

Demystifying the Convolutions in PyTorch

Avinash Kak
Purdue University

Lecture Notes on Deep Learning by Avi Kak and Charles Bouman

Sunday 8th February, 2026 17:15

©2026 A. C. Kak, Purdue University

On the face of it, the convolution is one of the simplest ideas in computer vision and image processing.

All that's meant by a convolution is that you sweep an image with a flipped kernel (which is assumed to be smaller in size compared to the image), you sum the product of the two at each position of the kernel, and report the value calculated to the output.

But, as with so many things in life, this simplicity can be deceptive — especially so in the context of deep learning.

Hidden behind the simplicity is the fact that calculating a convolution calls for making assumptions about what to do at the border of the input. While the consequences of those assumptions can be ignored in computer vision and image processing, **that's not so easily done in DL where the resolution hierarchies can be deep and, at the top of a resolution pyramid, each pixel may represent a significant chunk of the image at the bottom.**

Preamble (contd.)

Other issues regarding convolutions in DL relate to the role played by the channels. How do M channels in the input go into N channels at the output for literally arbitrary values for M and N ?

And what about the “groups” option when you call PyTorch’s functions for convolutions? What does that do?

Finally, what about the fact that DL convolutions are really not convolutions, but cross-correlations?

Here is a “NOTE” on the doc page for `torch.nn.Conv2d`:

“Depending on the size of your kernel, several (of the last) columns of your input might be lost because it is a valid cross-correlation and not a full-correlation. It is up to the user to add proper padding.”

What exactly does that mean — considering especially the fact that padding involves making assumptions about imagined pixels outside the image array?

Outline

- 1 2D Convolution — The Basic Definition 5
- 2 What About `scipy.signal.convolve2d()` for 2D Convolutions 9
- 3 Input and Kernel Specs for PyTorch's Convolution Function
`torch.nn.functional.conv2d()` 12
- 4 Squeezing and Unsqueezing the Tensors 18
- 5 Using `torch.nn.functional.conv2d()` 26
- 6 2D Convolutions with the PyTorch Class `torch.nn.Conv2d` 28
- 7 Verifying That a PyTorch Convolution is in Reality a
Cross-Correlation 36
- 8 Multi-Channel Convolutions 40
- 9 Reshaping a Tensor with `reshape()` and `view()` 52

Outline

- 1 **2D Convolution — The Basic Definition** 5
- 2 What About `scipy.signal.convolve2d()` for 2D Convolutions 9
- 3 Input and Kernel Specs for PyTorch's Convolution Function
`torch.nn.functional.conv2d()` 12
- 4 Squeezing and Unsqueezing the Tensors 18
- 5 Using `torch.nn.functional.conv2d()` 26
- 6 2D Convolutions with the PyTorch Class `torch.nn.Conv2d` 28
- 7 Verifying That a PyTorch Convolution is in Reality a
Cross-Correlation 36
- 8 Multi-Channel Convolutions 40
- 9 Reshaping a Tensor with `reshape()` and `view()` 52

2D Convolution

- The following snippet of Python code nicely says it all as far as the definition of 2D convolution is concerned:

```
def convo2d(input, kernel):
    H,W = input.shape
    M,N = kernel.shape
    out = numpy.zeros((H-M+1,W-N+1), dtype=float)
    kernel = numpy.flip(kernel)
    for i in range(H-M+1):
        for j in range(W-N+1):
            out[i,j] = numpy.sum( input[i:i+M,j:j+N] * kernel)
    return out
```

- If you are a beginner Python programmer, pay attention to the role of `numpy.flip()` in the script.,

[NOTE: Note my use of the “mnemonics” for the variables **H** for the “height” and **W** for the “width” of the input pattern. This is to help cope with possible mental confusion when you are also using the PIL library in the same program. The image related functions in that library are based on the notion of **(x,y)** coordinates, with **x** standing for the horizontal axis and **y** for the vertical axis.]

2D Convolution

- Let's now define an **input** array for the convolutions:

```
arr = numpy.zeros((8, 8), dtype=float)
arr[:, :4] = 4.0
arr[:, 4:] = 1.0

print(arr)
#           [[4. 4. 4. 4. 1. 1. 1. 1.]
#           [4. 4. 4. 4. 1. 1. 1. 1.]
#           [4. 4. 4. 4. 1. 1. 1. 1.]
#           [4. 4. 4. 4. 1. 1. 1. 1.]
#           [4. 4. 4. 4. 1. 1. 1. 1.]
#           [4. 4. 4. 4. 1. 1. 1. 1.]
#           [4. 4. 4. 4. 1. 1. 1. 1.]
#           [4. 4. 4. 4. 1. 1. 1. 1.]]

print( type(arr) )           ## <class 'numpy.ndarray'>
print( type(arr.dtype) )    ## <class 'numpy.dtypes.Float64DType'>
print( type(arr[0, 0]) )    ## <class 'numpy.float64'> NOTE: numpy default for floats: 64 bits
```

- Next we need to define a **kernel**:

```
ker = numpy.zeros((3, 3), dtype=float)

ker[:, 0] = -1.0
ker[:, 2] = 1.0

print(ker)
#           [[-1.  0.  1.]
#           [-1.  0.  1.]
#           [-1.  0.  1.]]
```

2D Convolution (contd.)

- Applying the convolution function `convo2d` on Slide 6 to the input array `arr` and the kernel `ker` defined on the previous slide, we get

```
convo_out = convo2d(arr, ker)

print(convo_out)
#           [[0. 0. 9. 9. 0. 0.]
#           [0. 0. 9. 9. 0. 0.]
#           [0. 0. 9. 9. 0. 0.]
#           [0. 0. 9. 9. 0. 0.]
#           [0. 0. 9. 9. 0. 0.]
#           [0. 0. 9. 9. 0. 0.]]

print(convo_out.shape)           ## (6, 6)
```

- The size of our input array went down from (8,8) to (6,6).
- There will be almost nothing left of this “image” after 3 or 4 application so this convolutional operator.**
- Now think of a 256×256 image and think of a convolutional processing chain that uses several rounds of $k \times k$ kernels. You may not be left with much of the image at the output of the chain.

Outline

- 1 2D Convolution — The Basic Definition 5
- 2 **What About `scipy.signal.convolve2d()` for 2D Convolutions** 9
- 3 **Input and Kernel Specs for PyTorch's Convolution Function**
`torch.nn.functional.conv2d()` 12
- 4 **Squeezing and Unsqueezing the Tensors** 18
- 5 **Using `torch.nn.functional.conv2d()`** 26
- 6 **2D Convolutions with the PyTorch Class `torch.nn.Conv2d`** 28
- 7 **Verifying That a PyTorch Convolution is in Reality a Cross-Correlation** 36
- 8 **Multi-Channel Convolutions** 40
- 9 **Reshaping a Tensor with `reshape()` and `view()`** 52

Using `scipy.signal.convolve2d()`

- Let's now try the popular `scipy.signal.convolve2d()` for 2D convolutions. In the following function call, we use the same 8×8 input array `arr` and the same 3×3 kernel `ker` that were defined earlier on Slide 7.

```
output = scipy.signal.convolve2d(arr, ker, mode='valid')
print(output)

#           [[0. 0. 9. 9. 0. 0.]
#           [0. 0. 9. 9. 0. 0.]
#           [0. 0. 9. 9. 0. 0.]
#           [0. 0. 9. 9. 0. 0.]
#           [0. 0. 9. 9. 0. 0.]
#           [0. 0. 9. 9. 0. 0.]

print(out.shape)      ## (6, 6)
```

- Pay particular attention to the option string `mode='valid'` in the call to the convolution function. What that means is that we want the convolution function to only use the actual input-array pixels — and not any hallucinated pixels outside the input-array just for the sake of returning an 8×8 array.

Comparing My `convo2d()` with `scipy.signal.convolve2d()`

- So you see that my implementation of 2D convolution on Slide 6 is the same as what's produced by `scipy.signal.convolve2d()` when the latter is used in the `valid` mode.
- We could have tried using `scipy.signal.convolve2d()` with a different set of options as in

```
output = scipy.signal.convolve2d(arr, ker, mode='full', boundary='symm')
```

But that would have required making assumptions about the pixels outside the input array — **which may not always be a good thing to do.**

- **So what we have here is a dilemma:** Either we choose to accept a shrunk convolutional output that uses only valid pixels, or we choose an output that is of the same size as the input, but is based on assumptions at the imagined pixels beyond the border.

Outline

- 1 2D Convolution — The Basic Definition 5
- 2 What About `scipy.signal.convolve2d()` for 2D Convolutions 9
- 3 Input and Kernel Specs for PyTorch's Convolution Function**
`torch.nn.functional.conv2d()` 12
- 4 Squeezing and Unsqueezing the Tensors 18
- 5 Using `torch.nn.functional.conv2d()` 26
- 6 2D Convolutions with the PyTorch Class `torch.nn.Conv2d` 28
- 7 Verifying That a PyTorch Convolution is in Reality a
Cross-Correlation 36
- 8 Multi-Channel Convolutions 40
- 9 Reshaping a Tensor with `reshape()` and `view()` 52

Input Specs for PyTorch's torch.nn.functional.conv2d()

- Moving on to PyTorch's functions for convolutions, these only work on input tensors whose shapes corresponds to:

```
(batch_size, num_input_channels, image_height, image_width)
```

- In general, when your input data consists of images, you'll first need to convert each image into a numpy array, as you saw in my Week 2 lecture, and, subsequently, convert the numpy array into a tensor, again as described in that lecture. But you would still be faced with the problem of **packaging** the tensors obtained to conform to the format shown above. **How does one do that?**

[In deep-learning code, it is the job of the dataloader to take care of type conversions and tensor packaging. For popular datasets, you might be able to download the dataloader directly from the PyTorch website. However, for custom datasets, when you need to write your own code for the downloader, you'd yourself be doing the type conversions and tensor packaging.]

- Before I get into the issue of packaging a tensor into a desired shape, let's first talk about converting numpy arrays, in general, into tensors in the next couple of slides.

Input Specs for torch.nn.functional.conv2d() (contd.)

- Converting the input 8×8 numpy array `arr` of Slide 7 into a tensor:

```

tensor_arr = torch.from_numpy( arr )           ## arr is the same as on Slide 7
print(tensor_arr)

#          tensor([[4., 4., 4., 4., 1., 1., 1., 1.],
#                 [4., 4., 4., 4., 1., 1., 1., 1.],
#                 [4., 4., 4., 4., 1., 1., 1., 1.],
#                 [4., 4., 4., 4., 1., 1., 1., 1.],
#                 [4., 4., 4., 4., 1., 1., 1., 1.],
#                 [4., 4., 4., 4., 1., 1., 1., 1.],
#                 [4., 4., 4., 4., 1., 1., 1., 1.],
#                 [4., 4., 4., 4., 1., 1., 1., 1.]], dtype=torch.float64)      (A)

print(tensor_arr.shape)                       ## torch.Size([8, 8])
print(tensor_arr.size())                     ## torch.Size([8, 8])

print(type(arr))                             ## <type 'numpy.ndarray'>      (B)
print(type(arr[0,0]))                       ## <type 'numpy.float64'>      (C)

```

- As shown by the `dtype` value in Line (A), the data elements stored in the tensor returned by the `torch.from_numpy()` converter are of type `torch.float64`. As for the reason, note the datatype for the elements in the numpy array as shown in Line (C).
- All of the different tensor data types are listed at:

Kernel Specs for `torch.nn.functional.conv2d()`

- The kernels that you feed into `torch.nn.functional.conv2d()` must be tensors of shape:

```
(out_channels, in_channels, kernel_height, kernel_width)
```

- You'll notice that the shape specification for a kernel is NOT the same as for the input.**
- Before worrying about the kernel tensor reshaping issues as required by the above spec, let's go ahead and convert the 3×3 kernel shown on Slide 7 into a tensor:

```
tensor_ker = torch.from_numpy( ker )

print(tensor_ker)
#          tensor([[ -1.,  0.,  1.],
#                  [-1.,  0.,  1.],
#                  [-1.,  0.,  1.]])
#          dtype=torch.float64

print(tensor_ker.shape)      ## torch.Size([3, 3])
```

- Notice again the datatype for the elements of the kernel tensor.

Repackaging the Input and the Kernel Tensors

- Now we must reformat (or, you could say, repack) the input and the kernel tensors so that they correspond to the specs for the `torch.nn.functional.conv2d()` function. Here is a link to the doc page where you will see the shape specifications for the two tensors:

<https://pytorch.org/docs/stable/nn.functional.html>

- BTW, that doc page also has a very important link related to generating reproducible results (important while you are debugging your code):

<https://pytorch.org/docs/stable/notes/randomness.html>

- **Summary of the two shape specifications:**

input: (batch_size, num_input_channels, image_height, image_width)

kernel: (out_channels, in_channels, kernel_height, kernel_width)

For a square-shaped input or a kernel, you are allowed to use a single argument for the last two.

Repackaging the Input Tensor

- I'm finally getting back to the question raised on Slide 13: **How to REPACKAGE the input tensor so that its “axis 0” corresponds to the batch size and its “axis 1” to the number of input channels?**

[As you will see later, “repackaging” and “reshaping” are different concepts. The former may involve augmenting the axes associated with the data, the latter merely reorganizes the data elements in the existing data space.]

- This is how you do it in PyTorch:

```

tensor_arr = torch.unsqueeze(tensor_arr, 0)      ## Yes, you need two invocations      (A)
tensor_arr = torch.unsqueeze(tensor_arr, 0)      ##           of unsqueeze()                  (B)

#           tensor([[[[4., 4., 4., 4., 1., 1., 1., 1.],
#                    [4., 4., 4., 4., 1., 1., 1., 1.],
#                    [4., 4., 4., 4., 1., 1., 1., 1.],
#                    [4., 4., 4., 4., 1., 1., 1., 1.],
#                    [4., 4., 4., 4., 1., 1., 1., 1.],
#                    [4., 4., 4., 4., 1., 1., 1., 1.],
#                    [4., 4., 4., 4., 1., 1., 1., 1.],
#                    [4., 4., 4., 4., 1., 1., 1., 1.]]]], dtype=torch.float64)

print(tensor_arr.shape)                          ## torch.Size([1, 1, 8, 8])      (C)

```

- As shown in Line (C), this now has the correct shape for the input tensor.
- But what are the calls `torch.unsqueeze(tensor_arr, 0)` in Lines (A) and (B) actually doing and why do we need two such calls?

Outline

- 1 2D Convolution — The Basic Definition 5
- 2 What About `scipy.signal.convolve2d()` for 2D Convolutions 9
- 3 Input and Kernel Specs for PyTorch's Convolution Function
`torch.nn.functional.conv2d()` 12
- 4 Squeezing and Unsqueezing the Tensors 18**
- 5 Using `torch.nn.functional.conv2d()` 26
- 6 2D Convolutions with the PyTorch Class `torch.nn.Conv2d` 28
- 7 Verifying That a PyTorch Convolution is in Reality a
Cross-Correlation 36
- 8 Multi-Channel Convolutions 40
- 9 Reshaping a Tensor with `reshape()` and `view()` 52

But First Let's Talk About How to Talk About the Shape of a Tensor

- Let's say you are thinking out loud as you are staring at a tensor whose shape is $(1, 1, 16, 16)$. How would you describe such a data object to yourself?
- If you said to yourself that you were dealing with a 4-dimensional data object, **you are in deep trouble** — a bad place to be in this age of deep learning.
- If you are one of those who thinks of a **10-element vector** as a 10-dimensional data object and a 10×10 array as a 2-dimensional data object, **you are again in deep trouble**.
- Thinking of a **10-element vector** as a 10-dimensional data object and a 10×10 array as a 2-dimensional data object could create in you what's known as cognitive dissonance and you just don't want to go there.

Talking About the Shape of a Tensor (contd.)

- When a tensor has shape $(1, 1, 16, 16)$, **we say that the tensor has FOUR AXES**. The first entry in the shape is for Axis 0, the second entry for Axis 1, the third for Axis 2, and the last for Axis 3.
- For the same data object, whose shape is $(1, 1, 16, 16)$, we say that the dimensionality along Axis 0 is 1, the dimensionality along Axis 1 is also 1, the dimensionality along Axis 2 is 16, and the dimensionality along Axis 3 is also 16.
- Now we can refer to a vector as a data object of a single axis, a matrix as a data object of two axes, an RGB image as a data object with 3 axes, and so on.
- When a convolutional layer is meant for RGB images of size 256×256 , and the layer produces 64 channels at its output, the input to the layer is of shape $(3, 256, 256)$ and its output of shape $(64, 256, 256)$, both the input and the output data objects being of 3 axes.

Talking About the Shape of a Tensor (contd.)

- In your own mind's eye, can you now visualize the difference between the following three data objects: one of shape $(16, 16)$, the other of shape $(1, 16, 16)$, and the last of shape $(1, 1, 16, 16)$?
- Think of the first one as an image in the XY-plane, the second the same image in the same plane but in the ZXY-space, and the last as the same image in the same plane but in the WZXY-space.
- Let's say we are given a data object, an image actually, of shape $(16, 16)$ and we want it to be processed by a function that expects its inputs of shape, say, $(c, 16, 16)$ where the integer c stands for the number of channels associated with the image.
- Assume that ours is a grayscale image, implying that $c = 1$. So before we can invoke the function, we need to convert the image into the shape $(1, 16, 16)$. We can do that by calling on the `unsqueeze()` function as you saw earlier.

You Can Call `torch.unsqueeze()` on Different Axes

- In order to convert

```
tensor_A = torch.tensor( [1,2,3,4] )
print(tensor_A.shape)                ## torch.Size([4])
```

into the tensor

```
tensor_B = torch.tensor( [[1,2,3,4]] )    ## Note nested square-brackets
print(tensor_B.shape)                    ## torch.Size([1, 4])
```

I'd need to call `unsqueeze()` on Axis 0 of `tensor_x`:

```
tensor_B = torch.unsqueeze(tensor_A,0)
print(tensor_B)                        ## tensor([[1, 2, 3, 4]])
print(tensor_B.shape)                  ## torch.Size([1, 4])
```

- In general, you can call `unsqueeze()` on any axis of a tensor. While a call to `unsqueeze()` on axes other than 0 may yield results that at first sight look like a strange rearrangement of the data, you never lose any data in the process. That is, the overall data content in the augmented space is the same as in the original space.

Calling `torch.unsqueeze()` on Different Axes (contd.)

- Here is applying `unsqueeze()` to Axis 1 of the same tensor:

```

tensor_A = torch.tensor( [1,2,3,4] )
print(tensor_A.shape)           ## torch.Size([4])
tensor_B = torch.unsqueeze( tensor_A, 1 )
print(tensor_B)
#                               tensor([[1],
#                               [2],
#                               [3],
#                               [4]])
print(tensor_B.shape)           ## torch.Size([4, 1])

```

- Applying `unsqueeze()` to Axis 1 made the dimensionality along that axis to 1. If you want to associate the matrix imagery with what happened, it turned a row vector into a column vector.
- The next slide shows a slightly more elaborate example in which we apply `unsqueeze()` to a 3-Axis tensor defined in the upper half of the slide. Comparing the shapes shown in Lines (A) and (B), do you understand the data rearrangement caused by the function?

An Example of Calling `torch.unsqueeze()` on Axis 1

```

tensor_in = torch.tensor( [ [[1,2,1,2], [3,4,3,4], [1,3,1,3]], [[5,6,5,6], [6,7,6,7], [8,9,8,9]] ] )
print(tensor_in)
#           tensor([[[[1, 2, 1, 2],
#                    [3, 4, 3, 4],
#                    [1, 3, 1, 3]],
#                   [[5, 6, 5, 6],
#                    [6, 7, 6, 7],
#                    [8, 9, 8, 9]]]])
print(tensor_in.shape)           ## torch.Size([2, 3, 4])           (A)

```

```

tensor_out = torch.unsqueeze(tensor_in, 1)
print(tensor_out)
#           tensor([[[[1, 2, 1, 2],
#                    [3, 4, 3, 4],
#                    [1, 3, 1, 3]]],
#                  [[5, 6, 5, 6],
#                    [6, 7, 6, 7],
#                    [8, 9, 8, 9]]]])
print(tensor_out.shape)         ## torch.Size([2, 1, 3, 4])         (B)

```

The Function `squeeze()`

- For the opposite of data space augmentation, there's the function `squeeze()` that drops **all axes** whose dimensionality is just 1.
- This example illustrates the idea:

```
x = torch.tensor( [[[[[1,2,3,4]]]] ] )
print(x)                ## tensor([[[[1, 2, 3, 4]]]])
print(x.shape)          ## torch.Size([1, 1, 1, 4])
x = torch.squeeze(x)
print(x)                ## tensor([1, 2, 3, 4])
print(x.shape)          ## torch.Size([4])
```

- What if the axis whose dimensionality is 1 is in the middle?

```
x = torch.zeros(4,1,5)
print(x)
# tensor([[[[0., 0., 0., 0., 0.],
#          [0., 0., 0., 0., 0.],
#          [0., 0., 0., 0., 0.],
#          [0., 0., 0., 0., 0.]]]])
x = torch.squeeze(x)
print(x)
# tensor([[0., 0., 0., 0., 0.],
#         [0., 0., 0., 0., 0.],
#         [0., 0., 0., 0., 0.],
#         [0., 0., 0., 0., 0.]])
```

Outline

- 1 2D Convolution — The Basic Definition 5
- 2 What About `scipy.signal.convolve2d()` for 2D Convolutions 9
- 3 Input and Kernel Specs for PyTorch's Convolution Function
`torch.nn.functional.conv2d()` 12
- 4 Squeezing and Unsqueezing the Tensors 18
- 5 **Using `torch.nn.functional.conv2d()`** 26
- 6 2D Convolutions with the PyTorch Class `torch.nn.Conv2d` 28
- 7 Verifying That a PyTorch Convolution is in Reality a
Cross-Correlation 36
- 8 Multi-Channel Convolutions 40
- 9 Reshaping a Tensor with `reshape()` and `view()` 52

Finally, Using torch.nn.functional.conv2d()

- Now that you know how to repackage the input tensor so that its shape is as specified on Slide 16, we must do the same for the kernel shown on Slide 15:

```

tensor_ker = torch.unsqueeze(tsr_ker, 0)           ## You need both these calls.
tensor_ker = torch.unsqueeze(tsr_ker, 0)         ## And, yes, they are identical.
print(tensor_ker)
#          tensor([[[[-1.,  0.,  1.],
#                    [-1.,  0.,  1.],
#                    [-1.,  0.,  1.]]]], dtype=torch.float64)
print(tensor_ker.shape)    ## torch.Size([1, 1, 3, 3])          [ out_ch, in_ch, ker_H, ker_W ]

```

- At long last, we are ready to use torch.nn.functional.conv2d():

```

output = torch.nn.functional.conv2d( tensor_arr, tensor_ker, stride=1)
print(output)
#          tensor([[[[ 0.,  0., -9., -9.,  0.,  0.],
#                    [ 0.,  0., -9., -9.,  0.,  0.],
#                    [ 0.,  0., -9., -9.,  0.,  0.],
#                    [ 0.,  0., -9., -9.,  0.,  0.],
#                    [ 0.,  0., -9., -9.,  0.,  0.],
#                    [ 0.,  0., -9., -9.,  0.,  0.]]]], dtype=torch.float64)

print(output.shape)    ## torch.Size([1, 1, 6, 6])          [B, C, H, W]

```

- Compare this convolution output to what was produced by `scipy.signal.convolution2d()` on Slide 10. **The signs of the nonzero entries are reversed!!!!!!!!!!!!!! But, why?**

Outline

- 1 2D Convolution — The Basic Definition 5
- 2 What About `scipy.signal.convolve2d()` for 2D Convolutions 9
- 3 Input and Kernel Specs for PyTorch's Convolution Function
`torch.nn.functional.conv2d()` 12
- 4 Squeezing and Unsqueezing the Tensors 18
- 5 Using `torch.nn.functional.conv2d()` 26
- 6 2D Convolutions with the PyTorch Class `torch.nn.Conv2d` 28**
- 7 Verifying That a PyTorch Convolution is in Reality a
Cross-Correlation 36
- 8 Multi-Channel Convolutions 40
- 9 Reshaping a Tensor with `reshape()` and `view()` 52

The Class `torch.nn.Conv2d` and its Callable Instances

- This slide brings us to the most important section of this lecture: a presentation of `torch.nn.Conv2d`. You need to become very familiar with `torch.nn.Conv2d` because of the role it's going to play in your own networks for deep learning.
- Note that, unlike what we have dealt with so far regarding convolutions, `torch.nn.Conv2d` is NOT a function. It is a Python class. You construct instances of this class and, because they are callable, you can use them like functions.
- Because you are now dealing with a class, you now have two types of specifications to deal with: **The parameter structure for calling the constructor and the parameter structure for invoking the callable instance.**

The Class `torch.nn.Conv2d` (contd.)

- The three **required** arguments for calling the **constructor** of `torch.nn.Conv2d` are:

```
(in_channels, out_channels, kernel_size)
```

The last argument, `kernel_size`, can be a scalar when using a square kernel, or a tuple for non-square kernels.

- A callable instance of `torch.nn.Conv2d` is subsequently called with an input tensor as its only argument. This input tensor must be of the following shape in which `H` and `W` refer to the height and the width of the image array.

```
(batch_size, in_channels, H, W)
```

- **But where is the kernel specified for `torch.nn.Conv2d`?**
- **By default, the kernel is implicit.** Since the kernel is a learnable entity, the user need not worry about its precise specification — **except, of course, for its shape.**

Using an Instance of `torch.nn.Conv2d` on Random Input

- Here we construct an instance of `torch.nn.Conv2d` for the **constructor parameters** `in_channels=1`, `out_channels=1`, and `kernel_size=3`:

```
conop = nn.Conv2d(1, 1, 3, bias=False)          ## default for 'stride' is 1 and 'bias' is True
```

- I'll now construct a random input tensor of the needed shape for the callable convolutional operator `conop` defined above:

```
input = torch.randn(1, 1, 8, 8)
```

```
print(input)
#          tensor([[[[ 0.4372,  0.4913, -0.2041, -0.0885,  0.5239, -0.6659,  0.8504, -1.3527],
#                    [-1.3453,  0.7854,  0.9928, -0.1932, -0.3090,  0.5026, -0.8594,  0.7502],
#                    [-0.1577,  1.4437,  0.2660,  0.1665,  0.8744, -0.1435, -0.1116, -0.6136],
#                    [ 1.2590,  2.0050,  0.0537,  0.6181, -0.4128, -0.8411, -2.3160, -0.1023],
#                    [-0.7425,  0.5627,  0.2596, -0.1740, -0.6787,  0.9383,  0.4889, -0.6731],
#                    [ 0.0845, -1.2001, -0.0048, -0.5181, -0.3067, -1.5810,  1.7066, -0.4462],
#                    [-0.4503, -0.5731, -0.5554,  0.5943,  1.5419,  0.5073, -0.5910, -0.5692],
#                    [ 0.1886, -0.0691, -0.4949, -1.4959, -0.1938,  0.4455,  1.3253, -1.6293]]]])
```

```
print(input.shape)          ## (1,1,8,8)
print(input.type())        ## torch.FloatTensor          ## dtype is 32-bit floats
```

Using an Instance of `torch.nn.Conv2d` on Random Input (contd.)

- Here is the convolution with the input shown on the previous slide with a 3×3 kernel:

```
output = conop(input)

print(output)
#          tensor([[[[ 0.8354, -0.2950, -0.3890,  0.6428, -0.4596,  0.5685],
#                    [ 0.2685, -0.1622,  0.2145, -0.4469, -0.0307, -0.9696],
#                    [-0.2300, -0.3751, -0.3678, -0.1845,  0.2430,  0.7862],
#                    [ 0.1561, -0.3299, -0.0062,  0.3382,  0.1095, -0.2624],
#                    [ 0.0321,  0.1809,  0.3099,  0.0354,  0.6075,  0.2559],
#                    [-0.0840,  0.3232, -0.1971,  0.0273, -1.1666,  0.7276]]]])],
#          grad_fn=<MkldnnConvolutionBackward>)

print(output.shape)          ## torch.Size([1, 1, 6, 6])
print(output.type())        ## torch.FloatTensor
print( type(output) )      ## <class 'torch.Tensor'>
                           ## NOTE: tensor.type() is more useful than type( tensor )
```

- Note that whereas the input was of size 8×8 , the output is of size 6×6 . That is the because, by default, the convolution is carried in the valid mode that you saw earlier on Slide 10.

Accessing the Kernel Used for the Convolution

- About the kernel used by `convop` for the convolution output shown on the previous slide, what if at this very moment we had an uncontrollable desire to eat pizza? Sorry! I must be hungry. Meant to say what if we have an uncontrollable desire to see what exactly was used for the kernel. Is it possible to do that?
- Yes, by accessing the `weight` attribute of the `convop` operator:

```
ker = convop.weight

print(ker)
#           Parameter containing:
#           tensor([[[[-0.0025,  0.1788, -0.2743],
#                    [-0.2453, -0.1284,  0.0894],
#                    [-0.0066,  0.2643, -0.0296]]]], requires_grad=True)
#
print(ker.shape)           ## (1, 1, 3, 3)      [out_ch, in_ch, ker_h, ker_w]
print(ker.type())         ## torch.FloatTensor

print( type(ker) )        ## <'torch.nn.parameter.Parameter'>
```

- Note the two special “things” about the `weight` attribute that I have printed out: It is of type `torch.nn.Parameter` and its `requires_grad` property is set. [As you already know, a quantity is not learnable unless its `requires_grad` property is set.]

Using an Instance of `torch.nn.Conv2d` on a Deterministic Input and With Our Own Kernel

- In order to experiment with our own kernels, let's apply the `convop` operator to the same deterministic input that you saw earlier on Slide 17. (The previous `convop` output on Slide 32 was for random input.) Here is the input tensor from Slide 17:

```
print(tensor_arr)
#          tensor([[[[4., 4., 4., 4., 1., 1., 1., 1.],
#                    [4., 4., 4., 4., 1., 1., 1., 1.],
#                    [4., 4., 4., 4., 1., 1., 1., 1.],
#                    [4., 4., 4., 4., 1., 1., 1., 1.],
#                    [4., 4., 4., 4., 1., 1., 1., 1.],
#                    [4., 4., 4., 4., 1., 1., 1., 1.],
#                    [4., 4., 4., 4., 1., 1., 1., 1.],
#                    [4., 4., 4., 4., 1., 1., 1., 1.]]]], dtype=torch.float64)
print(tensor_arr.shape)          ## (1,1,8,8)
print(tensor_arr.type())        ## torch.DoubleTensor          (A)
```

- As shown in Line (A), we note that the datatype of our input-tensor elements is **not** the same as what `torch.nn.Conv2d` uses for its internally generated kernel. So we do the type conversion:

```
tensor_arr = tensor_arr.float()
print(tensor_arr.type())        ## torch.FloatTensor          (B)
```

With Our Own Input and Kernel (contd.)

- Now we specify our kernel:

```
ker = numpy.array([[[-1,0,1],
                   [-1,0,1],
                   [-1,0,1]])
ker = torch.from_numpy(ker)
print( ker.type() )           ## torch.LongTensor
ker = ker.float()            ## this makes ker elements 32-bit floats
ker = torch.unsqueeze(ker, 0)
ker = torch.unsqueeze(ker, 0)
```

- And, define the convolutional operator **and set its weight attribute**:

```
conop = nn.Conv2d(1, 1, 3, bias=False)           # assumes a stride of 1
conop.weight = torch.nn.Parameter( ker )
```

- Finally, we are ready to carry out the convolution:

```
output = conop(tensor_arr)
print(output)
#           tensor([[[[ 0.,  0., -9., -9.,  0.,  0.],
#                    [ 0.,  0., -9., -9.,  0.,  0.],
#                    [ 0.,  0., -9., -9.,  0.,  0.],
#                    [ 0.,  0., -9., -9.,  0.,  0.],
#                    [ 0.,  0., -9., -9.,  0.,  0.],
#                    [ 0.,  0., -9., -9.,  0.,  0.] ] ] ]],
#           grad_fn=<MklDnnConvolutionBackward>)
print(output.shape)      ## torch.Size([1, 1, 6, 6])
```

Outline

- 1 2D Convolution — The Basic Definition 5
- 2 What About `scipy.signal.convolve2d()` for 2D Convolutions 9
- 3 Input and Kernel Specs for PyTorch's Convolution Function
`torch.nn.functional.conv2d()` 12
- 4 Squeezing and Unsqueezing the Tensors 18
- 5 Using `torch.nn.functional.conv2d()` 26
- 6 2D Convolutions with the PyTorch Class `torch.nn.Conv2d` 28
- 7 Verifying That a PyTorch Convolution is in Reality a
Cross-Correlation 36**
- 8 Multi-Channel Convolutions 40
- 9 Reshaping a Tensor with `reshape()` and `view()` 52

Specifying the Input and the Kernel for the Convolution vs. Correlation Test

- Here is the input:

```
input = torch.zeros(1,1,8,8,dtype=float)
input[0,0,:,4] = 1
print(input)
#           tensor([[[[0., 0., 0., 0., 1., 0., 0., 0.],
#                    [0., 0., 0., 0., 1., 0., 0., 0.],
#                    [0., 0., 0., 0., 1., 0., 0., 0.],
#                    [0., 0., 0., 0., 1., 0., 0., 0.],
#                    [0., 0., 0., 0., 1., 0., 0., 0.],
#                    [0., 0., 0., 0., 1., 0., 0., 0.],
#                    [0., 0., 0., 0., 1., 0., 0., 0.],
#                    [0., 0., 0., 0., 1., 0., 0., 0.]]]]], dtype=torch.float64)
```

- In this case, I have specified the input tensor in its correct 4-Axis shape in one go by directly calling `torch.zeros()` function as shown.
- Since in PyTorch programming, it is not uncommon to go back and forth between numpy and torch, note that the syntax for `numpy.zeros()` in Slide 7 is NOT identical to that for `torch.zeros()` shown above. The former is typically called with two args, as shown on Slide 7, one a tuple for the shape, and the other for the datatype for the elements.

Specifying the Input and the Kernel for the Test (contd.)

- And here is the kernel:

```
ker = numpy.zeros((1,1,3,3), dtype=float)
print(ker)
#           array([[[[0., 0., 0.],
#                   [0., 0., 0.],
#                   [0., 0., 0.]])])
ker[0,0,:] = [1,2,3]           # NOTE: You can't do this with tensors directly
#                             # because they are more strongly datatyped
print(ker)
#           array([[[[1., 2., 3.],
#                   [1., 2., 3.],
#                   [1., 2., 3.]])])
ker = torch.from_numpy(ker)
```

- Do you understand why I have constructed the input and the kernel in the manner shown? If not, you need to think about what is different between a convolution and a cross-correlation.

Aquí Está el Resultado de la Prueba

- Now we are ready to do the convolution:

```

conop = nn.Conv2d(1, 1, 3, bias=False)
conop.weight = nn.Parameter( ker )
output = conop(input)
print(output)
#          tensor([[[[0., 0., 9., 6., 3., 0.],
#                    [0., 0., 9., 6., 3., 0.],
#                    [0., 0., 9., 6., 3., 0.],
#                    [0., 0., 9., 6., 3., 0.],
#                    [0., 0., 9., 6., 3., 0.],
#                    [0., 0., 9., 6., 3., 0.]]]],
#                grad_fn=<MklDnnConvolutionBackward>)
print(output.shape)          ## torch.Size([1, 1, 6, 6])

```

- Can you tell that only a cross-correlation could have produced this output?

Outline

- 1 2D Convolution — The Basic Definition 5
- 2 What About `scipy.signal.convolve2d()` for 2D Convolutions 9
- 3 Input and Kernel Specs for PyTorch's Convolution Function
`torch.nn.functional.conv2d()` 12
- 4 Squeezing and Unsqueezing the Tensors 18
- 5 Using `torch.nn.functional.conv2d()` 26
- 6 2D Convolutions with the PyTorch Class `torch.nn.Conv2d` 28
- 7 Verifying That a PyTorch Convolution is in Reality a
Cross-Correlation 36
- 8 Multi-Channel Convolutions 40**
- 9 Reshaping a Tensor with `reshape()` and `view()` 52

Questions Related to Multi-Channel Convolutions

- The first convolutional layer in a network meant for processing color images is likely to have 3 input channels, one for each color channel.
- And it would not be uncommon for the same first layer to have 64 or 128 or even 256 output channels.
- And, again, it would not be uncommon for there to be another convolutional layer further up the stack whose input channels and output channels would both equal, say, 128.
- That raises the following sorts of questions:
 - What is the shape of the kernel tensor for `in_ch=3` and `out_ch=128`?
 - And what might the kernel tensor look like for the case `in_ch=128` and `out_ch=128`?
 - When `in_ch == out_ch`, is it possible to set up, say, a 1-1 convolutional connection between the input and the output channels? That is, can we tell PyTorch that each channel on the input side should contribute values to only the corresponding channel on the output side?

Specifying a Two-Channel Input

- As you will recall from Slide 30, an instance of `torch.nn.Conv2d` is invoked as a callable and the input data fed into it must have four axes: the first is for the batch size, the second for the number of input channels, the third for the height of the image, and the last for the width of the image:

```
input = torch.zeros(1,2,8,8)           # args:  batch_size in_ch, H, W
input[0,0,:,2] = 1.0                   #      ch index is 0
input[0,1,:,6] = 1.0                   #      ch index is 1

print(input.type())                    # torch.FloatTensor
print( input[0,0,0,0].type() )         # torch.FloatTensor
```

- As you can see, by default, the `torch.zeros()` returns a tensor with 32-bit floats for the element type. However, if we had created the tensor with the call `torch.zeros(1,2,8,8, dtype=float)`, it would have set the tensor element type to `torch.float64` and type of the tensor to `torch.DoubleTensor`.
I find that so annoying.
- As you might have surmised from the previous slides, by default `torch.nn.Conv2d` class likes to do its thing with 32-bit floats.

Specifying a Two-Channel Input

- Let's look at the input we constructed on the previous slide:

```
print(input)
#          tensor([[[[0., 0., 1., 0., 0., 0., 0., 0.],
#                    [0., 0., 1., 0., 0., 0., 0., 0.],
#                    [0., 0., 1., 0., 0., 0., 0., 0.],
#                    [0., 0., 1., 0., 0., 0., 0., 0.],
#                    [0., 0., 1., 0., 0., 0., 0., 0.],
#                    [0., 0., 1., 0., 0., 0., 0., 0.],
#                    [0., 0., 1., 0., 0., 0., 0., 0.],
#                    [0., 0., 1., 0., 0., 0., 0., 0.]],
#                  [[0., 0., 0., 0., 0., 0., 1., 0.],
#                    [0., 0., 0., 0., 0., 0., 1., 0.],
#                    [0., 0., 0., 0., 0., 0., 1., 0.],
#                    [0., 0., 0., 0., 0., 0., 1., 0.],
#                    [0., 0., 0., 0., 0., 0., 1., 0.],
#                    [0., 0., 0., 0., 0., 0., 1., 0.],
#                    [0., 0., 0., 0., 0., 0., 1., 0.],
#                    [0., 0., 0., 0., 0., 0., 1., 0.]])])
#
#          ## torch.Size([1, 2, 8, 8])
print(input.shape)
```

- I intentionally chose these two patterns for the two input channels in order to see how the input channels would contribute to the single output channel I will use for the convolution example on the next slide (that is based on `in_ch=2` and `out_ch=1`).

Examining the Kernel for `in_ch=2` and `out_ch=1`

- Here is an instance of `torch.nn.Conv2d` for `in_ch=2` and `out_ch=1`:

```

conop = nn.Conv2d(2, 1, 3, bias=False)          ## ker size 3x3 and assumes a convo stride of 1
ker = conop.weight
print(ker)
#          tensor([[[[ 0.0624, -0.0712, -0.0463],
#                    [-0.2252, -0.1561, -0.0972],
#                    [ 0.0087,  0.0932,  0.1414]],
#
#                    [[-0.1598, -0.1026,  0.0856],
#                    [ 0.1957, -0.0485,  0.1764],
#                    [-0.0380,  0.0249,  0.2134]]]])], requires_grad=True)

print(ker.shape)                               ## torch.Size([1, 2, 3, 3])      [This is in keeping with the kernel
##                                             specs at the bottom of Slide 16]

output = conop(input)                          ## input is as shown on the previous slide
print(output)
#          tensor([[[[-0.0021, -0.1342, -0.1541,  0.0000,  0.4754, -0.1262],
#                    [-0.0021, -0.1342, -0.1541,  0.0000,  0.4754, -0.1262],
#                    [-0.0021, -0.1342, -0.1541,  0.0000,  0.4754, -0.1262],
#                    [-0.0021, -0.1342, -0.1541,  0.0000,  0.4754, -0.1262],
#                    [-0.0021, -0.1342, -0.1541,  0.0000,  0.4754, -0.1262],
#                    [-0.0021, -0.1342, -0.1541,  0.0000,  0.4754, -0.1262]]]])],
#          grad_fn=<MkldnnConvolutionBackward>)

print(output.shape)                            ## torch.Size([1, 1, 6, 6])

```

- Do you understand why the kernel is shaped the way it is for the case of a 2-channel input?

Another Look at the Output for `in_ch=2` and `out_ch=1`

- In the output shown on the previous slide, can you visualize the relationship between (1) the output, (2) the two 3×3 operators in the kernel, and (3) the two input channels?

[[Regarding the output shown on the previous slide, by running the first 3×3 operator in the kernel over the first channel of the input, you can tell that the first 3 columns of the output are a result of the convolution produced by those two. Similarly, you can verify that the last three columns in the output are a result of the second 3×3 operator in the kernel convolving with the the second channel.]

- To make it more convenient to visualize the relationship between the three, perhaps the following discretized form of the output would help:

```
output = output * 10 + 0.5
output = output.type(torch.int8)
print(output)
#           tensor([[[[ 0,  0, -1,  0,  5,  0],
#                    [ 0,  0, -1,  0,  5,  0],
#                    [ 0,  0, -1,  0,  5,  0],
#                    [ 0,  0, -1,  0,  5,  0],
#                    [ 0,  0, -1,  0,  5,  0],
#                    [ 0,  0, -1,  0,  5,  0]]]], dtype=torch.int8)
```

Examining the Kernel for `in_ch=2` and `out_ch=3`

- For a more “complex” kernel, here is an instance of `torch.nn.Conv2d` for `in_ch=2` and `out_ch=3`:

```

conop = nn.Conv2d(2, 3, 3, bias=False)          ## args: in_channel out_channel ker_size
ker = conop.weight
print(ker)

#
#          | tensor([[[[-0.2187, -0.1484, -0.0597],
#          |          | [-0.0919,  0.2036, -0.1528],
#          |          | [-0.1085, -0.1647, -0.2207]],
#          |          |
#          |          | [[-0.1376,  0.2026,  0.1052],
#          |          | [ 0.1142,  0.0124, -0.1208],
#          |          | [ 0.0399, -0.2201, -0.1703]]]),
#          |
#          |
#          |          | [[[-0.1215,  0.1487,  0.1382],
#          |          | [-0.1045, -0.0085,  0.1507],
#          |          | [ 0.2343,  0.0935,  0.0318]],
#          |          |
#          |          | [[ [ 0.1580, -0.1388,  0.0439],
#          |          | [-0.1827, -0.1634, -0.1218],
#          |          | [ 0.1066,  0.0948, -0.1396]]],
#          |          |
#          |          |
#          |          | [[[ 0.0712,  0.1294, -0.0297],
#          |          | [ 0.0090,  0.0546,  0.1462],
#          |          | [ 0.2263, -0.1816, -0.0864]],
#          |          |
#          |          | [[ [ 0.0926,  0.1953,  0.2051],
#          |          | [ 0.2080,  0.0469, -0.2050],
#          |          | [ 0.0217, -0.1475, -0.2197]]]], requires_grad=True)
#
#
print(ker.shape)          ## torch.Size([3, 2, 3, 3])

```

The `in_ch=2` and `out_ch=3` Case (contd.)

- Regarding the shape of the kernel shown on the previous slide — `torch.Size([3, 2, 3, 3])` — Axis 0 corresponds to the output channels, Axis 1 to the input channels, Axis 2 to H, and Axis 3 to W. **Again, this is in keeping with the kernel specs shown at the bottom of Slide 16.** [By this time you should have gotten used to the fact that the order in which you specify the number of input channels and the number of output channels is opposite for the `nn.Conv2d` constructor and for the kernel it needs.]
- After discretization in the same manner as before, here is the output for this convolution:

```

output = convop(input)
output = output * 10 + 0.5
output = output.type(torch.int8)
print(output)
#
#          tensor([[[[-3,  0, -3,  0, -1,  0],
#                    [-3,  0, -3,  0, -1,  0],
#                    [-3,  0, -3,  0, -1,  0],
#                    [-3,  0, -3,  0, -1,  0],
#                    [-3,  0, -3,  0, -1,  0],
#                    [-3,  0, -3,  0, -1,  0],
#                    [ 3,  2,  0,  0, -1, -1],
#                    [ 3,  2,  0,  0, -1, -1],
#                    [ 3,  2,  0,  0, -1, -1],
#                    [ 3,  2,  0,  0, -1, -1],
#                    [ 3,  2,  0,  0, -1, -1],
#                    [ 3,  2,  0,  0, -1, -1],
#                    [ 0,  0,  3,  0, -1,  1],
#                    [ 0,  0,  3,  0, -1,  1],
#                    [ 0,  0,  3,  0, -1,  1],
#                    [ 0,  0,  3,  0, -1,  1],
#                    [ 0,  0,  3,  0, -1,  1],
#                    [ 0,  0,  3,  0, -1,  1]]]]) dtype=torch.int8)

```

Using the `groups` Option in the `torch.nn.Conv2d` Constructor Call

- Let's now consider the `groups` parameter that controls how the input channels are grouped together for contributing to the output channels.
- The default value for this option is 1. So if you set `groups=1` when you construct an instance of `torch.nn.Conv2d`, you get the same kernel structure that you saw on the previous slide: Each output channel is produced by summing the outputs from ALL input channels.
- However, if you set `groups` to a value greater than 1, you have to be careful. In this case, both `in_ch` and `out_ch` must be divisible by `groups`.
- What that implies is that the following constructor call would be illegal:

```
conop = nn.Conv2d(2, 3, 3, groups=2, bias=False)
```

- In the next slide, I'll consider the case of `groups=4` when both `in_ch` and `out_ch` are also equal to 4.

Using groups=4 with in_ch=4 and out_ch=4

- When both `in_ch` and `out_ch` are the same and the value of `groups` is also the same number, you basically have a 1-1 connection between the input channels and the output channels.
- What that means, a single kernel operator will work on a single input channel and the result will be the corresponding output channel. Here is an example:

```

conop = nn.Conv2d(4, 4, 3, groups=4, bias=False)          # args: in_ch, out_ch, ker_size

ker = conop.weight
print(ker)
#
#           Parameter containing:
#           tensor([[[[ 0.0675,  0.2119,  0.3157],
#                    [ 0.2117,  0.3165, -0.0241],
#                    [-0.2994, -0.1580,  0.2270]]],
#                  [[[-0.0022, -0.1657, -0.2554],
#                    [-0.3120, -0.2813, -0.0676],
#                    [ 0.1828,  0.1802, -0.3215]]],
#                  [[[ 0.2079, -0.2608, -0.0705],
#                    [-0.1352, -0.0642, -0.0654],
#                    [-0.2991, -0.2878, -0.0522]]],
#                  [[[ 0.0043, -0.1514,  0.1256],
#                    [-0.3000, -0.0225,  0.2931],
#                    [-0.1360,  0.3010,  0.1207]]]],
#                 requires_grad=True)
#
#           ## The 3x3 op for that maps
#           ## the first input ch to the
#           ## first output ch
#
#           ## The 3x3 op that maps the
#           ## second input ch to the
#           ## second output ch
#
#           ## The 3x3 op that maps the
#           ## third input ch to the
#           ## third output ch
#
#           ## The 3x3 op that maps the
#           ## fourth input ch to the
#           ## fourth output ch

print(ker.shape)                                     ## torch.Size([4, 1, 3, 3])

```

Using groups=2 with in_ch=4 and out_ch=4

- When the value of **groups** divides both **in_ch** and **out_ch**, you have groupings of input channels sending information to the corresponding groupings of the output channels.
- For example, when **in_ch=16**, **out_ch=64**, and **groups=4**, and consider, for illustration, the case when the kernel size is 3×3 . In this case, there will exist sixteen 3×3 operators for each input channel, with each operator sending its output to one of the channels in a grouping of 16 output channels. In this case, you will have a total of $256 \ 3 \times 3$ operators in the kernel.
- The next slide shows an example of the kernel tensor when **groups=2** and with **in_ch=4** and **out_ch=4**.

[In the next slide, each grouping of 2 channels at the input will contribute to each grouping of two channels at the output. Within each group-to-group connections, you will need four 3×3 operators. Each 3×3 operator will take one input channel to all the output channels within each group. That calls for a total of eight 3×3 operators that are shown below.]

Using groups=2 with in_ch=4 and out_ch=4 (contd.)

- Shown below is the constructor call for the case when `in_ch=4` and `out_ch=4` and `groups=2` and the resulting kernel tensor:

```

conop = nn.Conv2d(4, 4, 3, groups=2, bias=False)           # args: in_ch, out_ch, ker_size
ker = conop.weight
print(ker)
#
#      Parameter containing:
#      tensor([[[[ 0.1885, -0.1905,  0.0253],
#                [-0.0493,  0.1683,  0.0658],
#                [ 0.1133,  0.0832, -0.0567]],
#              [[-0.0496, -0.1942,  0.1277],
#                [ 0.1871,  0.1613, -0.1663],
#                [ 0.0105, -0.1662, -0.1298]]],
#            [[[-0.1373,  0.0806, -0.1405],
#                [-0.0051,  0.0099,  0.1519],
#                [-0.1782, -0.1618, -0.1369]],
#              [[ 0.1650, -0.0847,  0.1988],
#                [ 0.0852,  0.0298, -0.0018],
#                [-0.0466,  0.0296, -0.0538]]],
#            [[[-0.0017,  0.0301, -0.1844],
#                [-0.1235,  0.1903, -0.1913],
#                [-0.0169,  0.2332,  0.0851]],
#              [[ 0.0067, -0.2043,  0.1168],
#                [-0.1679, -0.0669, -0.0791],
#                [-0.0349,  0.0026,  0.1944]]],
#            [[[ 0.0294,  0.2111,  0.1442],
#                [-0.1490,  0.1057, -0.1666],
#                [-0.0999,  0.0693,  0.0778]],
#              [[ 0.1768, -0.0759,  0.0004],
#                [ 0.1213, -0.2279,  0.1704],
#                [-0.1949,  0.0032, -0.0401]]], requires_grad=True)
#
print(ker.shape)      ## torch.Size([4, 2, 3, 3])  Axis 0: out_ch  Axis 1: in_ch  Axis 2: H  Axis 3: W

```

Outline

- 1 2D Convolution — The Basic Definition 5
- 2 What About `scipy.signal.convolve2d()` for 2D Convolutions 9
- 3 Input and Kernel Specs for PyTorch's Convolution Function
`torch.nn.functional.conv2d()` 12
- 4 Squeezing and Unsqueezing the Tensors 18
- 5 Using `torch.nn.functional.conv2d()` 26
- 6 2D Convolutions with the PyTorch Class `torch.nn.Conv2d` 28
- 7 Verifying That a PyTorch Convolution is in Reality a
Cross-Correlation 36
- 8 Multi-Channel Convolutions 40
- 9 Reshaping a Tensor with `reshape()` and `view()` 52

Reshaping Tensors

- It is not at all uncommon that your implementation of a deep learning algorithm would call for reshaping a tensor before it is input to the next layer. This is especially the case when the last layer is fully-connected and the penultimate layer is convolutional. When such is the case, the output of the penultimate layer is likely to be of shape `(C, H, W)`, which would be a tensor with 3 axes (**not including batching**), while the shape of the tensor at the input to the fully-connected layer will need to have just one axis (**again not including batching**).
- The two most commonly used functions for reshaping a tensor in PyTorch are `view()` and `reshape()`.
- **While these two functions seem to do the same thing most of the time, they are not identical.**
- The goal of this section is to present examples of tensor reshaping with `view()` and `reshape()` and to explain the difference between the behaviors of these two important functions.

Introducing `reshape()` and `view()`

- On Slide 57, in order to demonstrate how you can carry out tensor reshaping with `reshape()` and `view()`, I start by constructing in Line (A) a 2-axis random tensor of shape (3, 4).
- Subsequently, in Line (C) we call on `view()` to reshape the originally (3, 4) tensor into a (2, 6) tensor. And, we do the same thing in Line (E) but by calling on `reshape()`.
- As you can see from the results displayed in Lines (D) and (F), both `view()` and `reshape()` are doing exactly the same thing.
- Just to illustrate that the tensors that are returned by `reshape()` and `view()` are independent of the original tensor `x`, in Line (G) we delete the tensor reference held by the variable `x` on which we had invoked the two reshaping functions. As shown in Lines (H) and (I), deleting `x` does not affect the tensor references held by the variables `y` and `z`.

Using -1 as a Reshaping Argument

- Continuing with the explanation of the `reshape1.py` script, the syntax shown in Lines (J) through (M) is more for fun than anything else. Despite the fact that we already know that the `y` and `z` tensors are identical (since they are both obtained from the same underlying tensor with exactly the same reshaping operation), we nonetheless test them for equality in a couple of different ways.
- The main goal of the code in Lines (N) through (V) is to illustrate the use of “-1” as one of the arguments to `reshape()` and `view()`.
- Using “-1” means that you want PyTorch to figure out what the value should be at that position in the call syntax. **PyTorch can easily do that out since any reshaping must use all of the data in the original tensor.**
- For example, in the call to `reshape()` in Line (Q), it is trivial to conclude that shape value to use where “-1” is indicated is 5. [That is because the original tensor has a total of 10 numbers in it and the call in Line (Q) wants to reshape the original tensor into a tensor of two axes with the stipulation that the dimensionality along the first axis is set to 2.]

Using -1 as a Reshaping Argument (contd.)

- Obviously, then, the dimensionality along the “-1” in the call in Line (Q) must be 5. Exactly the same logic applies to the call to `view()` in Line (U).
- It is important to understand why the tensor reshaping calls in Lines (W) and (Y) elicit run-time errors. The `x` tensor has 10 data elements in it. Since 3 is not a factor of 10, there is no way to reshape `x` into a tensor of two axes with one of the axes having dimension 3.

Using reshape() and view() in Code

```

## reshape1.py

import torch
torch.manual_seed(0)

x = torch.randn(3,4)                                     ## (A)
print(x)                                                 ## (B)
## tensor([[ 1.5410, -0.2934, -2.1788,  0.5684],
##         [-1.0845, -1.3986,  0.4033,  0.8380],
##         [-0.7193, -0.4033, -0.5966,  0.1820]])

y = x.view(2,6)                                         ## (C)
print(y)                                                 ## (D)
## tensor([[ 1.5410, -0.2934, -2.1788,  0.5684, -1.0845, -1.3986],
##         [ 0.4033,  0.8380, -0.7193, -0.4033, -0.5966,  0.1820]])

z = x.reshape(2,6)                                     ## (E)
print(z)                                                 ## (F)
## tensor([[ 1.5410, -0.2934, -2.1788,  0.5684, -1.0845, -1.3986],
##         [ 0.4033,  0.8380, -0.7193, -0.4033, -0.5966,  0.1820]])

del x                                                  ## (G)
print(y)                                                 ## (H)
## tensor([[ 1.5410, -0.2934, -2.1788,  0.5684, -1.0845, -1.3986],
##         [ 0.4033,  0.8380, -0.7193, -0.4033, -0.5966,  0.1820]])

print(z)                                               ## (I)
## tensor([[ 1.5410, -0.2934, -2.1788,  0.5684, -1.0845, -1.3986],
##         [ 0.4033,  0.8380, -0.7193, -0.4033, -0.5966,  0.1820]])

## torch.eq() and torch.tensor.eq() test for element-wise equality:
##
print(torch.eq(y,z))  ## tensor([[True, True, True, True, True, True],
##                               [True, True, True, True, True, True]])  ## (J)

print(y.eq(z))      ## tensor([[True, True, True, True, True, True],
##                               [True, True, True, True, True, True]])  ## (K)

```

Using reshape() and view() in Code (contd.)

(..... continued from the previous slide)

```

## An interesting construct from stackoverflow.com:
print(torch.all(torch.lt(torch.abs(torch.sum(y - z)), 1e-12)))      ## tensor(True)      ## (L)
print(torch.allclose(y,z))                                       ## True               ## (M)

## A second reshaping example to illustrate the use of "-1" as an arg:

x = torch.randn(10)                                             ## (N)

z = x.reshape(2,5)                                             ## (O)
print(z)                                                        ## tensor([[ -0.8567,  1.1006, -1.0712,  0.1227, -0.5663],      ## (P)
##           [ 0.3731, -0.8920, -1.5091,  0.3704,  1.4565]])

z = x.reshape(2,-1)                                           ## (Q)
print(z)                                                        ## tensor([[ -0.8567,  1.1006, -1.0712,  0.1227, -0.5663],      ## (R)
##           [ 0.3731, -0.8920, -1.5091,  0.3704,  1.4565]])

z = x.view(2,5)                                               ## (S)
print(z)                                                        ## tensor([[ -0.8567,  1.1006, -1.0712,  0.1227, -0.5663],      ## (T)
##           [ 0.3731, -0.8920, -1.5091,  0.3704,  1.4565]])

z = x.view(2,-1)                                              ## (U)
print(z)                                                        ## tensor([[ -0.8567,  1.1006, -1.0712,  0.1227, -0.5663],      ## (V)
##           [ 0.3731, -0.8920, -1.5091,  0.3704,  1.4565]])

# z = x.reshape(3, -1) ## RuntimeError: shape '[3, -1]' is invalid for input of size 10      ## (W)

# z = x.view(3, -1)    ## RuntimeError: shape '[3, -1]' is invalid for input of size 10      ## (Y)

```

But `view()` and `reshape()` Are Not the Same

- The code shown in the previous script, `reshape1.py`, might create the impression that the functions `view()` and `reshape()` possess identical behaviors. With the help of the next script, `flatten.py`, I'll now show that that's NOT the case.
- But first a bit on what is meant by *flattening* a tensor. **Flattening means reshaping a tensor into a single-axis data object — a vector really.** Think about flattening a matrix by scanning one row at a time and arranging all the elements in a single linear structure. We can extend this idea to any tensor by invoking `reshape(-1)` or `view(-1)`, as shown in Lines (E) through (H) on the next slide.
- **However, flattening a tensor with `view(-1)` works only if a tensor is contiguous.** Its meaning will be explained shortly. Suffice it to say here that whereas the tensor `y` created originally in Line (C) on the next slide is contiguous, the tensor `w`, obtained in Line (K) by transposing `y`, is NOT.
- When we try to flatten `w` by calling `view(-1)` in Line (P) on the next slide, we run into a runtime error from the system.

Flattening a Tensor

```

## flatten.py

import torch
torch.manual_seed(0)

x = torch.arange(10)
print(x)                                ## tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])    ## (A)
                                          ## (B)

## In this script, I'll use tensor y and its variants to test for contiguity
##
y = x.view(2,5)
print(y)                                ## y is contiguous --- stored in row-major format    ## (C)
                                          ## tensor([[0, 1, 2, 3, 4],                          ## (D)
                                          ##          [5, 6, 7, 8, 9]])

z1 = y.reshape(-1)
print(z1)                                ## Will always work since reshape() returns a new tensor    ## (E)
                                          ## tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])              ## (F)

z2 = y.view(-1)
print(z2)                                ## Works because y is contiguous                        ## (G)
                                          ## tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])              ## (H)

z3 = y.view(1,-1)
print(z3)                                ## tensor([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]])            ## (I)
                                          ## (J)

w = y.T
print(w)                                ## w is NOT contiguous                                  ## (K)
                                          ## tensor([[0, 5],                                     ## (L)
                                          ##          [1, 6],
                                          ##          [2, 7],
                                          ##          [3, 8],
                                          ##          [4, 9]])

z3 = w.reshape(-1)
print(z3)                                ## Always works because reshape() returns a new tensor    ## (M)
                                          ## tensor([0, 5, 1, 6, 2, 7, 3, 8, 4, 9])              ## (N)

#z4 = w.view(-1)
print(z4)                                ## DOES NOT WORK: RuntimeError: view size is not compatible ## (P)
                                          ## with input tensor's size

```

What is a Contiguous Tensor?

- The call to `view(-1)` in Line (P) of the script on the previous slide does not work because the tensor `w` there is NOT contiguous. In order to explain what that means, I'll start with the 2×5 tensor `x1` created in line (C) of the next script, `reshape2.py`, shown on Slide 64.
- Note the output produced by the statements in Lines (D) and (E) of the next script. Line (D) tells us that the “strides” associated with `x1` are 5 and 1, arranged in the order of the axes, and Line (E) says that `x1` is contiguous. For an array of 2 axes, **contiguous means that the elements of the array are stored in a row-major format** — that is, if you wanted to walk through all the data elements sequentially, first you will see the first row, followed by the second row, and so on. **And the stride value of 5 means that you have to take five steps to go from one row to the next, whereas the stride value of 1 means that, in the same row, you have to take only one step to go to the next column entry.**

What is a Contiguous Tensor? (contd.)

- Lines (H) and (I) show the same properties of the tensor `y` which is the transpose of the tensor `x1`. The tensor `y` is NOT contiguous and its strides are given by (1, 5).
- The tensor `y` is not contiguous because the transpose operation in Line (F) leaves unchanged the storage of the actual data elements that were placed there when `x1` was defined. All that is different for `y` is that its element access logic knows how to convert its row and column index values into where those elements are actually stored in the memory. This is reflected in the stride values for `y` in Line (I). Now jumping from one row to the next requires only a one-step hop. On the other hand, scanning along each row requires making 5-step jumps as you go from one data element to the next.
- As to how `x1` is stored may be referred to as the *natural order* for its elements. So whereas the data in `x1` is stored in its natural order, the data elements in the transpose `y` are not.

What is a Contiguous Tensor? (contd.)

- As shown in Lines (J) and (K), we can flatten `y` by invoking `reshape()` on it. **That is because the contract of `reshape()` alters the underlying storage of the data elements so that they are relocated in the memory in their natural order.**
- On the other hand, **the contract of `view()` does not allow it to automatically alter how the data elements are actually stored in the memory.** So it is unable to flatten `y` as evidenced by the invocation in Line (N) and the resulting error message in Line (O).
- What is interesting is that you are allowed to convert a non-contiguous tensor into a contiguous tensor, as shown by the call in line (P). So even though both `y` and `y2` as returned by the call in Line (P) get displayed as identical tensors, the former is non-contiguous whereas the latter is.
- The tensor `y2` being contiguous, we can now flatten it with a call to `view(-1)` as before, as shown in Line (T).

What is a Contiguous Tensor? (contd.)

```

## reshaping2.py

import torch
torch.manual_seed(0)

x = torch.arange(10)
print(x)                                ## tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])    ## (A)
print("\nIs x contiguous: ", x.is_contiguous())    ## True                                          ## (B)

x1 = x.view(-1,5)
print(x1)                                ## tensor([[0, 1, 2, 3, 4],                    ## (C)
          [5, 6, 7, 8, 9]])

print("\nStride for x1: ", x1.stride())    ## (5, 1)                                       ## (D)
print("\nIs x1 contiguous: ", x1.is_contiguous())    ## True                                          ## (E)

y = x1.T
print(y)                                ## tensor([[0, 5],                             ## (F)
          [1, 6],                             ## (G)
          [2, 7],
          [3, 8],
          [4, 9]])

print("\nIs y contiguous: ", y.is_contiguous())    ## False                                         ## (H)
print("\nStride for y: ", y.stride())            ## (1,5)                                        ## (I)

y1 = y.reshape(-1)
print(y1)                                ## tensor([0, 5, 1, 6, 2, 7, 3, 8, 4, 9])        ## (J)
print("\nIs y1 contiguous: ", y1.is_contiguous())    ## True                                          ## (K)
print("\nStride for y1: ", y1.stride())            ## (1,)                                         ## (L)

```

(Continued on the next slide)

What is a Contiguous Tensor? (contd.)

(..... continued from the previous slide)

```

try:
    y2 = y.view(-1)                                ## (N)
except RuntimeError as e:
    print(e)                                       ## (O)
    y2 = y.contiguous()                           ## (P)
    print(y2)                                     ## (Q)
                                                ## tensor([[0, 5],
                                                ##          [1, 6],
                                                ##          [2, 7],
                                                ##          [3, 8],
                                                ##          [4, 9]])

print("\nIs y2 contiguous: ", y2.is_contiguous()) ## (R)
print("\nStride for y2: ", y2.stride())          ## (S)
                                                ## (2,1)

y3 = y2.view(-1)                                  ## (T)
print(y3)                                         ## tensor([0, 5, 1, 6, 2, 7, 3, 8, 4, 9]) ## (U)
print("\nIs y3 contiguous: ", y3.is_contiguous()) ## (V)
print("\nStride for y3: ", y3.stride())          ## (1,) ## (V)

```

Summary comments on `view()` vs. `reshape()`:

- `view()` is faster because it does not involve fresh memory allocation, but it only works on tensors that are contiguous.
- `reshape()` always works but is more expensive because it involves allocating fresh memory.