Data 188: Introduction to Deep Learning

Convolutional Networks

Speaker: Eric Kim
Lecture 11 (Week 06)
2026-02-24, Spring 2026. UC Berkeley.

Acknowledgement: slides are based on CMU's Deep Learning Systems course (10-414/714, Fall 2025). Thanks to: Prof. Zico Kolter, Prof. Tianqi Chen, Prof. Tim Dettmers.

Announcements

- HW2 is out!
- Midterm in 2 weeks!
  - (Action Needed) Midterm scheduling form (on Edstem)
    - Please fill this form out!
    - Due: Friday February 27th, 2026, 11:59 PM PST
  - "Advice on how to succeed in this course" (Ed post)

Outline

Convolutional operators in deep networks

Outline

Convolutional operators in deep networks

Images as tensors

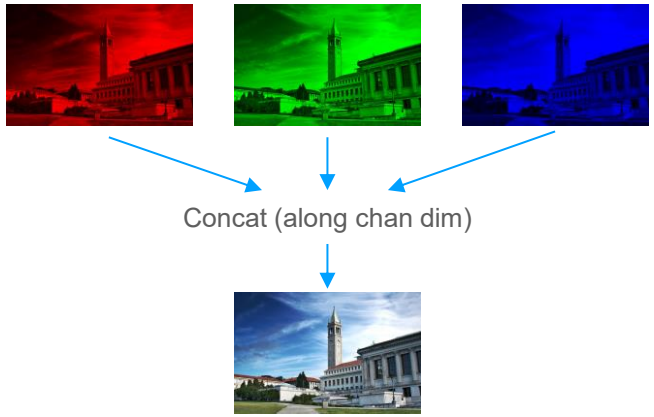
In vision models (eg pytorch), RGB (three channel) images are often represented in uint8 format (ints from [0,255], 0 is black/"off" and 255 is white/"on"), with shape=[3, img_height, img_width], and in R-G-B channel order:

Fun fact: OpenCV (another popular open-source computer vision library) uses B-G-R order. Always good to double check which channel order to use when working on a new project!

```
from PIL import Image
import torchvision
imagepath = "UCBerkeleyCampus-scaled.jpg"
with open(imagepath, "rb") as fh:
    pil_image = Image.open(fh)
    image_tensor = torchvision.transforms.functional.pil_to_tensor(pil_image)
print(f"{type(image_tensor) = }, {image_tensor.dtype = }, {image_tensor.shape = }")
print(f"Red channel: image_tensor[0, :, :]: {image_tensor[0, :, :]}")
print(f"Green channel: image_tensor[1, :, :]: {image_tensor[1, :, :]}")
print(f"Blue channel: image_tensor[2, :, :]: {image_tensor[2, :, :]}")
```

Image has shape [channels=3, height=1665, width=2560]

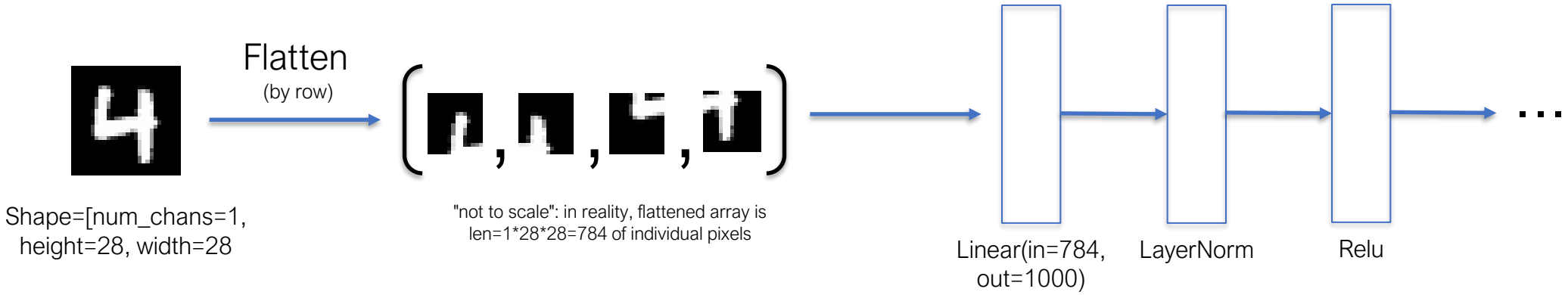
```
type(image_tensor) = <class 'torch.Tensor'>, image_tensor.dtype = torch.uint8, image_tensor.shape = torch.Size([3, 1665, 2560])
Red channel: image_tensor[0, :, :]: tensor([[ 13,  13,  13,  ..., 141, 137, 133],
        [ 13,  13,  13,  ..., 144, 141, 139],
        [ 12,  13,  14,  ..., 144, 144, 142],
        ...,
        [ 68,  72,  87,  ...,  59,  72,  93],
        [ 78,  82,  96,  ...,  72,  73,  72],
        [ 90, 100, 114,  ...,  74,  62,  62]], dtype=torch.uint8)
Green channel: image_tensor[1, :, :]: tensor([[ 39,  39,  39,  ..., 166, 163, 159],
        [ 39,  39,  39,  ..., 169, 167, 165],
        [ 38,  39,  40,  ..., 169, 170, 168],
        ...,
...
```



Learning on image data: low-res, grayscale

So far we have considered networks that treat input images as flattened vectors

Ex: MNIST digit classification:

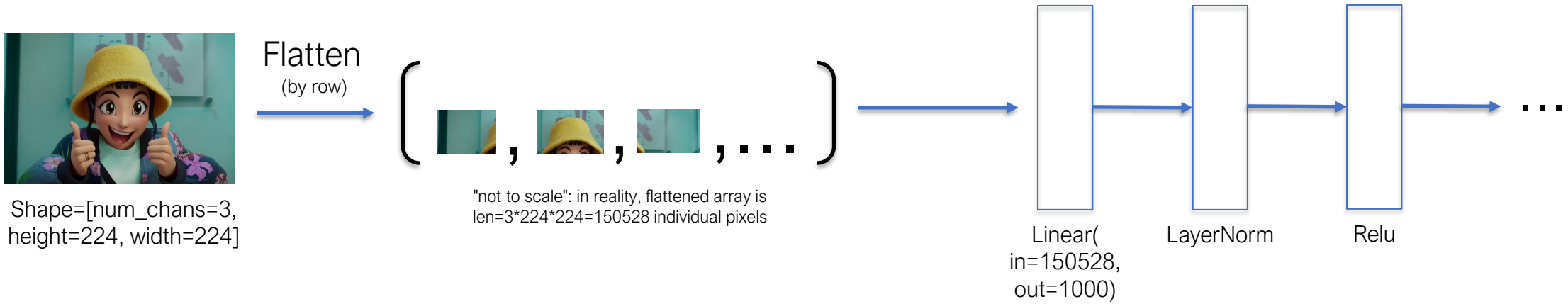


Number of parameters for first Linear layer (with bias): 784*1000 + 1000 = 785000.

Sort of OK for very low resolution, grayscale (single-channel) images

Learning on image data: high-res, color

What about for larger resolution, color (RGB) images?



Num parameters for first Linear layer (with bias): 150528 *1000 + 1000 = 150529000 (~150M!).

This is only for the first layer! Computationally expensive.

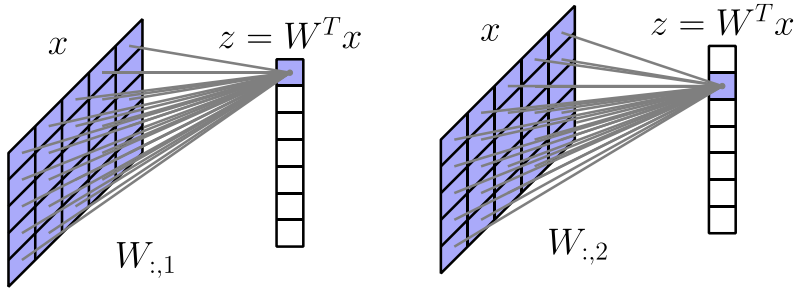
The problem with fully connected networks

From a modeling perspective: Linear layers don't capture any of the “intuitive” invariances that we expect to have in images (e.g., shifting image one pixel leads to very different next layer)

Slight horizontal translation

To us ("humans"), nothing's really changed about the image.

The semantics are still the same (two jellybeans sitting in director chairs)



...but to the Linear layer, the two images look completely different!

Perhaps, linear layers can (with enough width+depth) learn translation invariance. But, what if we could design a layer that already "knew" translation invariance?

Pixar "Dream Productions", 2024

(Detour) Image processing: 2D filters

Many interesting image processing algorithms are implemented by sweeping a 2D filter ("kernel") across an image.

Original image z

Gaussian blur

Image gradient

$$z * \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 4 & 4 & 1 \end{bmatrix} /273$$

$$\left(\left(z * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \right)^2 + \left(z * \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \right)^2 \right)^{\frac{1}{2}}$$

Example: applying a filter to an image (1/3)

Original image I

$$I = \begin{bmatrix} 200 & 200 & 200 & 200 & 200 \\ 200 & 20 & 20 & 45 & 200 \\ 100 & 20 & 25 & 65 & 150 \\ 80 & 20 & 40 & 60 & 80 \\ 80 & 30 & 50 & 80 & 80 \end{bmatrix}$$

Convention: pixel values in range [0,255] ("uint8"), where 0 is black, 255 is white.

$$g = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \cdot \frac{1}{16}$$

A 3x3 kernel.
Defines the weights for a weighted average of neighboring pixel values.

$$\begin{bmatrix} 200 & 200 & 200 \\ 200 & \mathbf{20} & 20 \\ 100 & 20 & 25 \end{bmatrix} \circ \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \cdot \frac{1}{16} = (1*200 + 2*200 + 1*200 + 2*200 + 4*20 + 2*20 + 1*100 + 2*20 + 1*25) / 16 = \mathbf{92.8125}$$

$$I = \begin{bmatrix} 200 & 200 & 200 & 200 & 200 \\ 200 & \mathbf{20} & 20 & 45 & 200 \\ 100 & 20 & 25 & 65 & 150 \\ 80 & 20 & 40 & 60 & 80 \\ 80 & 30 & 50 & 80 & 80 \end{bmatrix}$$

$$X \star g = I_{blur} = \begin{bmatrix} \mathbf{92.8125} & - & - \\ - & - & - \\ - & - & - \end{bmatrix}$$

Example: applying a filter to an image (2/3)

Original image I

$$I = \begin{bmatrix} 200 & 200 & 200 & 200 & 200 \\ 200 & 20 & 20 & 45 & 200 \\ 100 & 20 & 25 & 65 & 150 \\ 80 & 20 & 40 & 60 & 80 \\ 80 & 30 & 50 & 80 & 80 \end{bmatrix}$$

Convention: pixel values in range [0,255] ("uint8"), where 0 is black, 255 is white.

$$g = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \cdot \frac{1}{16}$$

A 3x3 kernel.
Defines the weights for a weighted average of neighboring pixel values.

$$\begin{bmatrix} 200 & 200 & 200 \\ 20 & \mathbf{20} & 45 \\ 20 & 25 & 65 \end{bmatrix} \circ \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \cdot \frac{1}{16} = (1*200 + 2*200 + 1*200 + 2*20 + 4*20 + 2*45 + 1*20 + 2*25 + 1*65) / 16 = \mathbf{71.5625}$$

$$I = \begin{bmatrix} 200 & 200 & 200 & 200 & 200 \\ 200 & 20 & \mathbf{20} & 45 & 200 \\ 100 & 20 & 25 & 65 & 150 \\ 80 & 20 & 40 & 60 & 80 \\ 80 & 30 & 50 & 80 & 80 \end{bmatrix}$$

$$X \star g = I_{blur} = \begin{bmatrix} 92.8125 & \mathbf{71.5625} & - \\ - & - & - \\ - & - & - \end{bmatrix}$$

Example: applying a filter to an image (3/3)

Original image I

$$I = \begin{bmatrix} 200 & 200 & 200 & 200 & 200 \\ 200 & 20 & 20 & 45 & 200 \\ 100 & 20 & 25 & 65 & 150 \\ 80 & 20 & 40 & 60 & 80 \\ 80 & 30 & 50 & 80 & 80 \end{bmatrix}$$

Convention: pixel values in range [0,255] ("uint8"), where 0 is black, 255 is white.

$$g = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \cdot \frac{1}{16}$$

A 3x3 kernel.
Defines the weights for a weighted average of neighboring pixel values.

$$\begin{bmatrix} 200 & 20 & 20 \\ 100 & \mathbf{20} & 25 \\ 80 & 20 & 40 \end{bmatrix} \circ \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \cdot \frac{1}{16} = (1*200 + 2*20 + 1*20 + 2*100 + 4*20 + 2*25 + 1*80 + 2*20 + 1*40) / 16 = \mathbf{46.875}$$

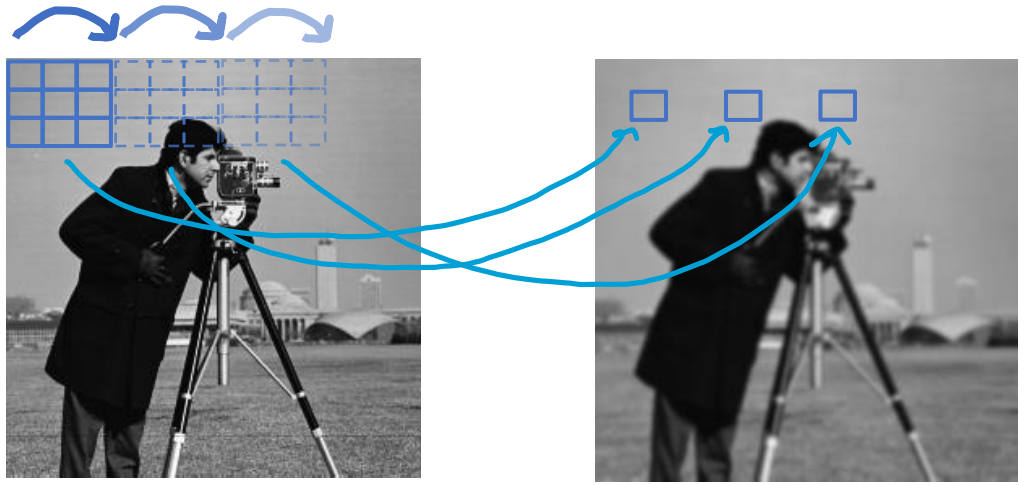
$$I = \begin{bmatrix} 200 & 200 & 200 & 200 & 200 \\ 200 & 20 & 20 & 45 & 200 \\ 100 & \mathbf{20} & 25 & 65 & 150 \\ 80 & 20 & 40 & 60 & 80 \\ 80 & 30 & 50 & 80 & 80 \end{bmatrix}$$

$$X \star g = I_{blur} = \begin{bmatrix} 92.8125 & 71.5625 & - \\ \mathbf{46.875} & - & - \\ - & - & - \end{bmatrix}$$

Pseudocode: applying a filter

```
Pseudocode:
# suppose kernel g has size [k, k]. Assume grayscale image.
delt = floor((k - 1) / 2)
filter_response = blank_image()
for each (x, y) in img:
    # retrieve [k, k] region centered at (x, y)
    cur_img_region = img[y-delt:(y+delt+1), x-delt:(x+delt+1)]
    local_resp = sum(cur_img_region * g)  # elemwise product
    filter_response[y, x] = local_resp
```

Aka "sweeping" a filter g across the image



Original image I

Gaussian blur

$$I \star g = I_{blur} = \begin{bmatrix} 92.8125 & 71.5625 & - \\ \mathbf{46.875} & - & - \\ - & - & - \end{bmatrix}$$

$$I = \begin{bmatrix} 200 & 200 & 200 & 200 & 200 \\ 200 & 20 & 20 & 45 & 200 \\ 100 & \mathbf{20} & 25 & 65 & 150 \\ 80 & 20 & 40 & 60 & 80 \\ 80 & 30 & 50 & 80 & 80 \end{bmatrix} \qquad g = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \cdot \frac{1}{16}$$

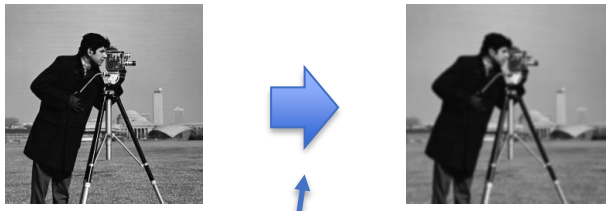
2D filters: can they do more?

Many low-level image processing algorithms are implemented by sweeping ("convolving") a 2D filter ("kernel") across an image.

Examples: image blurring, edge detection.

Idea 1: what if 2D filters could do more? Classify images? Detect visual objects?

Idea 2: blur/edge-detection kernels are hardcoded. What if we could **learn** our filters?



Input image z

Gaussian blur

$$z * \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 4 & 4 & 1 \end{bmatrix} /273$$

Filter that performs gaussian blur (hardcoded, specialized)

Input image z

...

Learnable filters?

$$z * \begin{bmatrix} g_{00} & g_{01} & g_{02} & \cdots \\ g_{10} & g_{11} & g_{12} & \cdots \\ g_{20} & g_{21} & g_{22} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

"Bucket hat detector" filter

$$z * \begin{bmatrix} g_{00} & g_{01} & g_{02} & \cdots \\ g_{10} & g_{11} & g_{12} & \cdots \\ g_{20} & g_{21} & g_{22} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

"Person detector" filter

"Conv2dSingle": single filter, single channel

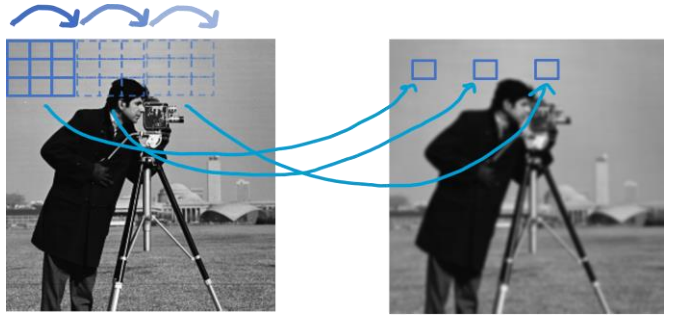
Let's define a convolution layer "**Conv2dSingle**" that applies a **single** learned 2D filter to a 2D input spatial feature map, producing a 2D output spatial feature map.

This is `1` because our image is single channel (grayscale)

Input: $[\text{batchsize}, 1, \text{height}_{\text{in}}, \text{width}_{\text{in}}]$

Shorthand: $[\text{B}, \text{C}_{\text{in}}, \text{H}_{\text{in}}, W_{\text{in}}]$

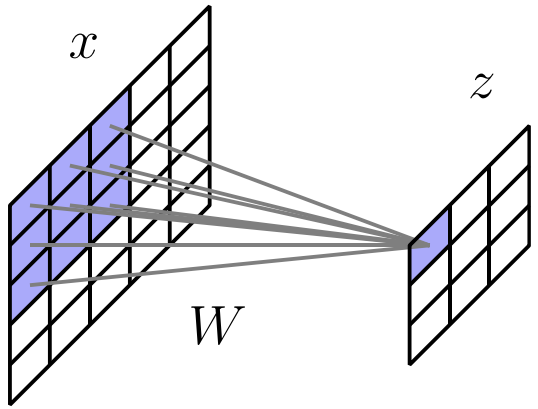
Output: $[\text{batchsize}, \text{height}_{\text{out}}, \text{width}_{\text{out}}]$



Input spatial feature map

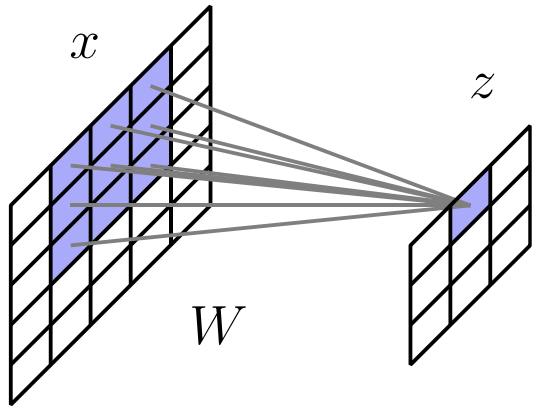
Output spatial feature map

(in blue) the filter being applied. Here, the filter shape is [3, 3].



Input image x

Output spatial feature map



"Sweep" the filter across the input image x

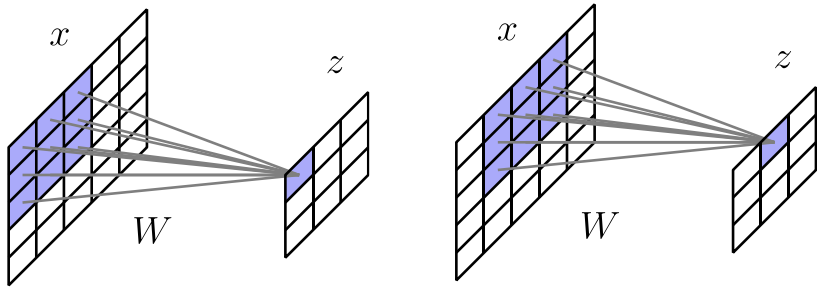
"Conv2dSingle": details

This is `1` because our image is single channel (grayscale)

Input: $[\text{batchsize}, 1, \text{height}_{\text{in}}, \text{width}_{\text{in}}]$

Shorthand: $[\text{B}, \text{C}_{\text{in}}, \text{H}_{\text{in}}, W_{\text{in}}]$

Output: $[\text{batchsize}, \text{height}_{\text{out}}, \text{width}_{\text{out}}]$



Layer trainable parameters:

This is `1` because our image is single channel (grayscale)

Filter weights: $[1, \text{k}, \text{k}]$

Bias parameters: $[1]$

k is kernel size. Determines spatial extent of filter.

Bias is a learned offset applied to the filter outputs, eg broadcasted add.

Forward pass calculates: $conv2dSingle(Z_{in}, g) = Z_{out} + bias$

Where: $Z_{out} = Z_{in} \star g$

Aka: first sweep the (learned) filter g to our input spatial feature map Z_{in}, then add a (learned) offset.

Note: in theory we can have non-square filters, eg $k_h = 3, k_w = 4$ for a filter with height=3, width=4. But, in practice, this is rarely done. Finally: in practice, k is often an odd integer.

"Conv2dSingle": forward pass (0/6)

Convolutions are a basic primitive in many computer vision and image processing algorithms

Idea is to “slide” the weights $k \times k$ weight w (called a filter, with kernel size k) over the image to produce a new image, written $y = z * w$

$$\begin{bmatrix} z_{11} & z_{12} & z_{13} & z_{14} & z_{15} \\ z_{21} & z_{22} & z_{23} & z_{24} & z_{25} \\ z_{31} & z_{32} & z_{33} & z_{34} & z_{35} \\ z_{41} & z_{42} & z_{43} & z_{44} & z_{45} \\ z_{51} & z_{52} & z_{53} & z_{54} & z_{55} \end{bmatrix} * \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} = \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ y_{31} & y_{32} & y_{33} \end{bmatrix}$$

"Conv2dSingle": forward pass (1/6)

Convolutions are a basic primitive in many computer vision and image processing algorithms

Idea is to "slide" the weights $k \times k$ weight w (called a filter, with kernel size k) over the image to produce a new image, written $y = z * w$

z_{11}	z_{12}	z_{13}	z_{14}	z_{15}
z_{21}	z_{22}	z_{23}	z_{24}	z_{25}
z_{31}	z_{32}	z_{33}	z_{34}	z_{35}
z_{41}	z_{42}	z_{43}	z_{44}	z_{45}
z_{51}	z_{52}	z_{53}	z_{54}	z_{55}

$*$

w_{11}	w_{12}	w_{13}
w_{21}	w_{22}	w_{23}
w_{31}	w_{32}	w_{33}

$=$

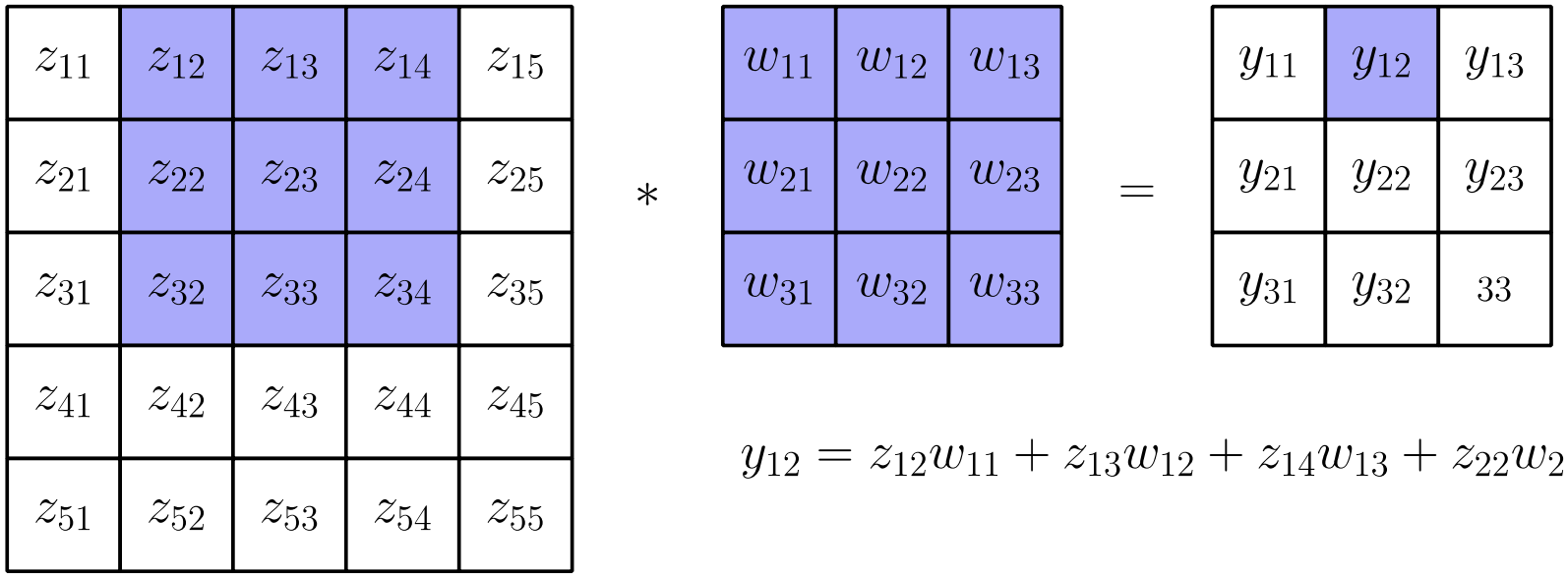
y_{11}	y_{12}	y_{13}
y_{21}	y_{22}	y_{23}
y_{31}	y_{32}	33

$$y_{11} = z_{11}w_{11} + z_{12}w_{12} + z_{13}w_{13} + z_{21}w_{21} + \ldots$$

"Conv2dSingle": forward pass (2/6)

Convolutions are a basic primitive in many computer vision and image processing algorithms

Idea is to "slide" the weights $k \times k$ weight w (called a filter, with kernel size k) over the image to produce a new image, written $y = z * w$



$$y_{12} = z_{12}w_{11} + z_{13}w_{12} + z_{14}w_{13} + z_{22}w_{21} + \ldots$$

"Conv2dSingle": forward pass (3/6)

Convolutions are a basic primitive in many computer vision and image processing algorithms

Idea is to "slide" the weights $k \times k$ weight w (called a filter, with kernel size k) over the image to produce a new image, written $y = z * w$

z_{11}	z_{12}	z_{13}	z_{14}	z_{15}
z_{21}	z_{22}	z_{23}	z_{24}	z_{25}
z_{31}	z_{32}	z_{33}	z_{34}	z_{35}
z_{41}	z_{42}	z_{43}	z_{44}	z_{45}
z_{51}	z_{52}	z_{53}	z_{54}	z_{55}

$*$

w_{11}	w_{12}	w_{13}
w_{21}	w_{22}	w_{23}
w_{31}	w_{32}	w_{33}

$=$

y_{11}	y_{12}	y_{13}
y_{21}	y_{22}	y_{23}
y_{31}	y_{32}	33

$$y_{13} = z_{13}w_{11} + z_{14}w_{12} + z_{15}w_{13} + z_{23}w_{21} + \ldots$$

"Conv2dSingle": forward pass (4/6)

Convolutions are a basic primitive in many computer vision and image processing algorithms

Idea is to “slide” the weights $k \times k$ weight w (called a filter, with kernel size k) over the image to produce a new image, written $y = z * w$

z_{11}	z_{12}	z_{13}	z_{14}	z_{15}
z_{21}	z_{22}	z_{23}	z_{24}	z_{25}
z_{31}	z_{32}	z_{33}	z_{34}	z_{35}
z_{41}	z_{42}	z_{43}	z_{44}	z_{45}
z_{51}	z_{52}	z_{53}	z_{54}	z_{55}

$*$

w_{11}	w_{12}	w_{13}
w_{21}	w_{22}	w_{23}
w_{31}	w_{32}	w_{33}

$=$

y_{11}	y_{12}	y_{13}
y_{21}	y_{22}	y_{23}
y_{31}	y_{32}	33

$$y_{21} = z_{21}w_{11} + z_{22}w_{12} + z_{23}w_{13} + z_{31}w_{21} + \ldots$$

"Conv2dSingle": forward pass (5/6)

Convolutions are a basic primitive in many computer vision and image processing algorithms

Idea is to "slide" the weights $k \times k$ weight w (called a filter, with kernel size k) over the image to produce a new image, written $y = z * w$

z_{11}	z_{12}	z_{13}	z_{14}	z_{15}
z_{21}	z_{22}	z_{23}	z_{24}	z_{25}
z_{31}	z_{32}	z_{33}	z_{34}	z_{35}
z_{41}	z_{42}	z_{43}	z_{44}	z_{45}
z_{51}	z_{52}	z_{53}	z_{54}	z_{55}

$*$

w_{11}	w_{12}	w_{13}
w_{21}	w_{22}	w_{23}
w_{31}	w_{32}	w_{33}

$=$

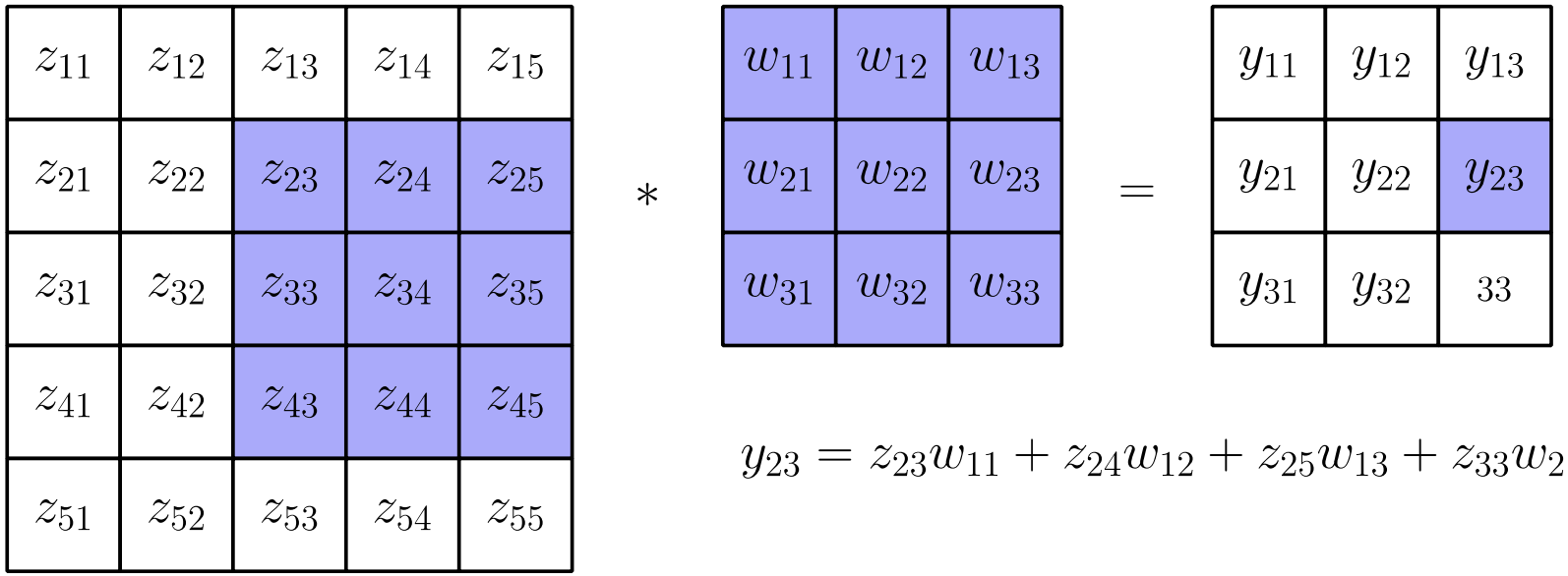
y_{11}	y_{12}	y_{13}
y_{21}	y_{22}	y_{23}
y_{31}	y_{32}	33

$$y_{22} = z_{22}w_{11} + z_{23}w_{12} + z_{24}w_{13} + z_{32}w_{21} + \ldots$$

"Conv2dSingle": forward pass (6/6)

Convolutions are a basic primitive in many computer vision and image processing algorithms

Idea is to "slide" the weights $k \times k$ weight w (called a filter, with kernel size k) over the image to produce a new image, written $y = z * w$



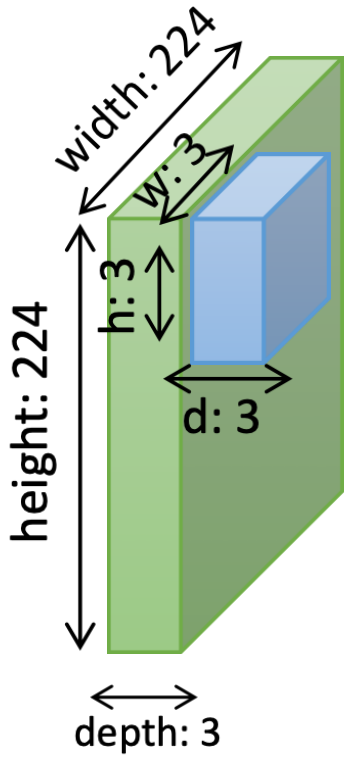
$$y_{23} = z_{23}w_{11} + z_{24}w_{12} + z_{25}w_{13} + z_{33}w_{21} + \ldots$$

Convolution: multiple input channels

Let's generalize Conv2dSingle to support **multichannel** input (eg RGB images) and **multiple filters**.

To support multichannel input with shape=$[B, C_{in}, h_{in}, w_{in}]$:

=> filters now have shape=$[C_{in}, k, k]$



Pictured: input is 224x224 three-channel (RGB) image with shape=$[1, C_{in} = 3, 224, w = 224]$.
Filter shape=$[C_{in} = 3, k = 3, 3]$

Conv2d: multichannel computation (1/4)

Convolution (aka "cross correlation"): **sliding window** computation

(Red+Green channels)

$$\begin{bmatrix} 1 & 128 & 100 & 255 \\ 4 & 0 & 100 & 10 & 80 & 122 & 0 & 215 \\ 20 & 10 & 90 & 255 & 70 & 50 & 0 & 2 \\ 40 & 0 & 50 & 90 & 60 & 70 & 2 & 3 \\ & 95 & 120 & 98 & 4 \end{bmatrix}$$

Zero Pad=1

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 128 & 0 & 100 & 0 & 255 \\ 0 & 0 & 4 & 0 & 10 & 0 & 10 & 80 & 122 & 0 & 215 \\ 0 & 0 & 20 & 10 & 90 & 255 & 70 & 50 & 0 & 2 \\ & 0 & 40 & 0 & 50 & 90 & 60 & 70 & 2 & 3 \\ & & & 95 & 120 & 98 & 4 \end{bmatrix}$$

```
>>> filters[0, :, :, :]
```

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 & 0.5 & 1 \\ 1 & 1.5 \end{bmatrix}$$

```
>>> filters[1, :, :, :]
```

$$\begin{bmatrix} 0 & 0.1 \\ 1 & 0 & 2 & 0.1 \\ 1 & 2 \end{bmatrix}$$

starting from the Red channel first

output[0, 0, 0] = 1 * 0 + 2 * 0 + 0 * 0 + 0.5 * 1 + 1 * 0 + 1 * 0 + 1 * 0 + 1.5 * 0 = 0.5

`output[0, :, :]:`

$$\begin{bmatrix} 0.5 \\ \\ \\ \end{bmatrix}$$

Conv2d: multichannel computation (2/4)

Convolution (aka "cross correlation"): **sliding window** computation

(Red+Green channels)

$$\begin{bmatrix} 1 & 128 & 100 & 255 \\ 4 & 0 & 100 & 10 & 80 & 122 & 0 & 215 \\ 20 & 10 & 90 & 255 & 70 & 50 & 0 & 2 \\ 40 & 0 & 50 & 90 & 60 & 70 & 2 & 3 \\ & 95 & 120 & 98 & 4 \end{bmatrix}$$

Zero Pad=1

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 128 & 0 & 100 & 0 & 255 \\ 0 & 0 & 4 & 0 & 100 & 10 & 80 & 122 & 0 & 215 \\ 0 & 0 & 20 & 10 & 90 & 255 & 70 & 50 & 0 & 2 \\ & 0 & 40 & 0 & 50 & 90 & 60 & 70 & 2 & 3 \\ & & 95 & 120 & 98 & 4 \end{bmatrix}$$

```
>>> filters[0, :, :, :]
```

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 & 0.5 & 1 \\ 1 & 1.5 \end{bmatrix}$$

```
>>> filters[1, :, :, :]
```

$$\begin{bmatrix} 0 & 0.1 \\ 1 & 0 & 2 & 0.1 \\ 1 & 2 \end{bmatrix}$$

starting from the Red channel first

output[0, 0, 1] = 1 * 0 + 2 * 0 + 0 * 1 + 0.5 * 128 + 1 * 0 + 1 * 0 + 1 * 0 + 1.5 * 10 = 79

`output[0, :, :]:`

$$\begin{bmatrix} 0.5 & 79 \\ & \\ & \end{bmatrix}$$

Conv2d: multichannel computation (3/4)

Convolution (aka "cross correlation"): **sliding window** computation

(Red+Green channels)

$$\begin{bmatrix} 1 & 128 & 100 & 255 \\ 4 & 0 & 100 & 10 & 80 & 122 & 0 & 215 \\ 20 & 10 & 90 & 255 & 70 & 50 & 0 & 2 \\ 40 & 0 & 50 & 90 & 60 & 70 & 2 & 3 \\ & 95 & 120 & 98 & 4 \end{bmatrix}$$

Zero Pad=1

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 128 & 0 & 100 & 0 & 255 \\ 0 & 0 & 4 & 0 & 100 & 10 & 80 & 122 & 0 & 215 \\ 0 & 0 & 20 & 10 & 90 & 255 & 70 & 50 & 0 & 2 \\ & 0 & 40 & 0 & 50 & 90 & 60 & 70 & 2 & 3 \\ & & & 95 & 120 & 98 & 4 \end{bmatrix}$$

```
>>> filters[0, :, :, :]
```

$$\begin{bmatrix} 1 & 2 \\ 0 & \begin{bmatrix} 1 & 0.5 \\ 1 & 1.5 \end{bmatrix} & 1 \end{bmatrix}$$

```
>>> filters[1, :, :, :]
```

$$\begin{bmatrix} 0 & 0.1 \\ 1 & \begin{bmatrix} 0 & 0.1 \\ 1 & 2 \end{bmatrix} & 2 \end{bmatrix}$$

starting from the Red channel first

output[0, 1, 0] = 1 * 0 + 2 * 1 + 0 * 0 + 0.5 * 4 + 1 * 0 + 1 * 0 + 1 * 0 + 1.5 * 10 = 19

`output[0, :, :]:`

$$\begin{bmatrix} 0.5 & 79 \\ 19 & \\ & \\ & \end{bmatrix}$$

Conv2d: multichannel computation (4/4)

Convolution (aka "cross correlation"): **sliding window** computation

(Red+Green channels)

$$\begin{bmatrix} 1 & 128 & 100 & 255 \\ 4 & 0 & 100 & 10 & 80 & 122 & 0 & 215 \\ 20 & 10 & 90 & 255 & 70 & 50 & 0 & 2 \\ 40 & 0 & 50 & 90 & 60 & 70 & 2 & 3 \\ & 95 & 120 & 98 & 4 \end{bmatrix}$$

Zero Pad=1

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 128 & 0 & 100 & 0 & 255 \\ 0 & 0 & 4 & 0 & 100 & 10 & 80 & 122 & 0 & 215 \\ 0 & 0 & 20 & 10 & 90 & 255 & 70 & 50 & 0 & 2 \\ & 0 & 40 & 0 & 50 & 90 & 60 & 70 & 2 & 3 \\ & & & 95 & 120 & 98 & 4 \end{bmatrix}$$

```
>>> filters[0, :, :, :]
```

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 & 0.5 & 1 \\ 1 & 1.5 \end{bmatrix}$$

```
>>> filters[1, :, :, :]
```

$$\begin{bmatrix} 0 & 0.1 \\ 1 & 0 & 2 & 0.1 \\ 1 & 2 \end{bmatrix}$$

starting from the Red channel first

output[0, 1, 0] = 1 * 1 + 2 * 128 + 0 * 4 + 0.5 * 100 + 1 * 0 + 1 * 10 + 1 * 10 + 1.5 * 255 = 709.5

`output[0, :, :]:`

$$\begin{bmatrix} 0.5 & 79 \\ 19 & 709.5 \end{bmatrix}$$

Convolution layer: multiple filters

Let's generalize Conv2dSingle to support **multichannel** input (eg RGB images) and **multiple filters**.

To support multiple independent filters:

⇒ have each filter write to a different output channel.

Our output shape is now $[B, C_{out}, h_{out}, w_{out}]$, where C_{out} is our number of filters.

$$X = \begin{bmatrix} 200 & 200 & 200 & 200 & 200 \\ 200 & \mathbf{20} & 20 & 45 & 200 \\ 100 & 20 & 25 & 65 & 150 \\ 80 & 20 & 40 & 60 & 80 \\ 80 & 30 & 50 & 80 & 80 \end{bmatrix}$$

$$Z_{out}[b, \mathbf{0}, :, :] = X \star \boldsymbol{G}[\mathbf{0}, :, :, :] = \begin{bmatrix} \mathbf{92.8125} & - & - \\ - & - & - \\ - & - & - \end{bmatrix}$$

Filter response (spatial feature map) for filter 0

$$X = \begin{bmatrix} 200 & 200 & 200 & 200 & 200 \\ 200 & \mathbf{20} & 20 & 45 & 200 \\ 100 & 20 & 25 & 65 & 150 \\ 80 & 20 & 40 & 60 & 80 \\ 80 & 30 & 50 & 80 & 80 \end{bmatrix}$$

$$Z_{out}[b, \mathbf{1}, :, :] = X \star \boldsymbol{G}[\mathbf{1}, :, :, :] = \begin{bmatrix} \mathbf{42.91} & - & - \\ - & - & - \\ - & - & - \end{bmatrix}$$

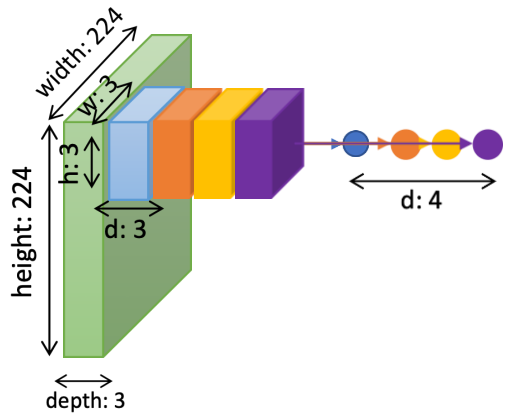
Filter response (spatial feature map) for filter 1

...

G is our "filter bank" with shape=$[C_{out}, C_{in}, k, k]$, where C_{out} is the number of filters owned by this layer.

Conv2d layer

A Conv2d layer applies a **series of learned 2D filters** to a multichannel 2D spatial feature map, producing a multichannel 2D spatial feature map.



Input: $[\text{batchsize}, C_{\text{in}}, \text{h}_{\text{in}}, \text{w}_{\text{in}}]$

Output: $[\text{batchsize}, C_{\text{out}}, h_{\text{out}}, w_{\text{out}}]$

Layer trainable parameters:

Filter weights ("filter bank"): $[\text{C}_{\text{out}}, \text{C}_{\text{in}}, \text{k}, \text{k}]$

Bias parameters: $[\text{C}_{\text{out}}]$

Number of filters is C_{out}

k is kernel size, the spatial extent of filter.

Bias is a learned offset applied to the filter outputs, eg broadcasted add.

Forward pass calculates: $conv2d(Z_{in}, G) = Z_{out} + bias$

Where: $Z_{out}[i,:,:] = Z_{in} \star G[i,:,:,:]$

Z_{in} has shape=$[B, C_{in}, h_{in}, w_{in}]$
G is filter weights, shape= $[\text{C}_{\text{out}}, \text{C}_{\text{in}}, \text{k}, \text{k}]$
Z_{out} has shape=$[B, C_{out}, h_{out}, w_{out}]$
bias has shape=$[C_{out}]$

Note: here, I'm defining a "3d conv" to be "sum the elemwise prod of Z_{in} and $G[i,:,:]$, for each spatial location in Z_{in}"

Sweep (convolve) each filter $G[i,:,:]$ to our input spatial feature map Z_{in}, and store the filter response to $Z_{out}[i,:,:]$

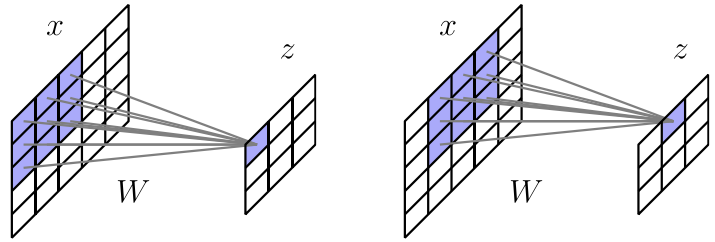
bias broadcast: each output channel in C_{out} has a separate scalar offset applied to the entire spatial window.

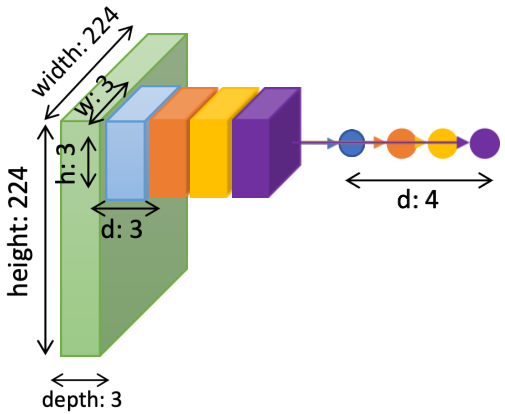
Conv2d: additional parameters

torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')

So far, we've only considered stride=1, padding=0 (and ignored "dilation", "groups").

We have additional control over how the convolution is done, via these parameters: **stride, padding, dilation, groups**.





Pictured: input has shape=$[1, 3, 224, 224]$.
Filter weights ("bank") shape=$[C_{out} = 4, C_{in} = 3, 3, 3]$
Stride=1, padding=0.

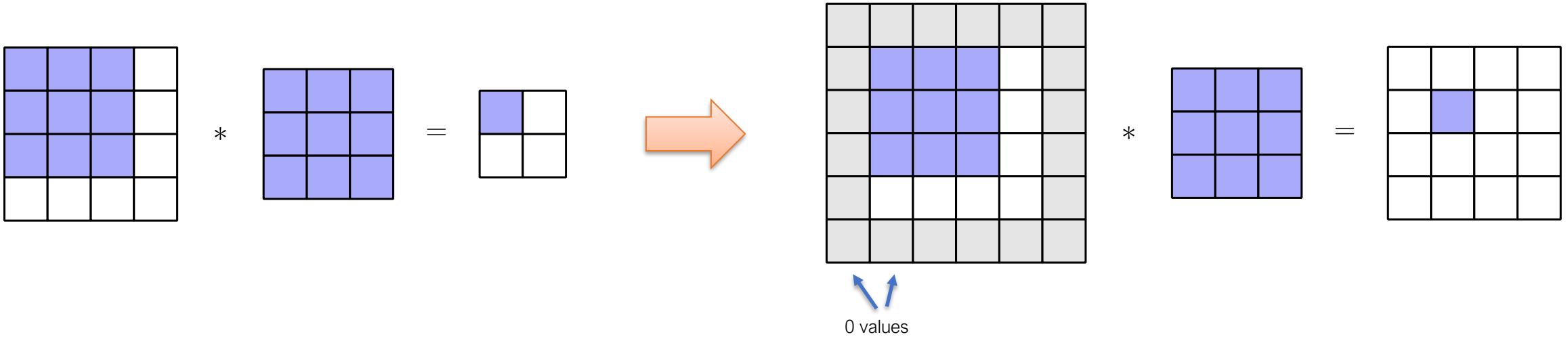
https://docs.pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

Padding

Challenge: "Naïve" convolutions produce a smaller output than input image

Solution: for (odd) kernel size k, pad input with $(k-1)/2$ zeros on all sides, results in an output that is the same size as the input

- Variants like circular padding, padding with mean values, etc

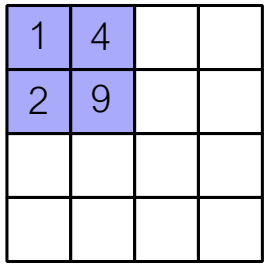


Strided Convolutions / Pooling

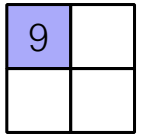
Challenge: Convolutions keep the same resolution of the input at each layer, don't naively allow for representations at different "resolutions"

Solution #1: incorporate max or average *pooling* layers to **aggregate** information

Solution #2: slide convolutional filter over image in increments >1 (= stride)



max ⟹

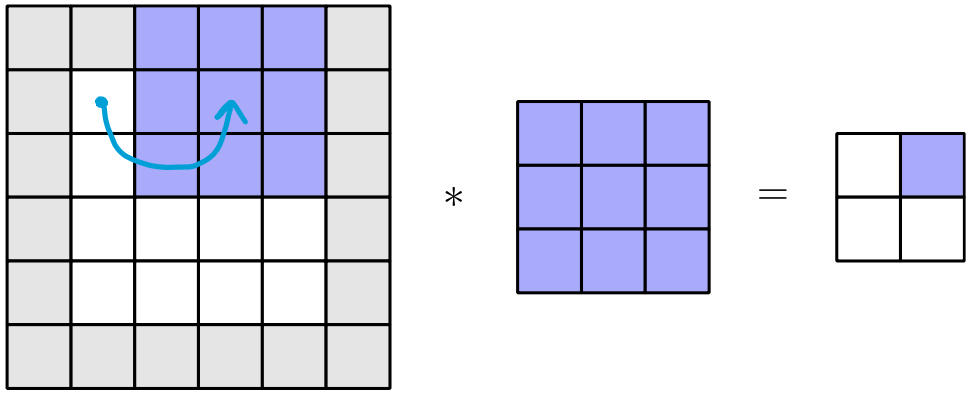


Pooling

MaxPool(k=2, stride=2,padding=0): apply max over each non-overlapping 2x2 window

AvgPool: similar, but calculate average value for each window

stride=2, eg skip a cell after each step

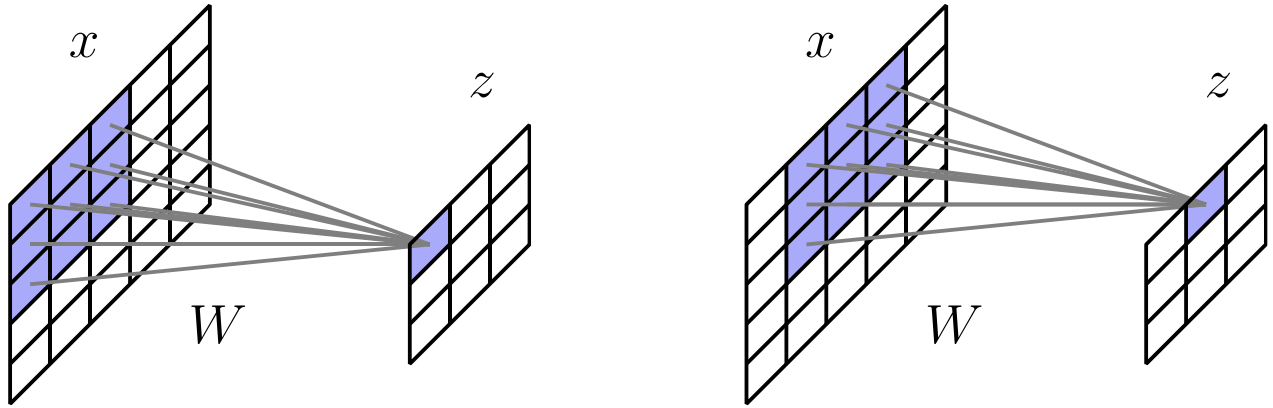


Strided convolution

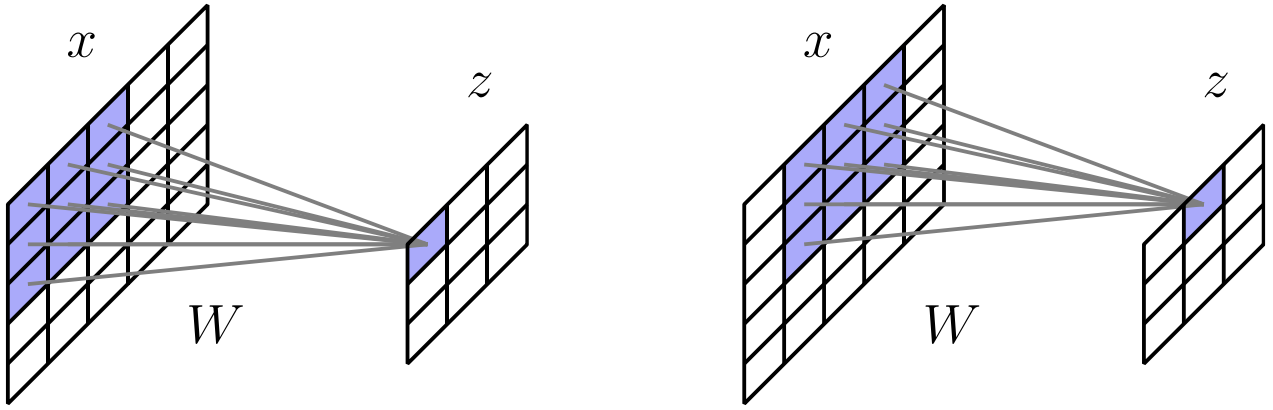
How convolutions “simplify” deep networks

Convolutions combine two ideas that are well-suited to processing images

1. Require that activations between layers occur only in a “local” manner, and treat hidden layers themselves as spatial images
2. Share weights across all spatial locations



Advantages of convolutions



Drastically reduces the parameter count

- 256x256 grayscale image ⟹ 256x256 single-channel hidden layer: 4 *billion parameters* in fully connected network to *9 parameters* in 3x3 convolution

Captures (some) “natural” invariances

- Shifting input image one pixel to the right shifts creates a hidden shifts the hidden unit “image”

Convolutions in image processing

Convolutions (typically with *prespecified* filters) are a common operation in many computer vision applications: convolution networks just move to *learned* filters

Original image z

Gaussian blur

$$z * \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 4 & 4 & 1 \end{bmatrix} /273$$

Image gradient

$$\left(\left(z * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \right)^2 + \left(z * \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \right)^2 \right)^{\frac{1}{2}}$$

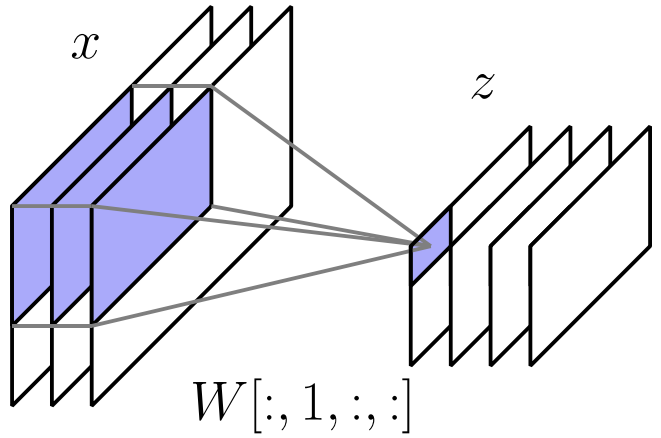
Convolutions in deep networks: summary

Convolutions in deep networks are virtually always *multi-channel* convolutions: map multi-channel (e.g., RGB) inputs to multi-channel outputs (hidden units)

- $x \in \mathbb{R}^{c_{in} \times h_{in} \times w_{in}}$ denotes c_{in} channel, size $h_{in} \times w_{in}$ image input
- $z \in \mathbb{R}^{c_{out} \times h_{out} \times w_{out}}$ denotes c_{out} channel, size $h_{out} \times w_{out}$ output
- $W \in \mathbb{R}^{c_{out} \times c_{in} \times k \times k}$ (order 4 tensor) denotes convolutional filters ("filter bank")

Multi-channel convolutions contain a convolutional filter for *each input-output channel pair*, single output channel is sum of convolutions over all input channels

$$z[:,:,s] = \sum_{r=1}^{c_{in}} x[:,:,r] * W[r,s,:,:]$$

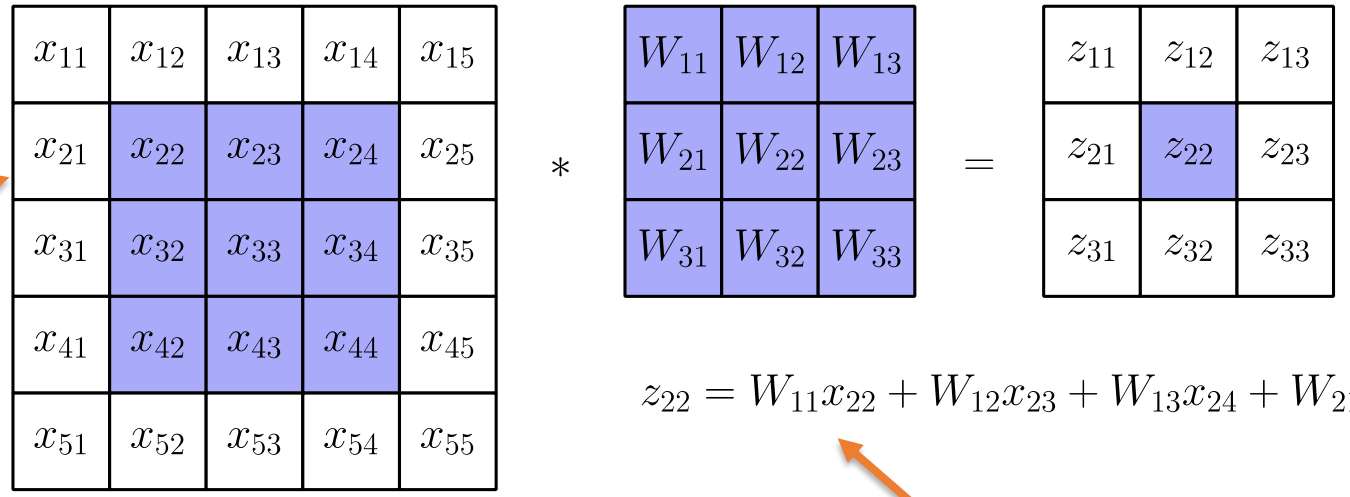


Multi-channel convolutions in matrix-vector form

Alternate (possibly more intuitive) way to think about multi-channel convolutions: they are a generalization of traditional convolutions with scalar multiplications replaced by matrix-vector products

These are each $\mathbb{R}^{c_{out} \times c_{in}}$ matrices

These are each vectors in $\mathbb{R}^{c_{in}}$



$$z_{22} = W_{11}x_{22} + W_{12}x_{23} + W_{13}x_{24} + W_{21}x_{32} + \dots$$

These are matrix-vector products