearn.datasets.load_sample_image('flower.j
A = torch.sum(A, dim=2) # Sum channels

for filt in [
    torch.tensor([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]]).float(
    torch.tensor([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]]).float(
    torch.tensor([[1, -1], [-1, 1]]).float(), # Checker board
    torch.ones((10, 10)).float(), # Blur
]:
    print('Filter size', filt.size(), 'A size', A.size())
    print(filt)
    B = F.conv2d(A.reshape(1, 1, *A.size()), filt.reshape(1,
    #B = F.conv2d(A.reshape(1, 1, *A.size()), filt.reshape(1,

    fig, axes = plt.subplots(1, 2, figsize=(14,4))
    axes[0].imshow(A.numpy(), cmap='gray')
    axes[1].imshow(B.numpy(), cmap='gray')
```

```
Filter size torch.Size([3, 3]) A size torch.Size([427, 640])
tensor([[-1.,  0.,  1.],
        [-1.,  0.,  1.],
        [-1.,  0.,  1.]])
Filter size torch.Size([3, 3]) A size torch.Size([427, 640])
tensor([[-1., -1., -1.],
        [ 0.,  0.,  0.],
        [ 1.,  1.,  1.]])
Filter size torch.Size([2, 2]) A size torch.Size([427, 640])
tensor([[ 1., -1.],
        [-1.,  1.]])
Filter size torch.Size([10, 10]) A size torch.Size([427,
640])
tensor([[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.],
        [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.],
        [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.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])
```

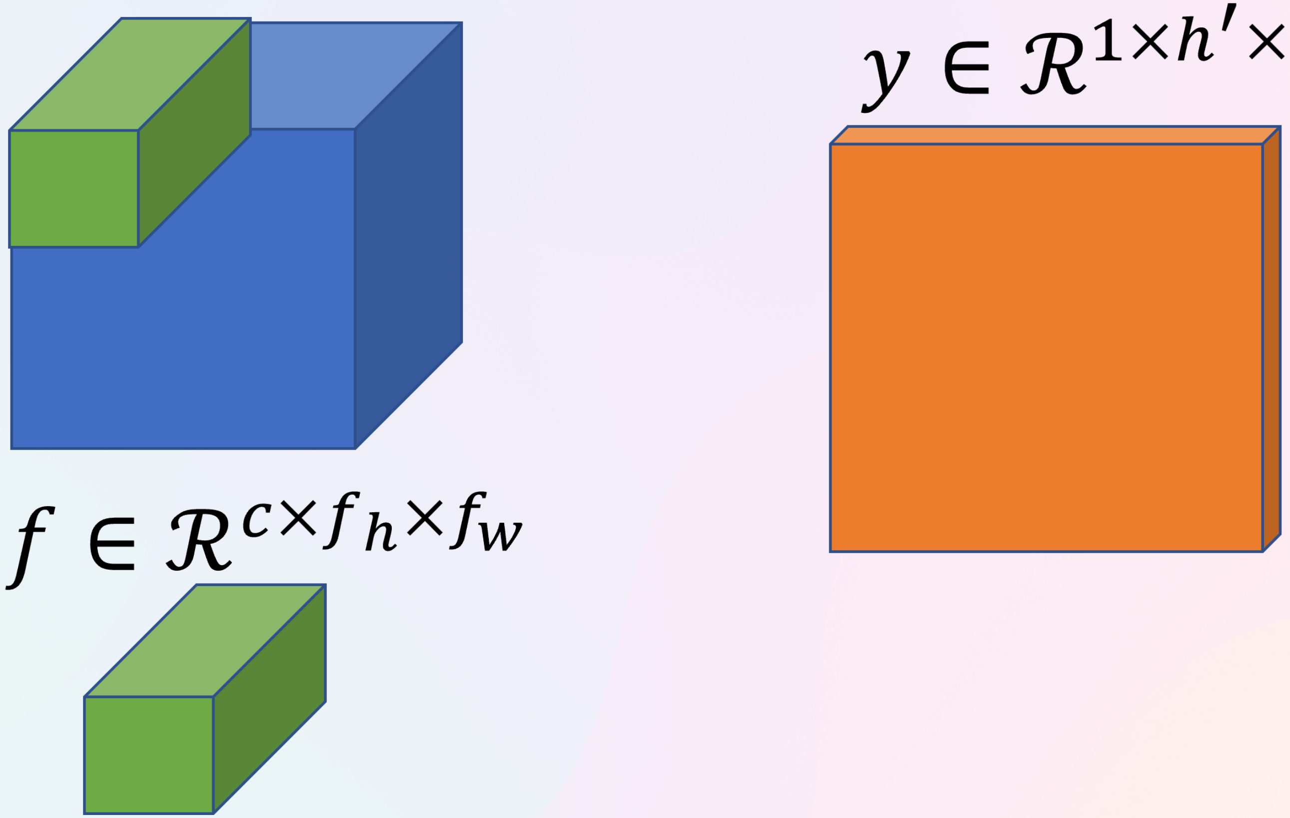


2D Convolutions With Channels Are Like Simple 2D Convolutions but All Arrays Have a Channel Dimension

$x \in \mathcal{R}^{c \times h \times w}$

$y \in \mathcal{R}^{1 \times h' \times w'}$

$f \in \mathcal{R}^{c \times f_h \times f_w}$

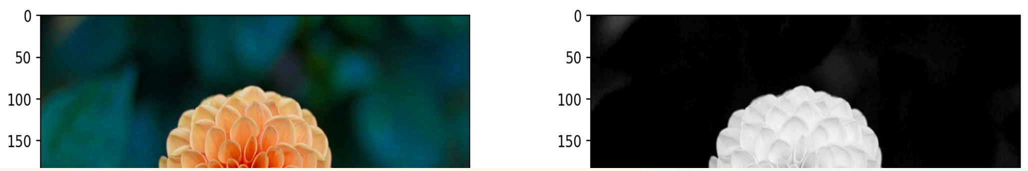


"$f_h \times f_w$ convolution" (channel dimension is assumed)

2D convolutions with channel dimension are similar (i.e., if there is more than 1 channel)

```python
A = torch.tensor(sklearn.datasets.load_sample_image('flower.j
A = A/255
A = A.permute(2,0,1)

for filt in [
    torch.tensor([1, 0, 0]).reshape(3, 1, 1).float(),
    torch.tensor([0, 1, 0]).reshape(3, 1, 1).float(),
    torch.tensor([0, 0, 1]).reshape(3, 1, 1).float(),
]:
    print('Filter size', filt.size(), 'A size', A.size(), 'B
    print(filt)
    B = F.conv2d(
        A.reshape(1, *A.size()),
        filt.reshape(1, *filt.size())
    ).squeeze()

    fig, axes = plt.subplots(1, 2, figsize=(14,4))
    axes[0].imshow(A.permute(1,2,0), cmap='gray')
    axes[1].imshow(B, cmap='gray')
```

```
Filter size torch.Size([3, 1, 1]) A size torch.Size([3, 427,
640]) B size torch.Size([420, 633])
tensor([[[1.]],

        [[0.]],

        [[0.]]])
Filter size torch.Size([3, 1, 1]) A size torch.Size([3, 427,
640]) B size torch.Size([427, 640])
tensor([[[0.]],

        [[1.]],

        [[0.]]])
Filter size torch.Size([3, 1, 1]) A size torch.Size([3, 427,
640]) B size torch.Size([427, 640])
tensor([[[0.]],

        [[0.]],

        [[1.]]])
```

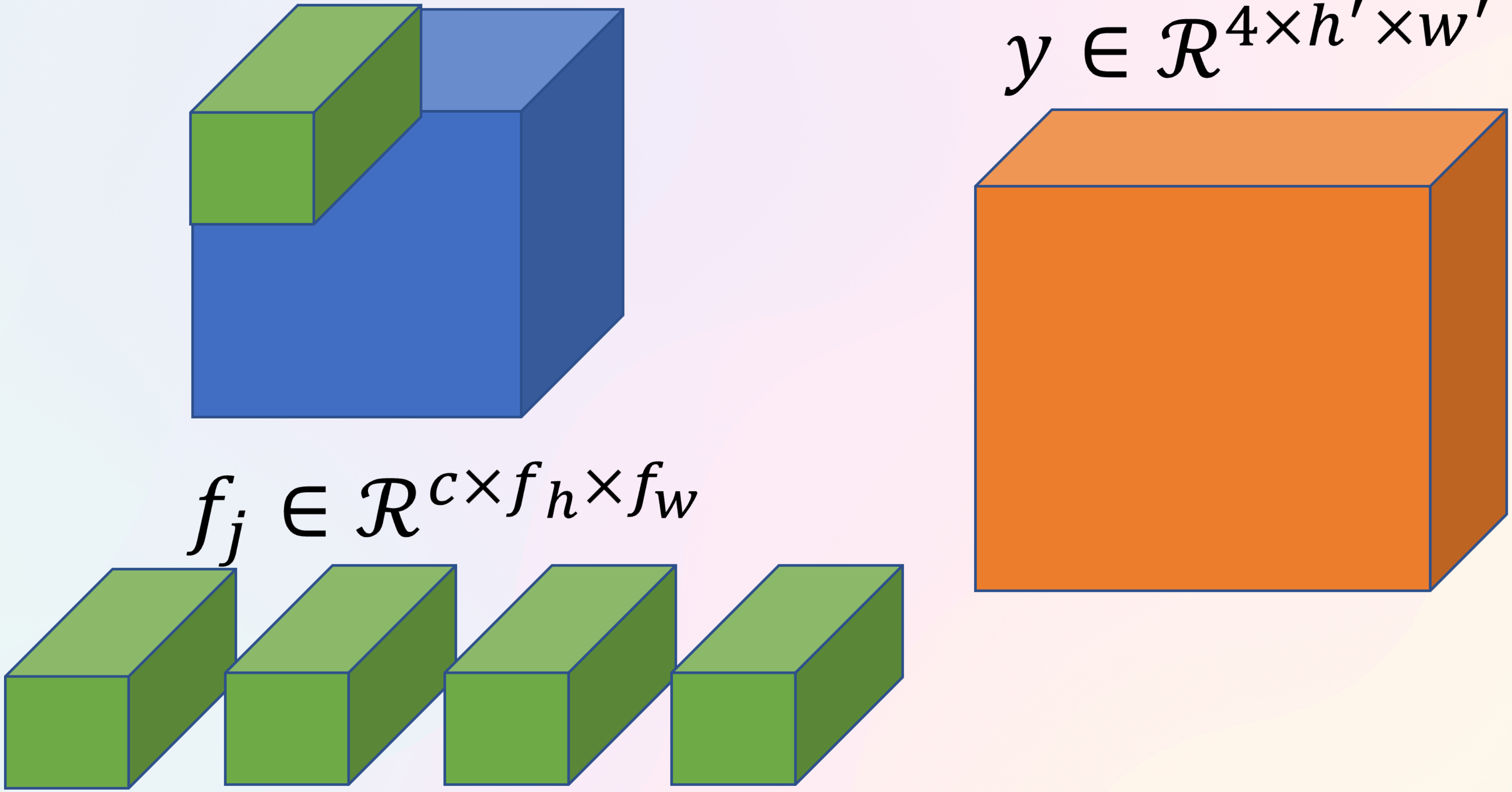


Multiple Convolutions Increase the Output Channel Dimension

$x \in \mathcal{R}^{c \times h \times w}$

$y \in \mathcal{R}^{4 \times h' \times w'}$

$f_j \in \mathcal{R}^{c \times f_h \times f_w}$



Reasoning About Input and Output Shapes Is Important for Debugging and Designing CNNs

- **Convolution input parameters**
  - $ChannelIn = C_{in}$
  - $ChannelOut = C_{out}$ (equivalent to # filters)
  - $KernelSize = [K_0, K_1]$
  - $Stride = [S_0, S_1]$
  - $Padding = [P_0, P_1]$
- C_{out} = # filters
- **Output spatial dimensions**
  - $$H_{out} = \lfloor \frac{H_{in} + 2P_0 - K_0}{S_0} + 1 \rfloor$$
  - $$W_{out} = \lfloor \frac{W_{in} + 2P_1 - K_1}{S_1} + 1 \rfloor$$
- **Output batch dimension should match input**

Common Convolution Configurations

$$H_{out} = \lfloor \frac{H_{in} + 2P_0 - K_0}{S_0} + 1 \rfloor$$

- **Output has same height and width as input**
  - 1×1 convolution with padding=0, stride=1
  - 3×3 convolution with padding=1, stride=1
  - 5×5 convolution with padding=2, stride=1
- **Output has half the height and width of input**
  - 2×2 convolution with padding=0, stride=2
  - 4×4 convolution with padding=1, stride=2

Need several other components for extracting features

- Activation functions
- Pooling layers

Why activation functions? Activation functions enable non-linear models

Consider a deep linear network

```python
torch.manual_seed(0)
A1 = torch.randn((10, 5))
A2 = torch.randn((10, 10))
A3 = torch.randn((1, 10))

x = torch.randn(5)
print('x', x)
y = torch.matmul(A1, x)
y = torch.matmul(A2, y)
y = torch.matmul(A3, y)
print('y', y)

b = torch.matmul(A3, torch.matmul(A2, A1))
y2 = torch.matmul(b, x)
print('y2', y2)
```

```
x tensor([ 1.4875, -0.2230, -1.0057, -0.4139,  1.1600])
y tensor([4.1752])
y2 tensor([4.1752])
```

Why activation functions? Activation functions enable non-linear models

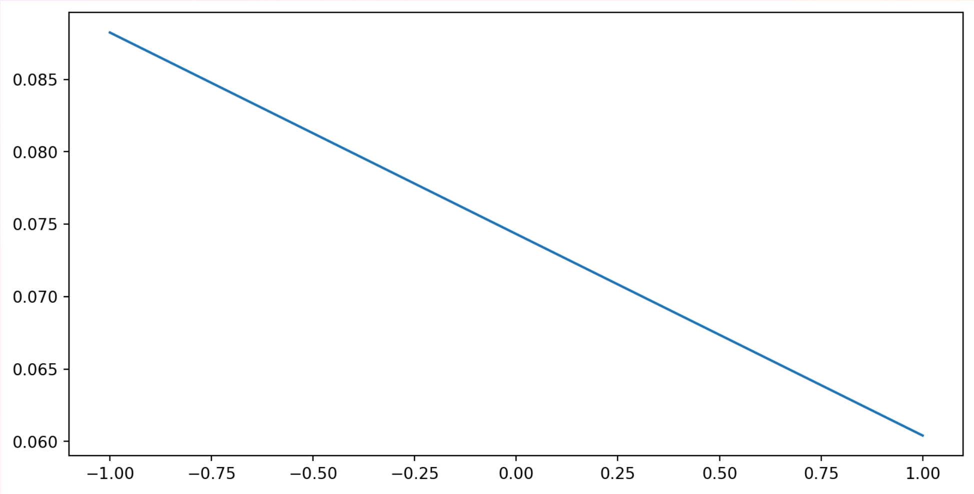
If you add activation functions, the deep function cannot be simplified

```
torch.manual_seed(0)
A1 = torch.randn((10, 5))
A2 = torch.randn((10, 10))
A3 = torch.randn((1, 10))

x = torch.randn(5)
print('x', x)
y = torch.matmul(A1, x)
y = torch.relu(y)
y = torch.matmul(A2, y)
y = torch.relu(y)
y = torch.matmul(A3, y)
print('y', y)

b = torch.matmul(A3, torch.matmul(A2, A1))
y2 = torch.matmul(b, x)
print('y2', y2)
```

```
x tensor([ 1.4875, -0.2230, -1.0057, -0.4139,  1.1600])
y tensor([18.9449])
y2 tensor([4.1752])
```

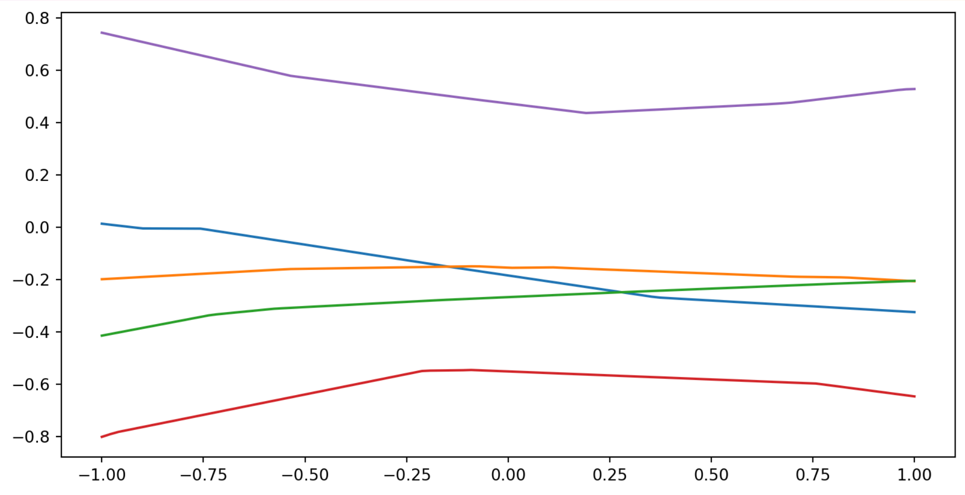
Without ReLU or activation function, the function can only be linear

```
N, D_in, H, D_out = 64, 1, 10, 1
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.Linear(H, D_out),
)
x = torch.linspace(-1, 1, 100).reshape(-1, 1)
y = model(x)
plt.plot(x.detach().numpy(), y.detach().numpy())
```



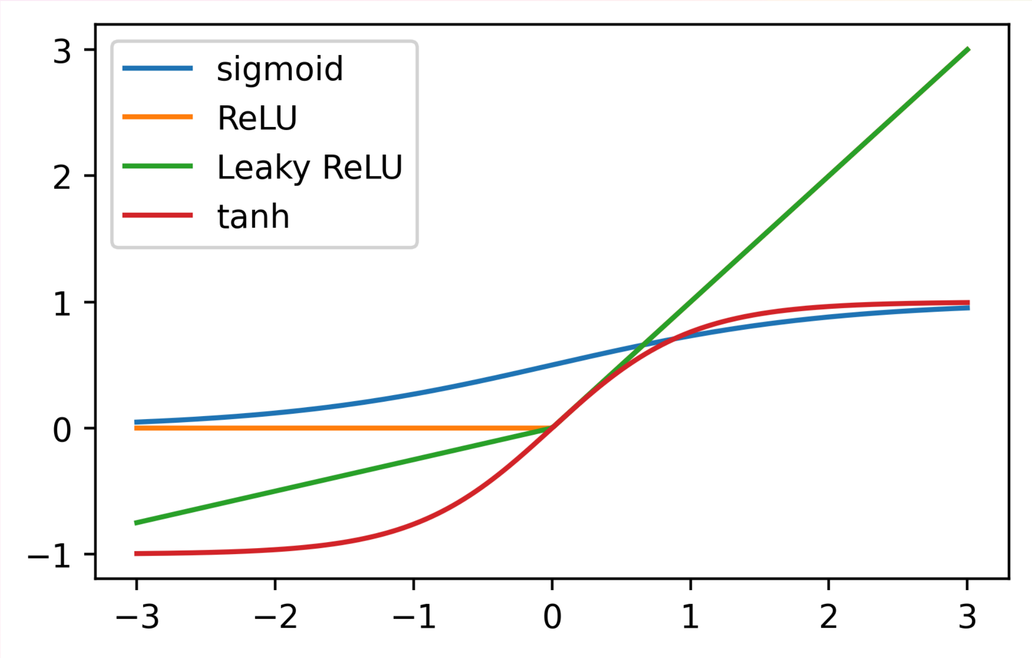
With ReLU activation function, the function is *piecewise* linear

```
N, D_in, H, D_out = 64, 1, 10, 1
for random_seed in [0, 1, 2, 3, 4]:
    torch.manual_seed(random_seed)
    model = torch.nn.Sequential(
        torch.nn.Linear(D_in, H),
        torch.nn.ReLU(),
        torch.nn.Linear(H, D_out),
    )
    x = torch.linspace(-1, 1, 100).reshape(-1, 1)
    y = model(x)
    plt.plot(x.detach().numpy(), y.detach().numpy())
```



Common activation functions include sigmoid, ReLU, Leaky ReLU, tanh

```
t = torch.linspace(-3, 3, 300)
fig = plt.figure(figsize=(5,3), dpi=200)
plt.plot(t.numpy(), torch.sigmoid(t).numpy(), label='sigmoid'
plt.plot(t.numpy(), F.relu(t).numpy(), label='ReLU')
plt.plot(t.numpy(), F.leaky_relu(t, negative_slope=0.25).nump
plt.plot(t.numpy(), torch.tanh(t).numpy(), label='tanh')
plt.legend()
```



Pooling layers are used to reduce dimensionality and introduce some location invariance

Max pooling layers

```
torch.manual_seed(0)
x = torch.randint(10, (10,)).float()
y = F.max_pool1d(x.reshape(1,1,-1), kernel_size=3)
y2 = F.max_pool1d(x.reshape(1,1,-1), kernel_size=3, stride=1)
y3 = F.max_pool1d(x.reshape(1,1,-1), kernel_size=3, stride=1,
print(x)
print(y)
print(y2)
print(y3)
```

```
tensor([4., 9., 3., 0., 3., 9., 7., 3., 7., 3.])
tensor([[[9., 9., 7.]]])
tensor([[[9., 9., 3., 9., 9., 9., 7., 7.]]])
tensor([[[9., 9., 9., 3., 9., 9., 9., 7., 7., 7.]]])
```

Pooling layers are used to reduce dimensionality and introduce some location invariance

Average pooling layers

```
torch.manual_seed(0)
x = torch.randint(10, (10,)).float()
y = F.avg_pool1d(x.reshape(1,1,-1), kernel_size=3)
y2 = F.avg_pool1d(x.reshape(1,1,-1), kernel_size=3, stride=1)
y3 = F.avg_pool1d(x.reshape(1,1,-1), kernel_size=3, stride=1,
print(x)
print(y)
print(y2)
print(y3)
```

```
tensor([4., 9., 3., 0., 3., 9., 7., 3., 7., 3.])
tensor([[[5.3333, 4.0000, 5.6667]]])
tensor([[[5.3333, 4.0000, 2.0000, 4.0000, 6.3333, 6.3333,
5.6667, 4.3333]]])
tensor([[[4.3333, 5.3333, 4.0000, 2.0000, 4.0000, 6.3333,
6.3333, 5.6667,
          4.3333, 3.3333]]])
```

- Is average pooling a linear or non-linear operation?
- Is max pooling a linear or non-linear operation?

The shape of pooling layers is slightly different than for convolutions

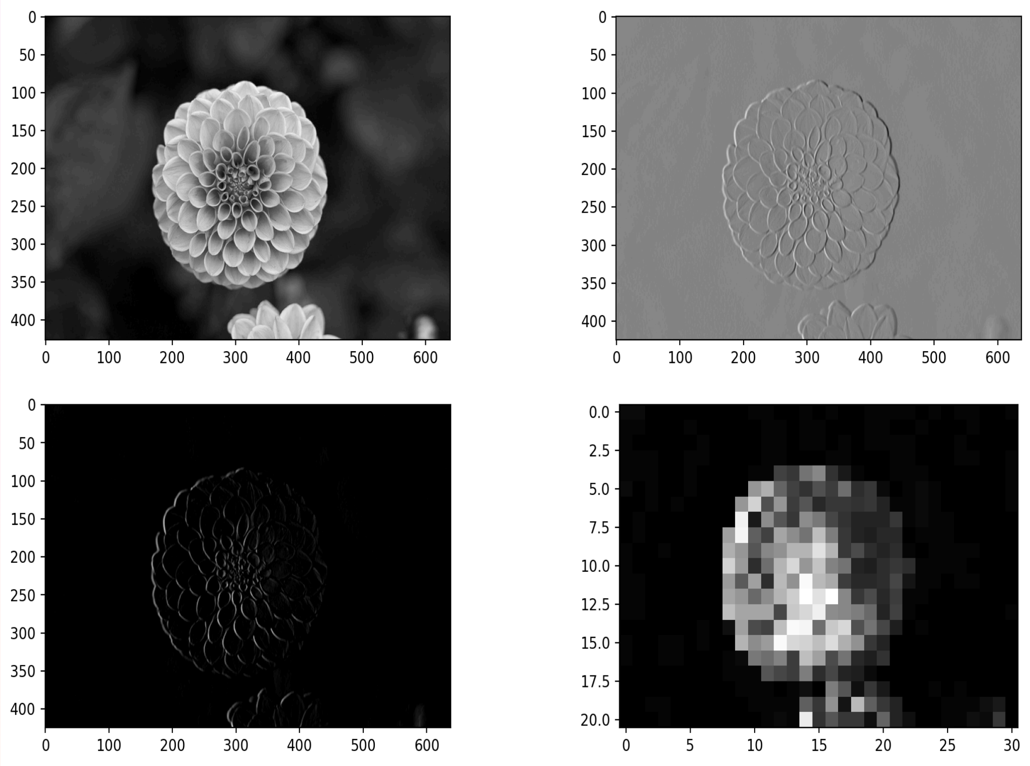
```python
x = torch.randn((3,4,10,20)).float()
print(x.shape, 'N x C x H x W')
y = F.max_pool2d(x, kernel_size=2)
print(y.shape, 'The number of channels does not change for po
y2 = F.max_pool2d(x, kernel_size=2)
print(y2.shape, 'Note that `stride=kernel_size` by default')
y3 = F.max_pool2d(x, kernel_size=2, stride=1)
print(y3.shape, 'Can set stride explicitly to 1')
y4 = F.max_pool2d(x, kernel_size=3, stride=1, padding=1)
print(y4.shape, 'Can produce the same size')
```

```
torch.Size([3, 4, 10, 20]) N x C x H x W
torch.Size([3, 4, 5, 10]) The number of channels does not
change for pooling
torch.Size([3, 4, 5, 10]) Note that `stride=kernel_size` by
default
torch.Size([3, 4, 9, 19]) Can set stride explicitly to 1
torch.Size([3, 4, 10, 20]) Can produce the same size
```

Convolution Neural Network (CNN) layers are compositions of convolution, activation and pooling

```python
import sklearn.datasets
A = torch.tensor(sklearn.datasets.load_sample_image('flower.j
A = torch.sum(A, dim=2)
filt = torch.tensor([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]]).flo
#filt = torch.tensor([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]]).fl
#filt = torch.tensor([[1, -1], [-1, 1]]).float() # Checker bo
#filt = torch.ones((10, 10)).float() # Blur
print('Filter')
print(filt)
B = F.conv2d(A.reshape(1, 1, *A.size()), filt.reshape(1, 1, *
print('A size', A.size(), 'B size', B.size())
C = torch.relu(B)
D = torch.max_pool2d(C, kernel_size=20)
#D = torch.max_pool2d(C, kernel_size=20, stride=1)

fig, axes = plt.subplots(2, 2, figsize=(14,8))
axes = axes.ravel()
for im, ax in zip([A, B, C, D], axes):
    ax.imshow(im.squeeze(), cmap='gray')
```

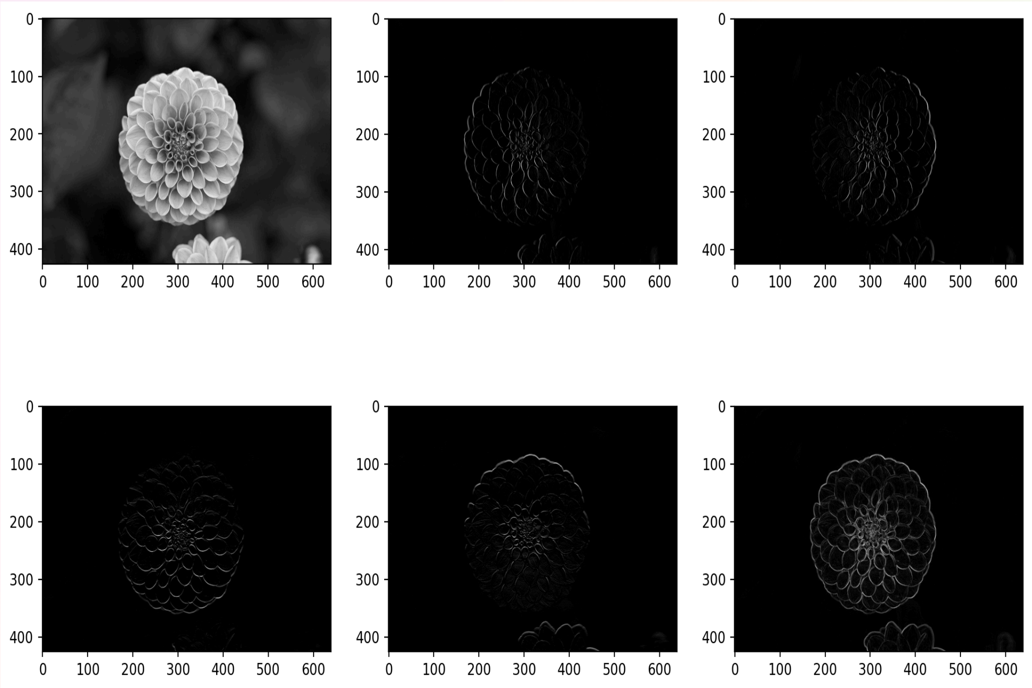
```
Filter
tensor([[-1.,  0.,  1.],
        [-1.,  0.,  1.],
        [-1.,  0.,  1.]])
A size torch.Size([427, 640]) B size torch.Size([1, 1, 425,
638])
```



How could you detect an edge from multiple angles by combining convolutions and ReLUs?

- Hint: First detect edges from all directions, then combine.

```
import sklearn.datasets
import torch
import numpy as np
A = torch.tensor(sklearn.datasets.load_sample_image('china.jp
A = torch.tensor(sklearn.datasets.load_sample_image('flower.j
A = torch.sum(A, dim=2)

filters = torch.tensor([
    [[[-1, 1], [-1, 1]]],
    [[[1, -1], [1, -1]]],
    [[[1, 1], [-1, -1]]],
    [[[-1, -1], [1, 1]]],
]).float()
B = F.conv2d(A.reshape(1, 1, *A.size()), filters)
C = torch.relu(B)

# Combine
filt = torch.ones(4).float()
D = F.conv2d(C, filt.reshape(1, 4, 1, 1))

fig, axes = plt.subplots(2, 3, figsize=(14,8))
```



Check out PyTorch tutorial on simple classifier on CIFAR10 dataset:

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

Transposed Convolution Can Be Used to **Upsample** a Tensor/Image to Have Higher Dimensions

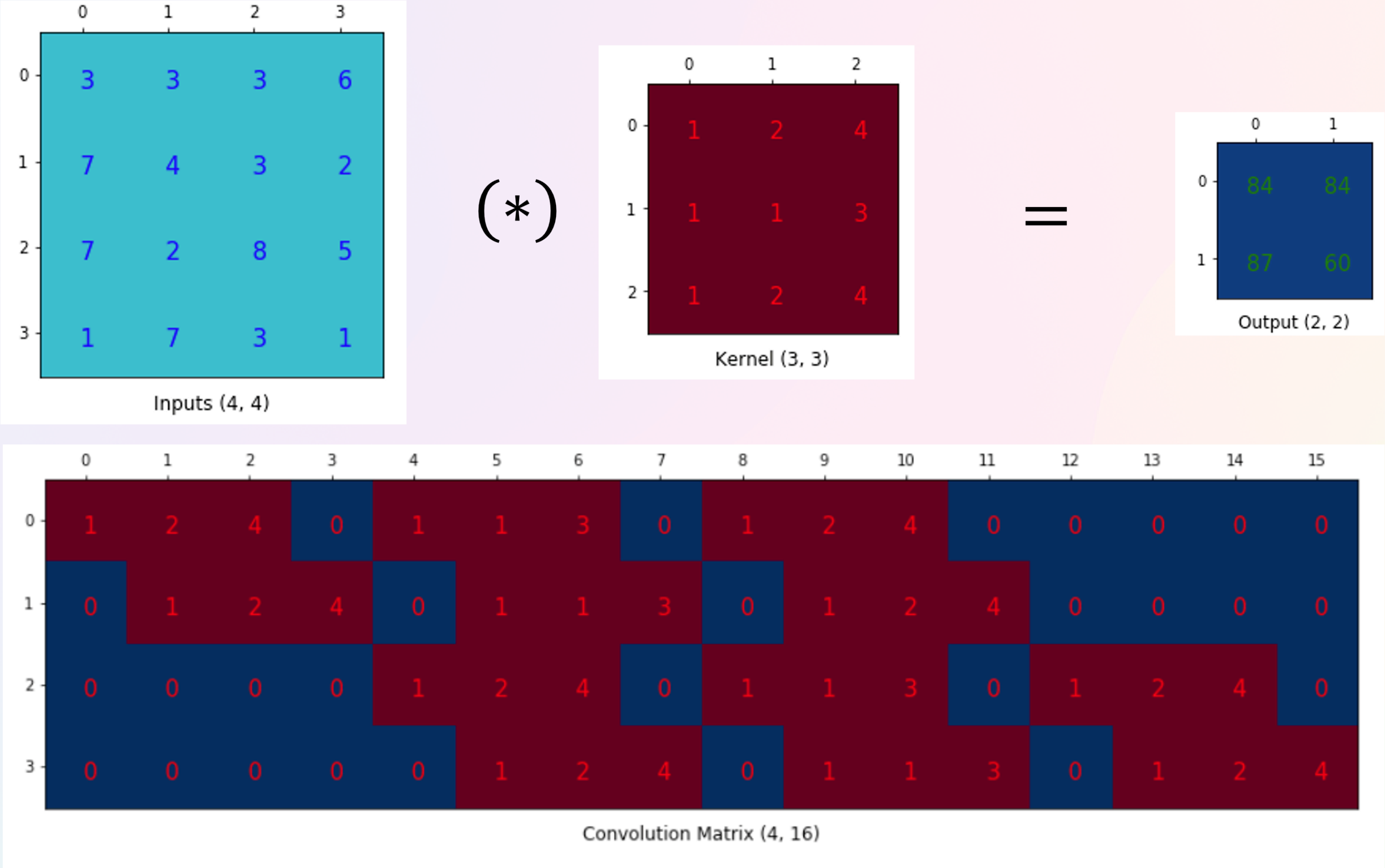
- **Also known as:**
  - **Fractionally-strided convolution**
  - Improperly, **deconvolution**
- **Remember:** Convolution is like matrix multiplication

$$y = x * f \iff \mathrm{vec}(y) = A_f \mathrm{vec}(x)$$

- **Transpose convolution** is the transpose of A_f:

$$\mathrm{vec}(y) = A_f^T \mathrm{vec}(x)$$

Convolution Operator With Corresponding Matrix



https://github.com/naokishibuya/deep-learning/blob/master/python/transposed_convolution.ipynb

Transposed Convolution Operator With Corresponding Matrix

$$
\begin{bmatrix}
1 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 \\
4 & 2 & 0 & 0 \\
0 & 4 & 0 & 0 \\
1 & 0 & 1 & 0 \\
1 & 1 & 2 & 1 \\
3 & 1 & 4 & 2 \\
0 & 3 & 0 & 4 \\
1 & 0 & 1 & 0 \\
2 & 1 & 1 & 1 \\
4 & 2 & 3 & 1 \\
0 & 4 & 0 & 3 \\
0 & 0 & 1 & 0 \\
0 & 0 & 2 & 1 \\
0 & 0 & 4 & 2 \\
0 & 0 & 0 & 4
\end{bmatrix}
*
\begin{bmatrix} 2 \\ 3 \\ 4 \\ 3 \end{bmatrix}
=
\begin{bmatrix} 2 \\ 7 \\ 14 \\ 12 \\ 6 \\ 16 \\ 31 \\ 21 \\ 6 \\ 14 \\ 29 \\ 21 \\ 4 \\ 11 \\ 22 \\ 12 \end{bmatrix}
$$

Transposed Convolution Matrix (16, 4)

Inputs (4, 1)

Output (16, 1)

Reshaped input

$$
\begin{bmatrix} 1 & 3 \\ 2 & 3 \end{bmatrix}
$$

Inputs (2, 2)

Reshaped output

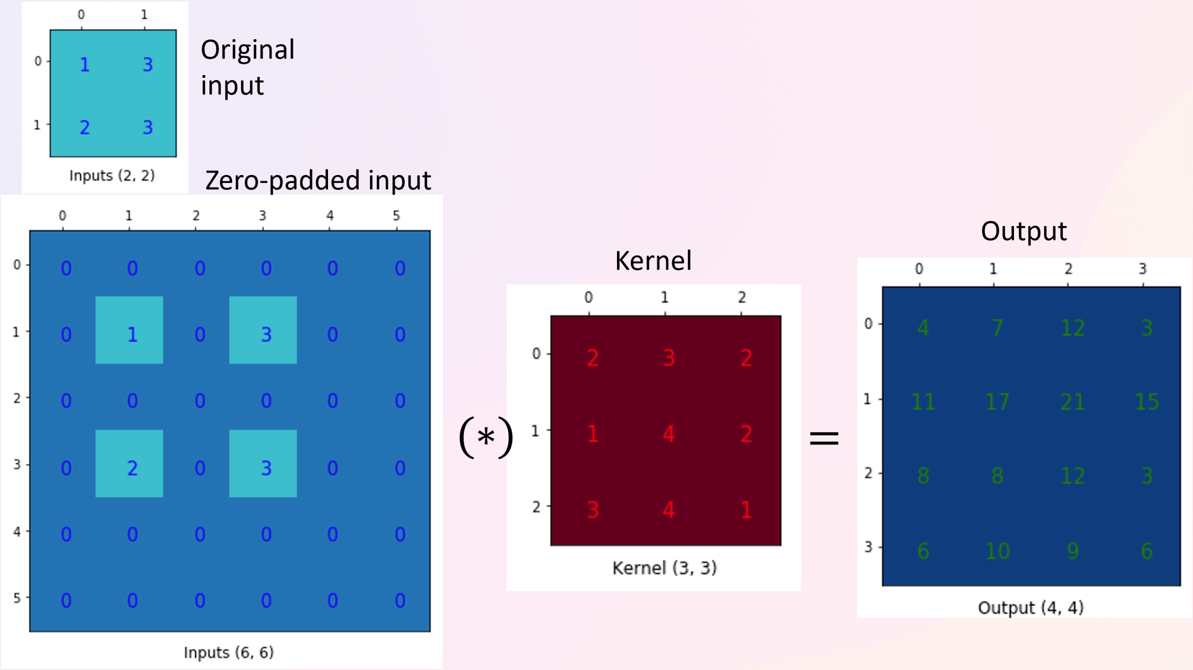
$$
\begin{bmatrix} 2 & 7 & 14 & 12 \\ 6 & 16 & 31 & 21 \\ 6 & 14 & 29 & 21 \\ 4 & 11 & 22 & 12 \end{bmatrix}
$$

Output (4, 4)

https://github.com/naokishibuya/deep-learning/blob/master/python/transposed_convolution.ipynb

Transposed Convolution Can Be Equivalent to a Simple Convolution With Zero Rows/Columns Added

(added zeros simulate fractional strides)



> **Note**
>
> Note: More modern upsampling layers upsample by imputing/interpolating non-zeros and then apply convolution.

Computing Tensor Shapes With Transpose Convolutions

- Channels is computed the same as convolution
- For spatial dimensions, you switch the input and output dimensions
  - Reason about the standard convolution dimensions
  - And then flip input and output dimensions
- Like convolutions, output has **same height and width** as input
  - 1×1 convolution with padding=0, stride=1
  - 3×3 convolution with padding=1, stride=1
  - (Stride of 1 is equivalent to stride of 1 convolution)
- Output has **double (upsample)** the height and width of input
  - 2×2 convolution with padding=0, stride=2
  - 4×4 convolution with padding=1, stride=2

Summary: Convolutional Neural Networks (CNNs)

- **Why CNNs?**
  - **Neuro-inspired**: CNNs learn feature-detecting filters similar to thos in the brain's visual cortex.
  - **Computationally Efficient**: They are efficient due to **sparse computation** (local connections) and **parameter sharing** (the same filter is used across the input).
- **Core Components**
  - **Convolution Layer**: Applies learnable filters (kernels) to input data to create feature maps. The output shape is controlled by `kernel_size`, `stride`, and `padding`.
  - **Activation Function (e.g., ReLU)**: Introduces essential non-linearity, allowing the network to learn complex, non-linear patterns.
  - **Pooling Layer (e.g., Max Pooling)**: Reduces the spatial dimensions (downsamples) of feature maps, which reduces computational load and provides local invariance.
  - **Upsampling with Transposed Convolution**: Used to **increase** the spatial dimensions.