Demystifying the Convolutions in PyTorch

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Tuesday 18th February, 2020 10:28

Preamble

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 than 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 whatever assumption you make for dealing with the border effects 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.Convd`:

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

What does that mean?

Outline

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

Outline

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

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 = np.zeros((H-M+1,W-N+1),  dtype=float)
    kernel = np.flip(kernel)
    for i in range(H-M+1):
        for j in range(W-N+1):
            out[i,j] =  np.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 with 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 = np.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.]]
```

- Next we need to define a kernel:

```
ker = np.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 to the input `arr` and to the kernel `ker` returns:

```
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
2. **What About `scipy.signal.convolve2d()` for 2D Convolutions**
3. Input and Kernel Specs for PyTorch's Convolution Function `torch.nn.functional.conv2d()`
4. Squeezing and Unsqueezing the Tensors
5. Using `torch.nn.functional.conv2d()`
6. 2D Convolutions with the PyTorch Class `torch.nn.Conv2d`
7. Verifying That a PyTorch Convolution is in Reality a Cross-Correlation
8. Multi-Channel Convolutions

Using scipy.signal.convolve2d()

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

```
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 valid pixels — and not hallucinated pixels outside the 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 hallucinations at the imagined pixels beyond the border.

Outline

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

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

- This PyTorch function only works on input tensors whose shape corresponds to:

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

- Depending on how we define our input initially, this may call for "repacking" the input tensors as you will soon see.

- Having to convert a numpy representation of the input into a tensor representation on the fly in a custom data loader is not an unlikely scenario. I can easily imagine an application where the multi-dimensional input data comes in the form of numpy arrays. Consider, for example, DL being used on the volumetric data produced by a weather radar (Re: Prof. Robin Tanamachi).

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

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

```
tensor_arr = torch.from_numpy( arr )
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)                                  ##  torch.Size([8, 8])
print(tensor_arr.size())                                 ##  torch.Size([8, 8])
print(type(arr))                                         ##  <type 'numpy.ndarray'>
print(type(arr[0,0]))                                    ##  <type 'numpy.float64'>
```

- As to why the data in the tensor returned by the torch.from_numpy() converter is of type `torch.float64`, note the datatype in the numpy array.

- All of the different tensor data types are listed at:

https://pytorch.org/docs/stable/tensors.html

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.
- In order to get there, let's first convert the 3 × 3 kernel 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 the input and the kernel tensors so that they correspond to the input specifications 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)
```

Repackaging the Input Tensor

- That raises the following very important question: How to reshape the input tensor so that its "axis 0" corresponds to the batch size and its "axis 1" to the number of input channels?
- This is how you do it in PyTorch:

```
tensor_arr = torch.unsqueeze(tsr_arr, 0)

tensor_arr = torch.unsqueeze(tsr_arr, 0)

#                        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])
```

- **This now has the correct shape for the input tensor.**
- But what is the call `torch.unsqueeze(tsr_arr, 0)` actually doing?

Outline

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

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

- Let's say you are thinking out loud and you run into 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 can create cognitive dissonance (which, if not properly dealt with, could result in mental anguish and, eventually, mental-health problems).

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.

- When a data object is of shape $(1, 1, 16, 16)$, 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 XYZ-space, and the last as the same image in the same plane but in the XYZW-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, $(m, 16, 16)$ where the integer stands for the number of channels associated with the image.

- Assume that ours is a grayscale image, implying that $m = 1$. 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_x  =  torch.tensor( [1,2,3,4] )
print(x.shape)                                         ##  torch.Size([4])
```

  into the tensor

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

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

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

- In general, through, 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_x  =  torch.tensor( [1,2,3,4] )
print(tensor_x.shape)                              ##  torch.Size([4])
tensor_y = torch.unsqueeze( tensor_x, 1 )
print(tensor_y)
#                        tensor([[1],
#                                [2],
#                                [3],
#                                [4]])
print(tensor_y.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 of applying `unsqueeze()`. Do you understand the data rearrangement caused by the function?

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

```
tensor_x  =  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_x)
#                            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_x.shape)          ## torch.Size([2, 3, 4])

tensor_y = torch.unsqueeze(tensor_x, 1)

print(tensor_y)
#                            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_y.shape)         ## torch.Size([2, 1, 3, 4])
```

The Function squeeze()

- Function `squeeze()` 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
2. What About `scipy.signal.convolve2d()` for 2D Convolutions
3. Input and Kernel Specs for PyTorch's Convolution Function `torch.nn.functional.conv2d()`
4. Squeezing and Unsqueezing the Tensors
5. **Using `torch.nn.functional.conv2d()`**
6. 2D Convolutions with the PyTorch Class `torch.nn.Conv2d`
7. Verifying That a PyTorch Convolution is in Reality a Cross-Correlation
8. Multi-Channel Convolutions

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

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

```
tsr_ker = torch.unsqueeze(tsr_ker, 0)
tsr_ker = torch.unsqueeze(tsr_ker, 0)
print(tsr_ker)
#                        tensor([[[[-1.,  0.,  1.],
#                                  [-1.,  0.,  1.],
#                                  [-1.,  0.,  1.]]]], dtype=torch.float64)
print(tsr_ker.shape)                                   ## torch.Size([1, 1, 3, 3])
```

- At long last, we are ready to use `torch.nn.functional.convo2d()`:

```
output = torch.nn.functional.conv2d( tsr_arr, tsr_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])
```

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

Outline

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

The Class torch.nn.Conv2d and its Callable Instances

- First of all, note that, unlike what we have dealt with so far, `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 three required arguments for calling the constructor of `torch.nn.Conv2d` are:

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

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

The Class torch.nn.Conv2d and its Callable Instances (contd.)

- A callable instance of torch.nn.Conv2d is subsequently called with an input tensor as its argument. This input tensor must be of the following shape:

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

  where H and W refer to the height and the width of the image array.

- **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)                          ## assumes a stride of 1
```

- In order to become more familiar with `torch.nn.Conv2d`, let's first construct a random input for the callable `conop`:

```
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
```

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
```

- 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 in these slides.

Accessing the Kernel Used for the Convolution

- About the kernel that conop used for the convolution output shown on the previous slide, 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 conop instance:

```
ker = conop.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)
print(ker.type())                                       ## torch.FloatTensor
```

- Note the two special "things" about the weight attribute we printed out: It is of type Parameter and its requires_grad property is set.

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

- Let's apply the `conop` instance to the same deterministic input that you saw earlier on Slide 17. Here is that input:

```
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
```

- We note that the datatype of our input tensor 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(tsr_arr.type())                             ## torch.FloatTensor
```

With Our Own Input and Kernel (contd.)

- Now we specify our kernel:

```
ker = np.array([[-1,0,1],
                [-1,0,1],
                [-1,0,1]])
ker = torch.from_numpy(ker)
print( ker.type() )                                         ## torch.LongTensor
ker = ker.float()
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
2. What About `scipy.signal.convolve2d()` for 2D Convolutions
3. Input and Kernel Specs for PyTorch's Convolution Function `torch.nn.functional.conv2d()`
4. Squeezing and Unsqueezing the Tensors
5. Using `torch.nn.functional.conv2d()`
6. 2D Convolutions with the PyTorch Class `torch.nn.Conv2d`
7. **Verifying That a PyTorch Convolution is in Reality a Cross-Correlation**
8. Multi-Channel Convolutions

Specifying the Input and the Kernel for the Convo 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 in one go by directly calling `torch.zeros()` function.

- Since in PyTorch programming, it is not uncommon to go back and forth between numpy and torch, note that the syntax for `numpy.zeros()` is NOT identical to that for `torch.zeros()`. The former takes a maximum of three args.

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

- And here is the kernel:

```
ker = np.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
2. What About `scipy.signal.convolve2d()` for 2D Convolutions
3. Input and Kernel Specs for PyTorch's Convolution Function `torch.nn.functional.conv2d()`
4. Squeezing and Unsqueezing the Tensors
5. Using `torch.nn.functional.conv2d()`
6. 2D Convolutions with the PyTorch Class `torch.nn.Conv2d`
7. Verifying That a PyTorch Convolution is in Reality a Cross-Correlation
8. **Multi-Channel Convolutions**

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 exists a 1-1 correspondence between the input channels and the output channels?

Specifying a Two-Channel Input

- As you'll recall for the input specification for an instance of `torch.nn.Conv2d`, 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, dtype=float)         # 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
input = input.float()
```

- One of the annoying aspects of PyTorch tensors is that the datatype `float` in a declaration actually means `double`, which gives you 64-bit floats.

- However, with its default settings, the `torch.nn.Conv2d class` likes to do its thing with 32-bit floats. So, you may need to cast the tensor to a 32-bit float and you do that by calling `float()`.

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.]]]])

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

- I intentionally chose these two patterns for the two input channels in order to see how the input channels would contribute to the output channels. You can do that for small (say, 3×3) convolutional operators, as you will see.

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])

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?

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 the output, the two 3×3 operators in the kernel, and 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, let's discretize the output shown on the previous slide:

```
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`

- 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],
#     For the first      |              [-0.1085, -0.1647, -0.2207]],
#     out channel        |
#                        |             [[-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],
#     For the second     |              [ 0.2343,  0.0935,  0.0318]],
#     out channel        |
#                        |             [[ 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],
#     For the third      |              [ 0.2263, -0.1816, -0.0864]],
#     out channel        |
#                        |             [[ 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.

- After descretization in the same manner as before, here is the output for this convolution:

```
output = conop(input)                   ## input is the same as shown on Slide 43
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

- 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 use an integer value for groups 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 a constructor call like

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

  would be illegal.

- In the next slide, we will 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],              ## The 3x3 op for that maps
#                                    [ 0.2117,  0.3165, -0.0241],              ## the first input ch to the
#                                    [-0.2994, -0.1580,  0.2270]]],            ## first output ch
#
#
#                                  [[[-0.0022, -0.1657, -0.2554],              ## The 3x3 op that maps the
#                                    [-0.3120, -0.2813, -0.0676],              ## second input ch to the
#                                    [ 0.1828,  0.1802, -0.3215]]],            ## second output ch
#
#
#                                  [[[ 0.2079, -0.2608, -0.0705],              ## The 3x3 op that maps the
#                                    [-0.1352, -0.0642, -0.0654],              ## third input ch to the
#                                    [-0.2991, -0.2878, -0.0522]]],            ## third output ch
#
#
#                                  [[[ 0.0043, -0.1514,  0.1256],              ## The 3x3 op that maps the
#                                    [-0.3000, -0.0225,  0.2931],              ## fourth input ch to the
#                                    [-0.1360,  0.3010,  0.1207]]]],           ## fourth output ch
                                  requires_grad=True)

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=4`, `out_ch=64`, and `groups=16`, 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, will 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 64 3×3 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 this case, 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 3x3 operators. Each 3x3 operator will take one input channel to all the output channels within each group. That calls for a total of eight 3x3 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_cha, 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
```