Torchvision and Random Tensors

Avinash Kak

Purdue University

Lecture Notes on Deep Learning by Avi Kak and Charles Bouman

Thursday 24th July, 2025 23:11

Preamble

To TOC

The three main issues I address in this lecture are: (1) How to extract pixels from images; (2) How to augment your training data by applying various kinds of transformations to the data you start out with; and (3) How to generate random tensors since that can be a useful thing to do when you are testing your code.

I want to shed some light on extracting pixels from images in order to bring to your attention the fact that there exist multiple representations for images and it is not unlikely that you would use them all in the same deep learning program.

There is, of course, the main tensor based representation in which a single color image is represented by a tensor of shape (C, H, W), where C is the number of channels, H the height of the image and W the width. However, when creating a displayable version of an image tensor, you are likely to go to its numpy representation which is of the format (H, W, C). Now add to the non-uniformity in image representation the fact that what we feed into a neural network is likely to be formatted as (B, C, H, W) where B is the batch size.

Preamble (contd.)

In addition to the two main representations mentioned on the previous slide, there is also the representation that you may bump into if you are engaged in creating your data loader from the ground up: This is the representation when your program first opens an image file as a PIL object. As you will see in this lecture, the array indexing in the PIL representation of an image is different from what it is in the other two representations mentioned on the previous slide.

After you have become familiar with the issues related to image representation, you must come to grips with the fact that pixel values in the images are fundamentally integer data that span the range [0.255]. But what we need for feeding into a neural network is floating-point input data whose range spans $[-1.0, +1.0]$. The `torchvision.transforms` module comes in handy here.

The class `torchvision.transforms.ToTensor` is what most people use to scale the pixel values from [0.255] range to the $[0, +1.0]$ range. Subsequently, the class `torchvision.transforms.Normalize` can be used to transform the range $[0, +1.0]$ to the range $[-1.0, +1.0]$.

Preamble (contd.)

That brings this Preamble to the second major topic addressed in this lecture: data augmentation that has played a huge role in many success stories associated with deep learning.

As to why training data augmentation is important, consider an autonomous vehicle of the future that must learn to recognize the stop signs with almost no error. As explained in this presentation, the learning required for this is not as simple as it may sound. As the vehicle approaches a stop sign, how it shows up in the camera image will undergo significant transformations.

Given a set of training images, you can use the functionality of `torchvision.transforms` to augment your training data in order to address the challenge described above.

In addition to addressing issues related to image representation and data augmentation, a third goal of this lecture is to make you familiar with PyTorch's functionality for generating random data. In particular, I'll focus on how to create tensors with random data drawn from different distributions.

Preamble (contd.)

The syntax examples I'll use with random tensors are also meant to make you familiar with the tensor shapes that are expected by neural networks at their inputs, and with the estimation of different types of histograms for those tensors.

While that brings us to the end of this Preamble, I would not be surprised if at this juncture the following question is nagging some of you at the back of your mind: "**But what exactly is a tensor?**"

In the context of Deep Learning, a tensor is a multidimensional array that is designed to reside in the memory of a GPU. Logically speaking, a tensor is really no different from the numpy.ndarray datatype. However, you cannot directly load a numpy.ndarray object into the memory of a GPU.

A more accurate way of stating the above would be that a tensor created in a CPU is logically similar to its numpy.ndarray counterpart, but can be instantaneously converted into a GPU compatible object by, say, calling cuda() on it. When you do so, its datatype changes to reflect the fact that you have created a "cuda" type object. **Check out the Wikipedia page on "CUDA"**. Although now a word unto itself, "Cuda" started out as an acronym for "Compute Unified Device Architecture."

Outline

Outline

The Nature of Image Data

- Typically, a color image consists of three color planes that are commonly referred to as R, G, and B planes for the primary color components red, green, and blue.
- In the context of deep learning, we refer to the color planes as channels. So a color image consists of three channels. On the other hand, a grayscale image consists of only one channel.
- Again typically, each pixel in each channel is an unsigned 8-bit integer, implying that each pixel in each channel of an image takes on an integer value between 0 and 255, both ends inclusive.
- **What the above says is that image data is fundamentally integer data. Against that reality, consider the fact that, typically, neural networks are designed for floating-point input data.**

The Nature of Image Data (contd.)

- **Even more particularly, modern neural networks expect the inputs to be floating point numbers between -1.0 and 1.0.**
- PyTorch provides two special classes, `tvt.ToTensor` and `tvt.Normalize`, **with callable instances** that can convert integer pixel data into the floating-point values that you can feed into a neural network. In the names of the two classes, I have used the short name `tvt` for the PyTorch module `torchvision.transforms`.
- In addition to the two classes named above, the `torchvision.transforms` module contains several other classes that are useful for what is known as data augmentation. We will consider some of those later in this lecture.

An Image Starts Out as a PIL Object in Python Code

- Every image in a neural network starts its life as a PIL object.
- PIL stands for the **Python Imaging Library**. About its popularity, let me put it this way: **If you are dealing with images in Python, the chances that you are not using PIL are nearly zero**. PIL comes bundled with all major Linux distributions.
- These days when you are importing any PIL functionality in your own Python script, you are likely to be using what is known as the "Pillow" fork of the original PIL library that was created by Fredrik Lundh and others. The Pillow fork was created by Alex Clark and others.

An Image Starts Out as a PIL Object (contd.)

- PIL consists of several classes with different image related functionalities and the standard way to use PIL in your own code is by importing those classes using the "From PIL import xx" syntax. To make it clearer, my Python module for the ICP (Iterative Closest Point) algorithm starts with the following declarations:

```
from PIL import Image
from PIL import ImageFilter
from PIL import ImageFont
from PIL import ImageDraw
from PIL import ImageChops
from PIL import ImageTk
```

- Of the different PIL classes mentioned above, it is the `Image` class that you need in order to create a PIL representation of an image file in our own code. Typically, you would say

```
from PIL import Image

im = Image.open("tulips.jpg")                    ##  name of image file

print(im.format, im.size, im.mode)               ##  JPEG (728, 472) RGB
```

An Image Starts Out as a PIL Object (contd.)

- Shown below is the "`tulips.jpg`" image that was supplied as the arg to the `Image.open()` call on the previous slide:

- Now go back to examining the image attributes printed out by the last line of the code fragment shown on the previous slide. Pay particular attention to the fact that it says that this image is of size 728×472, where obviously 728 pixels is the width (W) of the image and 472 pixels its height (H).

An Image Starts Out as a PIL Object (contd.)

- The size 728×472 returned by PIL's Image class is in the format $W \times H$.
- What's interesting is that there is nothing standard about stating the size of an image in the $W \times H$, as opposed to the $H \times W$. [I believe the $W \times H$ format came into existence when engineers first started creating electronic devices for image display. These devices were based originally on scanning a photo-sensitive display-head with an electron beam from left-to-right and top-to-bottom (from the perspective of a user watching the display). For these displays, an image was considered to be situated in an xy-plane whose origin was at the upper-left, and with positive x-axis from left-to-right and with positive y-axis from top-to-bottom.]
- While the format $W \times H$ for image size has persisted to this day at the device level, for image processing you are more likely to use the $H \times W$ format — simply because an algorithm designer is likely to think of an image as an array (or a matrix if you will) and, as you well know, it has become traditional to represent the size of an array using the format $num\_rows \times num\_cols$, which is the same as $H \times W$.

An Image Starts Out as a PIL Object (contd.)

- It is NOT just the size format difference that is the issue here — **there is also the issue of how the pixel coordinates are expressed in the PIL representation of an image vis-a-vis the array representation likely to be used by algorithm designers.**
- An important difference between the (x, y) coordinates for a pixel in PIL and (i, j) coordinates used in downstream digital image processing is that the first coordinate in the former is for the horizontal position of the pixel, whereas the first coordinate in the latter is for the vertical location of the pixel.
- The next slide presents a (**frequently annoying**) consequence of the above fact that is related to using the `getpixel(x,y)` method of the PIL's `Image` class.

An Annoying Aspect of Pixel Extraction from Images

- For a color image, the `getpixel(x,y)` method of the PIL's `Image` class returns a list of RGB values at the pixel coordinates (x, y). When creating a 3D *array* from the pixels in the three color channels, **we must remember to reverse the coordinate axes for the pixel locations**:

```
im = Image.open("tulips.jpg")                                    ## (A)
W,H = im.size                                                    ## (B)
im_arr = numpy.zeros((H,W,3), dtype=numpy.uint8)                 ## (C)
im_arr[150,100,:] = im.getpixel((100,150))                       ## (D)
print( im_arr[150,100,:] )                  #  [224  135  69]   ## (E)
```

- In the code segment shown above, we want to extract the RGB tuple at a location that is in the 150^{th} row of the image and its 100^{th} column. For that we must invoke the `getpixel(x,y)` function with the arguments $(100, 150)$ as shown.

Another Annoying Aspect of Pixel Extraction

- The numpy array that I declared in the code segment shown on the previous slide in Line (C) used the format (H, W, C) for the image — **because that is how an image is represented in numpy**. "C" stands for the number of channels, which in our case is 3.
- On the other hand, if our goal were to place the pixels directly in a PyTorch tensor, we would use the code segment shown below:

```
im = Image.open("tulips.jpg")                                          ## (F)
W,H = im.size                                                          ## (G)
im_arr = torch.zeros(3, H, W, dtype=torch.uint8)                       ## (H)
im_arr[:, 150,100] = torch.tensor(im.getpixel((100,150)))              ## (I)
print(im_arr[:, 150,100])    #  tensor([224, 135,  69], dtype=torch.uint8) ## (J)
```

- Now we are using the format (C, H, W) in the tensor declaration in Line (H). This is yet another source of annoyance when using PIL, numpy, and tensors, all at the same time. Also note that, as shown in lines (D) on the previous slide and (I) here, the syntax for stuffing the RGB values in a numpy array element is a bit simpler than what it is for the case of a tensor element.

Another Annoying Aspect (contd.)

- If you find yourself becoming unhappy with the annoyances I have listed on the previous slides, take heart from the following: **For the most common examples of programming with PyTorch, you are not likely to run into them. PyTorch does a great job of providing you with high-level constructs that, under the hood, automatically take care of the details I have described on the previous slides.**
- For example, in the PyTorch code for deep learning for image data, you may not see any calls at all to `Image.open()` for converting the images into PIL objects. Those calls would be taken care of by your invocations of the `torchvision.transforms` functionality.
- IMPORTANT: The fact that PyTorch does a great job of burying these annoyances in high-level code constructs does NOT mean that you can simply forget about them. When you display tensors as images and when you want to construct a "movie" from a sequence of tensors, you have to first convert the tensors back into numpy arrays and you run headlong into the issues I have raised in this section.

Outline

Creating Neural-Network-Compatible Tensors from Images

- The goal of this section is to "bridge the gap" between the data as it resides in the PIL representation of an image and the data that you need to feed into a neural network. And, in the process of bridging this gap, a second goal of this section is to highlight the role played by the `torchvision.transforms` module in this transformation.

- For neural processing, the $[0, 255]$ range pixel value integer data as made available by the PIL representation must, at the least, **be scaled down** to the $[0, 1.0]$ interval in the form of floating-point numbers.

- Since, if I were to use real images in my examples, it would be difficult to visually display what exactly happens to the numbers involved when the pixel values undergo the sort of scaling (and other transformations) mentioned above, let's go ahead and create small-sized artificial color "images" for the demonstrations to follow:

```
images = torch.randint(0, 256, (4, 3, 5, 9)).type(torch.uint8)      #(1)
```

Creating Neural-Network-Compatible Tensors from Images (contd.)

- The call in Line (1) on the previous slide populates a tensor of shape $(4, 3, 5, 9)$ that is meant to represent images of height H equal to 5 and width W equal to 9. The number 3 is for C, the number of color channels, and the number 4 is for the size of the batch. **One typically feeds a batch of images into a neural network at one time for reasons that will become clear later in this class.** It is not uncommon for the batch size to be a small number like 4. **The 'default' shape of a tensor that you would typically feed into a neural structure is**

```
(batch_size,  C,  H,  W)
```

- You'll notice that in the statement in Line (1) on the previous slide, I had to cast the output of `torch.randint()` to type `torch.uint8`, which stands for "unsigned 8-bit integer" values. Ordinarily, `torch.randint()` returns 64-bit integers of datatype `torch.int64`. [The datatype `torch.int64` is also known as `torch.long`, after C's 64-bit integer type called `long`. Even if you are using the latest version of PyTorch, the error messages your code generates may use legacy names for these datatypes. For example, I still see the old names `torch.ByteTensor` for `torch.uint8` and `torch.LongTensor` for `torch.int64`.]

Creating Neural-Network-Compatible Tensors from Images (contd.)

- The cast to `torch.uint8` in Line (1) is very important because of the input datatype requirements of image-facing PyTorch functions. **If you don't do the cast as I have shown, some of these functions will not complain but will do the wrong thing, leading to difficult-to-spot bugs in your code**.
- An example of that would be the callable instances created from the commonly used `tvt.ToTensor` for **pixel value scaling**. If you present such a function with the wrong datatype, it will simply pass through the image data without any scaling. For obvious reasons, that could lead to strange behavior from the neural network that would be difficult to troubleshoot.
- In general, though, you are safe if you use "standard" dataloaders that PyTorch provides because the dataloader would do the right thing for you. However, problems of the sort I have mentioned above could arise should there be a need to create your own dataloader from scratch for some new application of deep learning.

Creating Neural-Network-Compatible Tensors from Images (contd.)

- Going back to the statement in Line (1) shown previously, we now have four 5×9 color "images". Let's examine what these look like:

```
## Displaying the first image in the batch:
print(images[0])                                                                    ## (2)

#                tensor([[[172,  47, 117, 192,  67, 251, 195, 103,   9],
#                         [211,  21, 242,  36,  87,  70, 216,  88, 140],
#                         [ 58, 193, 230,  39,  87, 174,  88,  81, 165],
#                         [ 25,  77,  72,   9, 148, 115, 208, 243, 197],
#                         [254,  79, 175, 192,  82,  99, 216, 177, 243]],
#
#                        [[ 29, 147, 147, 142, 167,  32, 193,   9, 185],
#                         [127,  32,  31, 202, 244, 151, 163, 254, 203],
#                         [114, 183,  28,  34, 128, 128, 164,  53, 133],
#                         [ 38, 232, 244,  17,  79, 132, 105,  42, 186],
#                         [ 31, 120,   1,  65, 231, 169,  57,  35, 102]],
#
#                        [[119,  11, 174,  82,  91, 128, 142,  99,  53],
#                         [140, 121, 170,  84, 203,  68,   6, 196,  47],
#                         [127, 244, 131, 204, 100, 180, 232,  78, 143],
#                         [148, 227, 186,  23, 207, 141, 117,  85,  48],
#                         [ 49,  69, 169, 163, 192,  95, 197,  94,   0]]], dtype=torch.uint8)
#

print(images.shape)           # torch.Size([4, 3, 5, 9])                          ## (3)
```

- What you are seeing above is the image tensor `images[0,:,:,:]`, which is the same thing as the tensor `images[0]`, **this being the first image in the batch of 4 images** I created by the code statement in Line (1) in Slide 19.

Creating Neural-Network-Compatible Tensors from Images (contd.)

- In the tensor shape displayed in Line (3) on the previous slide, note its standard format: `(B,C,H,W)`, with B standing for batch size, C for the number of channels, H for the image height, and W for the width.
- As a first step in scaling the pixel values from the $[0, 255]$ range to the $[0, 1]$ range needed for neural-network based processing, let's just consider dividing all the pixel values by the maximum pixel value in our images. In order to carry out pixel scaling in this manner, we need to first find out the max value for the pixels. The code segment that follows shows how to get that information from the `images` tensor:

```
print( images.max() )          #  tensor(255, dtype=torch.uint8)              ## (4)

max_pixel_locations =  (images == images.max()).nonzero()                     ## (5)

print(max_pixel_locations)     # tensor([[2, 1, 3, 3]])                       ## (6)

min_pixel_locations =  (images == images.min()).nonzero()                     ## (7)

print(min_pixel_locations)                                                    ## (8)
                               #  tensor([[0, 2, 4, 8],                      ## (9)
                               #          [1, 0, 1, 6],
                               #          [2, 1, 3, 5],
                               #          [3, 1, 2, 6]])
```

Creating Neural-Network-Compatible Tensors from Images (contd.)

- From the statement in Line (4) on the previous slide, we know that the maximum pixel value in our batch is 255. The code in Lines (5) and (6) returns the locations of the pixels that have the max value. From the output shown in Line (6), there is only one pixel in all of the images of the batch where the pixel value is 255 and the tensor coordinates of that pixel are given by $[2, 1, 3, 3]$. [These index values say that the maximum pixel value occurs in the the batch image of index 2 (which would be the third image), in channel indexed 1 (which would be the Green channel), and at the pixel location $(3, 3)$. On the other hand, the pixels where the values are equal to the minimum value, we have four of them. Each image in the batch has one such pixel. In the first image of the batch, the pixel occurs at location $(4, 8)$ in the Blue channel. And so on.]
- Note the invocation of `nonzero()` in Lines (5) and (7) on the previous slide. If you only want the location of the pixels that have a specific value, you must invoke `nonzero()` as shown.
- Without calling `nonzero()`, what you get in Lines (5) and (7) would be Boolean images as shown on the next slide.

Creating Neural-Network-Compatible Tensors from Images (contd.)

- The Boolean images that you get when you don't also call `nonzero()` in Lines (5) and (7) have their pixels set to `True` or `False` depending on whether or not the predicate in the statements is satisfied.
- This is illustrated by the call shown below, which is the same sort of a call as in Lines (5) and (7), but without the invocation of `nonzero()`. Note also that I am showing only the first Boolean image in order to conserve that space that would otherwise be needed for the entire batch. As you can tell, the min value of 0 does indeed occur at the pixel location $(4, 8)$ in the third channel of the first image in the batch.

```
images_showing_min_pixels =  (images == images.min())                    ## (10)
print(images_showing_min_pixels[0])                                      ## (11)

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

Pixel Value Scaling for Creating Tensors from Images

- Continuing with the thought expressed in the second bullet on Slide 23, we can accomplish **pixel value scaling** by dividing the `images` tensor by the max value as returned by `images.max().float()`. **Note that type conversion to `float` is important since otherwise the purely integer division would yield a 0 at all pixels in the `images` tensor.** Again, the scaled values that are shown below are just for the first image of the batch. If I were to print out the entire `images_scaled` tensor, you will see similar pixel values in the other three images in the batch.

```
images_scaled = images / images.max().float()                                    ## (12)
print(images_scaled[0])                                                          ## (13)

        #tensor([[[0.6745, 0.1843, 0.4588, 0.7529, 0.2627, 0.9843, 0.7647, 0.4039, 0.0353],
        #         [0.8275, 0.0824, 0.9490, 0.1412, 0.3412, 0.2745, 0.8471, 0.3451, 0.5490],
        #         [0.2275, 0.7569, 0.9020, 0.1529, 0.3412, 0.6824, 0.3451, 0.3176, 0.6471],
        #         [0.0980, 0.3020, 0.2824, 0.0353, 0.5804, 0.4510, 0.8157, 0.9529, 0.7725],
        #         [0.9961, 0.3098, 0.6863, 0.7529, 0.3216, 0.3882, 0.8471, 0.6941, 0.9529]],
        #
        #        [[0.1137, 0.5765, 0.5765, 0.5569, 0.6549, 0.1255, 0.7569, 0.0353, 0.7255],
        #         [0.4980, 0.1255, 0.1216, 0.7922, 0.9569, 0.5922, 0.6392, 0.9961, 0.7961],
        #         [0.4471, 0.7176, 0.1098, 0.1333, 0.5020, 0.5020, 0.6431, 0.2078, 0.5216],
        #         [0.1490, 0.9098, 0.9569, 0.0667, 0.3098, 0.5176, 0.4118, 0.1647, 0.7294],
        #         [0.1216, 0.4706, 0.0039, 0.2549, 0.9059, 0.6627, 0.2235, 0.1373, 0.4000]],
        #
        #        [[0.4667, 0.0431, 0.6824, 0.3216, 0.3569, 0.5020, 0.5569, 0.3882, 0.2078],
        #         [0.5490, 0.4745, 0.6667, 0.3294, 0.7961, 0.2667, 0.0235, 0.7686, 0.1843],
        #         [0.4980, 0.9569, 0.5137, 0.8000, 0.3922, 0.7059, 0.9098, 0.3059, 0.5608],
        #         [0.5804, 0.8902, 0.7294, 0.0902, 0.8118, 0.5529, 0.4588, 0.3333, 0.1882],
        #         [0.1922, 0.2706, 0.6627, 0.6392, 0.7529, 0.3725, 0.7725, 0.3686, 0.0000]]])
```

Pixel Value Scaling for Creating Tensors from Images (contd.)

- What I have demonstrated on the previous slide is exactly what is accomplished by the callable instance of the `tvt.ToTensor` class, as shown by the code segment on the next slide. **But note that `tvt.ToTensor` is meant to work directly on a numpy array created from the the PIL representation of an image — provided the array elements are of datatype `torch.uint8`.**

- As I mentioned previously, in numpy, an image is represented by an array of shape (H, W, C) whereas a tensor expects the image arrays to be of shape (C, H, W). Since the `images` tensor we created in Line (1) on Slide 19 is already of shape (C, H, W), we cannot invoke `tvt.ToTensor` directly on it without first reshaping it.

- That leads to the question: How does one reshape a tensor? Later in this class, I'll go more into the subject of reshaping tensors directly, but for now I'll use the functionality provided by the numpy library.

Pixel Value Scaling for Creating Tensors from Images (contd.)

- Keeping in mind what was mentioned on the previous slide, the goal of the code segment shown below is two-fold: **(1)** to show how you can reshape a tensor so that you can supply it as an input to the `tvt.ToTensor`; and **(2)** to point out that the output of `tvt.ToTensor` is exactly the same as what was produced by the division by max pixel value in Line (12) previously on Slide 26.

```
import torchvision.transforms as tvt

images_scaled = torch.zeros_like(images).float()                                    ## (14)
for i in range(images.shape[0]):                                                    ## (15)
    images_scaled[i] = tvt.ToTensor()(numpy.transpose(images[i].numpy(), (1,2,0)))  ## (16)
print(images_scaled[0])                                                             ## (17)

        #Showing the images in batch scaled by image->numpy->torchvision.transforms.ToTensor:
        #tensor([[[0.6745, 0.1843, 0.4588, 0.7529, 0.2627, 0.9843, 0.7647, 0.4039, 0.0353],
        #         [0.8275, 0.0824, 0.9490, 0.1412, 0.3412, 0.2745, 0.8471, 0.3451, 0.5490],
        #         [0.2275, 0.7569, 0.9020, 0.1529, 0.3412, 0.6824, 0.3451, 0.3176, 0.6471],
        #         [0.0980, 0.3020, 0.2824, 0.0353, 0.5804, 0.4510, 0.8157, 0.9529, 0.7725],
        #         [0.9961, 0.3098, 0.6863, 0.7529, 0.3216, 0.3882, 0.8471, 0.6941, 0.9529]],
        #
        #        [[0.1137, 0.5765, 0.5765, 0.5569, 0.6549, 0.1255, 0.7569, 0.0353, 0.7255],
        #         [0.4980, 0.1255, 0.1216, 0.7922, 0.9569, 0.5922, 0.6392, 0.9961, 0.7961],
        #         [0.4471, 0.7176, 0.1098, 0.1333, 0.5020, 0.5020, 0.6431, 0.2078, 0.5216],
        #         [0.1490, 0.9098, 0.9569, 0.0667, 0.3098, 0.5176, 0.4118, 0.1647, 0.7294],
        #         [0.1216, 0.4706, 0.0039, 0.2549, 0.9059, 0.6627, 0.2235, 0.1373, 0.4000]],
        #
        #        [[0.4667, 0.0431, 0.6824, 0.3216, 0.3569, 0.5020, 0.5569, 0.3882, 0.2078],
        #         [0.5490, 0.4745, 0.6667, 0.3294, 0.7961, 0.2667, 0.0235, 0.7686, 0.1843],
        #         [0.4980, 0.9569, 0.5137, 0.8000, 0.3922, 0.7059, 0.9098, 0.3059, 0.5608],
        #         [0.5804, 0.8902, 0.7294, 0.0902, 0.8118, 0.5529, 0.4588, 0.3333, 0.1882],
        #         [0.1922, 0.2706, 0.6627, 0.6392, 0.7529, 0.3725, 0.7725, 0.3686, 0.0000]]])
```

Pixel Value Scaling for Creating Tensors from Images (contd.)

- In the code fragment that is shown in Lines (14) – (17) on the previous slide, we first create a zero tensor of the same shape as the original `images` tensor. **In general, a zero tensor must have its datatype declared, unless you construct it with a `zeros_like()` call as I have done, in which case it gets the same datatype as that of the argument to `zeros_like()`. That fact makes it important to cast the zero tensor in our case to the type `float` since otherwise the scaled values stored in the tensor are likely to become all zeros.**

- Pay close attention to the syntax in Line (16) — you will find it handy especially if you are going to be feeding non-image data into a neural network. We first convert each image in the batch to a numpy array because of the requirements of `tvt.ToTensor`. This part is pretty straightforward, as it will take the $(4, 3, 6, 8)$-shaped tensor into a numpy ndarray of the same shape. **But keep in mind that an image is represented by the format (C, H, W) format in this array. On the other hand, what `tvt.ToTensor` expects is the (H, W, C) format.** Hence the call to `numpy.transpose()`.

Pixel Value Scaling for Creating Tensors from Images (contd.)

- Continuing with the explanation of the code fragment on Slide 28, we iterate through each image in the batch in Lines (14) through (17) because `tvt.ToTensor`'s contract **is to process only one image at a time**.
- The call `tvt.ToTensor()` invokes the constructor of the `tvt.ToTensor()` class to return a callable instance of the class. Since the instance is callable, we can invoke it as a function and supply that function with the two arguments shown in Line (16) in Slide 28. In that line, after converting the i^{th} image in the `images` tensor into a numpy array, we reshape the array from the (C, H, W) shape into a (H, W, C) shape that is expected by `tvt.ToTensor`.
- Note that the scaled pixel values shown on Slide 28 are exactly the same as those produced by direct division by max in Line (12) on Slide 26.

Pixel Value Scaling for Creating Tensors from Images (contd.)

- Shown below is another way to accomplish the same thing. Now we call the constructor of the `tvt.ToPILImage` for a callable instance into which we feed the batch images one at a time. As you would expect, `tvt.ToPILImage` converts each image tensor into its numpy representation, which means reshaping a (C, H, W) array into a (H, W, C). Subsequently, `tvt.ToTensor` does pixel-value scaling and, at the same time, reshapes the result back into a (C, H, W) tensor.

```
images_scaled = torch.zeros_like(images).float()                                    ## (18)
for i in range(images.shape[0]):                                                    ## (19)
    images_scaled[i] = tvt.ToTensor()( tvt.ToPILImage()(images[i]) )                ## (20)
print(images_scaled[0])                                                             ## (21)

        #Showing the first image in batch scaled by image->PIL->torchvision.transforms.ToTensor:
        #tensor([[[0.6745, 0.1843, 0.4588, 0.7529, 0.2627, 0.9843, 0.7647, 0.4039, 0.0353],
        #         [0.8275, 0.0824, 0.9490, 0.1412, 0.3412, 0.2745, 0.8471, 0.3451, 0.5490],
        #         [0.2275, 0.7569, 0.9020, 0.1529, 0.3412, 0.6824, 0.3451, 0.3176, 0.6471],
        #         [0.0980, 0.3020, 0.2824, 0.0353, 0.5804, 0.4510, 0.8157, 0.9529, 0.7725],
        #         [0.9961, 0.3098, 0.6863, 0.7529, 0.3216, 0.3882, 0.8471, 0.6941, 0.9529]],
        #
        #        [[0.1137, 0.5765, 0.5765, 0.5569, 0.6549, 0.1255, 0.7569, 0.0353, 0.7255],
        #         [0.4980, 0.1255, 0.1216, 0.7922, 0.9569, 0.5922, 0.6392, 0.9961, 0.7961],
        #         [0.4471, 0.7176, 0.1098, 0.1333, 0.5020, 0.5020, 0.6431, 0.2078, 0.5216],
        #         [0.1490, 0.9098, 0.9569, 0.0667, 0.3098, 0.5176, 0.4118, 0.1647, 0.7294],
        #         [0.1216, 0.4706, 0.0039, 0.2549, 0.9059, 0.6627, 0.2235, 0.1373, 0.4000]],
        #
        #        [[0.4667, 0.0431, 0.6824, 0.3216, 0.3569, 0.5020, 0.5569, 0.3882, 0.2078],
        #         [0.5490, 0.4745, 0.6667, 0.3294, 0.7961, 0.2667, 0.0235, 0.7686, 0.1843],
        #         [0.4980, 0.9569, 0.5137, 0.8000, 0.3922, 0.7059, 0.9098, 0.3059, 0.5608],
        #         [0.5804, 0.8902, 0.7294, 0.0902, 0.8118, 0.5529, 0.4588, 0.3333, 0.1882],
        #         [0.1922, 0.2706, 0.6627, 0.6392, 0.7529, 0.3725, 0.7725, 0.3686, 0.0000]]])
```

Pixel Value Scaling for Creating Tensors from Images (contd.)

- It is important to bear in mind that pixel-value scaling carried out by `tvt.ToTensor` is on a per-image basis in a batch, as opposed to on a batch basis. The scope of a callable instance of this class is just one multi-channel image, as mentioned previously on Slide 30.

- **After the pixel values have been scaled from the $[0, 255]$ range to the $[0, 1.0]$ range, you are still left with the job of mapping the $[0, 1.0]$ range to what you ideally need for feeding into a neural network — the $[-1.0, +1.0]$ range.**

- If the scaled pixel color values are more or less uniformly distributed over $[0, 1.0]$, the logic of mapping the $[0, 1.0]$ range to the $[-1.0, +1.0]$ range can be quite simple: you subtract 0.5 from each pixel in a tensor and divide the result by 0.5. So a pixel value of 0 will be mapped to -1.0 and a pixel value of 1.0 will remain unchanged at 1.0. **You must do this on a per-channel basis.**

Pixel Value Normalization for Creating Tensors from Images

- Mapping the $[0, 1.0]$ range to the $[-1.0, +1.0]$ range is referred to as a **input data normalization**.
- **On account of its per-channel implementation, the most convenient way to carry out input data normalization is by using a callable instance of** `tvt.Normalize`. **While demonstrating the working of this class, I also want to introduce you to the class** `tvt.Compose` **that acts as a container for all the different possible transformations you may wish to apply to the input images.** Technically speaking, the transformation that is applied to each input image is a composition of all the transformations supplied as list argument to `tvt.Compose`.
- As you would expect based on the above description, the constructor of the class `tvt.Compose` expects a list of the transformation to be applied to the input images.

Pixel Value Normalization for Creating Tensors from Images (contd.)

- The code segment shown below carries out input data normalization with the help of the class `tvt.Normalize`, which is invoked through the container class `tvt.Compose`.
- Note the **two** args, with the first arg being $[0.5, 0.5, 0.5]$ and the second arg also being $[0.5, 0.5, 0.5]$, supplied to the constructor of `tvt.Normalize`. Each number in the first arg is subtracted from the corresponding-channel pixel value in the input image and the result divided by the corresponding number in the second arg. The callable `xform1` is applied to each image in the batch separately. Results shown are for the first image in the batch.

```
xform1 = tvt.Compose([ tvt.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5]) ])                    ## (22)
images_normed = torch.zeros_like(images).float()                                       ## (23)
for i in range(images.shape[0]):                                                       ## (24)
    images_normed[i] = xform1(images_scaled[i])                                        ## (25)
print(images_normed[0])                                                                ## (26)

        #tensor([[[ 0.3490, -0.6314, -0.0824,  0.5059, -0.4745,  0.9686,  0.5294, -0.1922, -0.9294],
        #         [ 0.6549, -0.8353,  0.8980, -0.7176, -0.3176, -0.4510,  0.6941, -0.3098,  0.0980],
        #         [-0.5451,  0.5137,  0.8039, -0.6941, -0.3176,  0.3647, -0.3098, -0.3647,  0.2941],
        #         [-0.8039, -0.3961, -0.4353, -0.9294,  0.1608, -0.0980,  0.6314,  0.9059,  0.5451],
        #         [ 0.9922, -0.3804,  0.3725,  0.5059, -0.3569, -0.2235,  0.6941,  0.3882,  0.9059]],
        #
        #        [[-0.7725,  0.1529,  0.1529,  0.1137,  0.3098, -0.7490,  0.5137, -0.9294,  0.4510],
        #         [-0.0039, -0.7490, -0.7569,  0.5843,  0.9137,  0.1843,  0.2784,  0.9922,  0.5922],
        #         [-0.1059,  0.4353, -0.7804, -0.7333,  0.0039,  0.0039,  0.2863, -0.5843,  0.0431],
        #         [-0.7020,  0.8196,  0.9137, -0.8667, -0.3804,  0.0353, -0.1765, -0.6706,  0.4588],
        #         [-0.7569, -0.0588, -0.9922, -0.4902,  0.8118,  0.3255, -0.5529, -0.7255, -0.2000]],
        #
        #        [[-0.0667, -0.9137,  0.3647, -0.3569, -0.2863,  0.0039,  0.1137, -0.2235, -0.5843],
        #         [ 0.0980, -0.0510,  0.3333, -0.3412,  0.5922, -0.4667, -0.9529,  0.5373, -0.6314],
        #         [-0.0039,  0.9137,  0.0275,  0.6000, -0.2157,  0.4118,  0.8196, -0.3882,  0.1216],
        #         [ 0.1608,  0.7804,  0.4588, -0.8196,  0.6235,  0.1059, -0.0824, -0.3333, -0.6235],
        #         [-0.6157, -0.4588,  0.3255,  0.2784,  0.5059, -0.2549,  0.5451, -0.2627, -1.0000]]])
```

Pixel Value Normalization for Creating Tensors from Images (contd.)

- As you surely noticed by the call in Line (25) on the previous slide, `xform1` was applied to the already scaled images produced by the call to `tvt.ToTensor` shown previously in Line (20) on Slide 31.
- **Shown on the next slide is how you are likely to use `tvt.Compose` in a deep-learning framework:** You supply a list of all the transformation you need to apply to the input data in a form of a list to the constructor of this class. In our case, a reference to the object returned by the constructor is assigned to the variable `xform2`. Now we apply the callable assigned to `xform2` directly on the original data object `images` created previously in Line (1) on Slide 19.
- The callable `xform2` is meant to be applied on a per-image basis, although what its component `Normalize` accomplishes is on a per-channel basis for each image.

Pixel Value Normalization for Creating Tensors from Images (contd.)

- As explained on the previous slide, using the `tvt.Compose` gives you a convenient way to bundle all the transformations you want to carry out on each image in your dataset — **keeping in mind that data normalization is applied to each channel separately**.

- As you would expect, the results shown (for only the first image in the batch) are exactly the same as those displayed on Slide 34.

```
xform2 = tvt.Compose([ tvt.ToPILImage(),                                    ## Return numpy array           ## (27)
                       tvt.ToTensor(),                                      ## For pixel val scaling        ## (28)
                       tvt.Normalize([0.5,0.5,0.5],[0.5,0.5,0.5])])         ## For pixel val normalization  ## (29)
images_normed2 = torch.zeros_like(images).float()                                                        ## (30)
for i in range(images.shape[0]):                                                                         ## (31)
    images_normed2[i] = xform2(images[i])                                                                ## (32)
print(images_normed2[0])

        #tensor([[[ 0.3490, -0.6314, -0.0824,  0.5059, -0.4745,  0.9686,  0.5294, -0.1922, -0.9294],
        #         [ 0.6549, -0.8353,  0.8980, -0.7176, -0.3176, -0.4510,  0.6941, -0.3098,  0.0980],
        #         [-0.5451,  0.5137,  0.8039, -0.6941, -0.3176,  0.3647, -0.3098, -0.3647,  0.2941],
        #         [-0.8039, -0.3961, -0.4353, -0.9294,  0.1608, -0.0980,  0.6314,  0.9059,  0.5451],
        #         [ 0.9922, -0.3804,  0.3725,  0.5059, -0.3569, -0.2235,  0.6941,  0.3882,  0.9059]],
        #
        #        [[-0.7725,  0.1529,  0.1529,  0.1137,  0.3098, -0.7490,  0.5137, -0.9294,  0.4510],
        #         [-0.0039, -0.7490, -0.7569,  0.5843,  0.9137,  0.1843,  0.2784,  0.9922,  0.5922],
        #         [-0.1059,  0.4353, -0.7804, -0.7333,  0.0039,  0.0039,  0.2863, -0.5843,  0.0431],
        #         [-0.7020,  0.8196,  0.9137, -0.8667, -0.3804,  0.0353, -0.1765, -0.6706,  0.4588],
        #         [-0.7569, -0.0588, -0.9922, -0.4902,  0.8118,  0.3255, -0.5529, -0.7255, -0.2000]],
        #
        #        [[-0.0667, -0.9137,  0.3647, -0.3569, -0.2863,  0.0039,  0.1137, -0.2235, -0.5843],
        #         [ 0.0980, -0.0510,  0.3333, -0.3412,  0.5922, -0.4667, -0.9529,  0.5373, -0.6314],
        #         [-0.0039,  0.9137,  0.0275,  0.6000, -0.2157,  0.4118,  0.8196, -0.3882,  0.1216],
        #         [ 0.1608,  0.7804,  0.4588, -0.8196,  0.6235,  0.1059, -0.0824, -0.3333, -0.6235],
        #         [-0.6157, -0.4588,  0.3255,  0.2784,  0.5059, -0.2549,  0.5451, -0.2627, -1.0000]]])
```

Outline

Making the Case for Augmenting the Training Data with Transformations

Let's say you want to create a neural-network (NN) based stop-sign detector for an autonomous vehicle of the future. Here are the challenges for your neural network:

The size scaling issue: A vehicle approaching an intersection would see the stop sign at different scales as far as the size of the object is concerned;

Illumination issues: The NN will have to work despite large variations in the image caused by illumination effects that would depend on the weather conditions and the time of the day and, when the sun is up, on the sun angle;

Viewpoint effects: As the vehicle gets closer to the intersection, its view of the stop sign will be increasingly non-orthogonal. The off-perpendicular views of the stop sign will first suffer from affine distortion and then from projective distortion. The wider a road, the larger these distortions. [Isn't it interesting that human drivers do not even notice such distortions — because of our expectation driven perception of the reality.]

While creating this slide, I was reminded of our own work on neural-network based indoor mobile robot navigation. This work is now 25-years old. **Did you know that Purdue-RVL is considered to be a pioneer in indoor mobile robot navigation research?** Here is a link to that old work:

`https://engineering.purdue.edu/RVL/Publications/Meng93Mobile.pdf`

Augmenting the Training Data

- Deep Learning based frameworks commonly use data augmentation to cope with the sort of challenges listed on the previous slide.
- Data augmentation means that in addition to using the images that you have actually collected, you have also created transformed versions of the images, with the transformations corresponding to the various effects mentioned on the previous slide.
- In the next several slides, I'll talk about what these transformations look like and what functions in `torchvision.transforms` can be invoked for the transformations.

Homographies for Image Transformations

- Image transformations in general are nonlinear. If you point a camera at a picture mounted on a wall (anywhere on the wall, really, and the wall can be at any orientation vis-a-vis you), because of the lens optics in the camera, the relationship between the coordinates of the points in the picture and the coordinates of the corresponding pixels in the camera image is strongly nonlinear.

- Nonlinear transformations are computationally expensive.

- Fortunately, **through the magic of homogeneous coordinates**, these transformations can be expressed as linear relationships in the form of matrix-vector products.

[**NOTE: Did you know that much of the power that modern engineering derives from the use of homogeneous coordinates is based on the work of Shreeram Abhyankar, a great mathematician from Purdue? He passed away in 2012 but his work on algebraic geometry continues to power all kinds of modern applications ranging from Google maps to robotics. Check him out through his Wikipedia page by clicking here:** https://en.wikipedia.org/wiki/Shreeram_Shankar_Abhyankar]

Homogeneous Coordinates

- To introduce you to the concept of homogeneous coordinates, let's go back to you taking a photo of a wall-mounted picture and let's focus for a moment on the relationship between the point coordinates on the wall and the pixel coordinates as the picture is viewed from different angles.
- Let (x, y) represent the coordinates of a point in a wall mounted picture.
- Let (x', y') represent the coordinates of the pixel in the camera image that corresponds to the point (x, y) on the wall.
- **In general, the pixel coordinates (x', y') depend nonlinearly on the point coordinates (x, y).** Even when the imaging plane in the camera is parallel to the wall, this relationship involves a division by the distance between the two planes. **And when the camera optic axis is at an angle with respect to the wall normal, you have an even more complex relationship between the two planes.**

Homogeneous Coordinates (contd.)

- We use homogeneous coordinates (HC) to simplify the relationship between the two planes.
- **With HC, we bring into play one more dimension and represent the same point with three coordinates (x_1, x_2, x_3) where**

$$x = \frac{x_1}{x_3}$$
$$y = \frac{x_2}{x_3}$$

- A 3×3 **homography** is a mapping from the wall plane represented by the coordinates (x, y) to the imaging plane represented by the coordinates (x', y'). It is given by the matrix shown below:

$$\begin{bmatrix} x_1' \\ x_2' \\ x_3' \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

- The above relationship is expressed more compactly as the matrix-vector product $\mathbf{x}' = \mathbf{H}\mathbf{x}$ where $\mathbf{H}$ is the homography.

Homography and its Dependence on the Distance Between the Camera and the Object

- The general form of the homography shown on the previous slide is known as the **Projective Homography**. [When an image is formed by what's known as a pin-hole camera, it may also be called Perspective Homography.]
- The homography matrix **H** acquires a simpler form when our autonomous vehicle of the future is at some distance from the stop sign. This simpler form is known as the **Affine Homography**.
- However, as the vehicle gets closer and closer to the stop sign, the relationship between the plane of the stop sign and camera imaging plane **will change from Affine to Projective**.
- All Affine Homographies are Projective Homographies, but **not** the other way around.
- A distinguishing characteristic of a Projective Homography is that it always maps straight lines into straight lines.

Affine Homography — A Special Cases of the 3×3 Projective Homography

- As mentioned on the previous slide, when the vehicle is at some distance from a stop sign, the relationship between the plane of the stop sign and the camera image plane is likely to be what is known as the Affine Homography.
- A 3×3 homography is Affine if and only if its last row is of the form [0 0 1] as shown below:

$$\begin{bmatrix} x_1' \\ x_2' \\ x_3' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & t_1 \\ a_{21} & a_{22} & t_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

- An Affine Homography always maps straight lines in a scene into straight lines in the image. **Additionally, the parallel lines remain parallel, as shown on the next slide.**

Affine vs. Projective Distortions

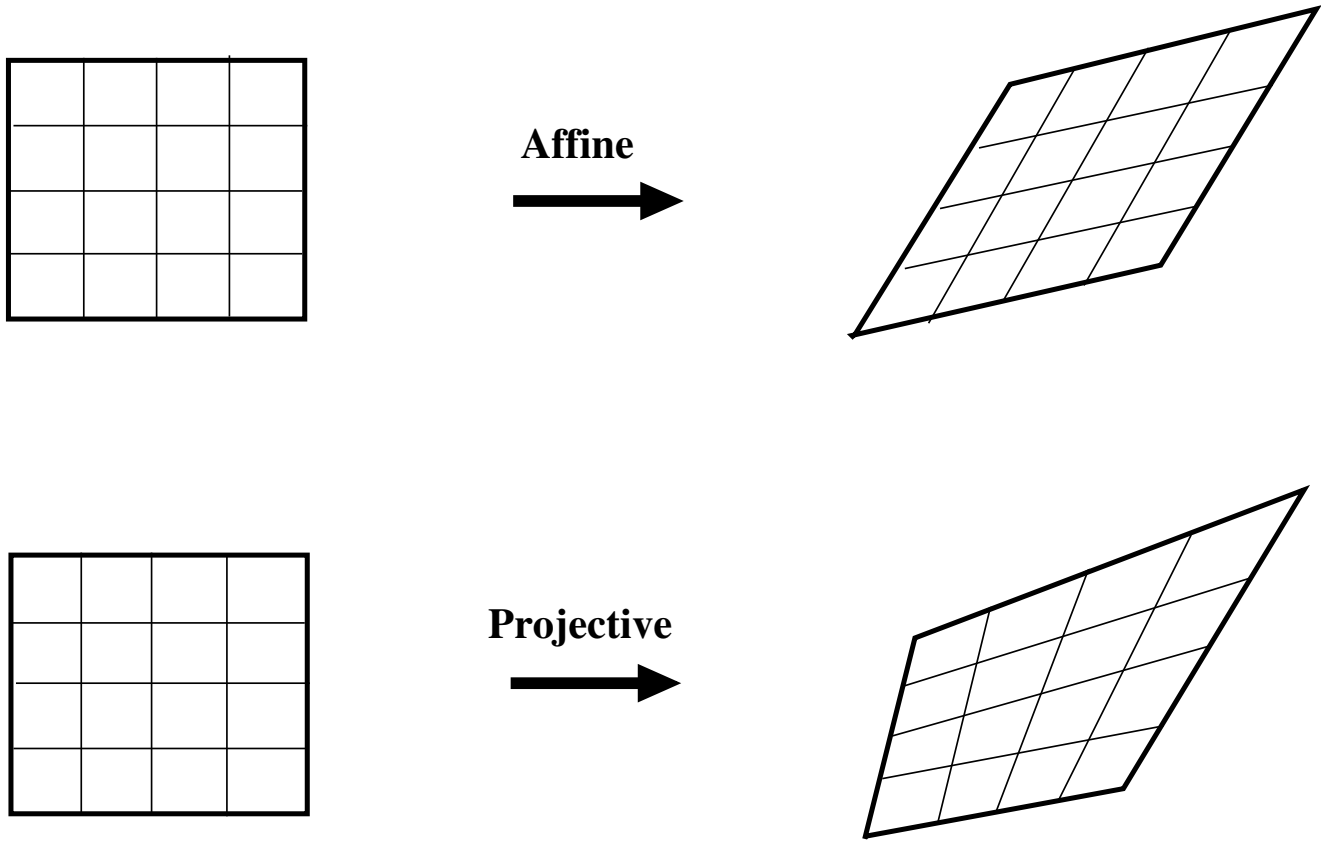

Figure: Affine Homography: Straight lines remain straight and parallel lines remain parallel. Projective Homography: Straight lines remain straight.

Affine Transformation Functionality in

torchvision.transforms

```
class
torchvision.transforms.RandomAffine(degrees, translate=None, scale=None, shear=None, resample=False,
                                                                                    fillcolor=0)

    Random affine transformation of the image keeping center invariant

    Parameters

    degrees (sequence or python:float or python:int) { Range of degrees to select from.
              If degrees is a number instead of sequence like (min, max), the range of
              degrees will be (-degrees, +degrees). Set to 0 to deactivate rotations.

    translate (tuple, optional) { tuple of maximum absolute fraction for horizontal and vertical translations.
              For example translate=(a, b), then horizontal shift is randomly sampled in the range
              -img_width * a < dx < img_width * a and vertical shift is randomly sampled in the range
              -img_height * b < dy < img_height * b. Will not translate by default.

    scale (tuple, optional) { scaling factor interval, e.g (a, b), then scale is randomly sampled
              from the range a <= scale <= b. Will keep original scale by default.

    shear (sequence or python:float or python:int, optional) { Range of degrees to select from. If
              shear is a number, a shear parallel to the x axis in the range (-shear, +shear)
              will be apllied. Else if shear is a tuple or list of 2 values a shear parallel to
              the x axis in the range (shear[0], shear[1]) will be applied. Else if shear is a
              tuple or list of 4 values, a x-axis shear in (shear[0], shear[1]) and y-axis shear
              in (shear[2], shear[3]) will be applied. Will not apply shear by default

     ....
     ....
```

Projective Transformation Functionality in torchvision.transforms

```
torchvision.transforms.functional.perspective(img, startpoints, endpoints, interpolation=3)

   Perform perspective transform of the given PIL Image.

   Parameters

     img (PIL Image) -- Image to be transformed.

     startpoints -- List containing [top-left, top-right, bottom-right, bottom-left] of the orignal image

     endpoints -- List containing [top-left, top-right, bottom-right, bottom-left] of the transformed image

     interpolation -- Default- Image.BICUBIC
```

Outline

Illumination Angle Effects

- While we understand quite well how the camera image of a planar object is likely to change as the camera moves closer to the plane of the object or farther away from it, the dependence of the image on the illumination angle is more complex and data augmentation with respect to these effects still **not** a part of DL frameworks.

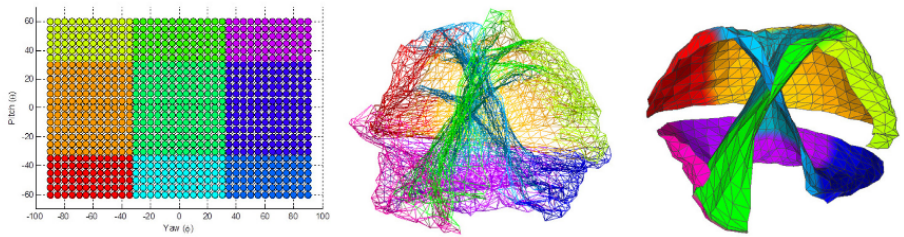

Figure: The face image data as a function of the camera view angle for three different human subjects resides on the three manifolds that are colored differently in the middle depiction. The depiction is based on the first three eigenvectors of the data. The right depiction is the average of the three manifolds

This figure shown above is from our publication:

`https://engineering.purdue.edu/RVL/Publications/FaceRecognitionUnconstrainedPurdueRVL.pdf`

- An application may also require data augmentation with respect to the spectrum of illumination. Regarding how the color content in an image changes with different illumination spectra, see `http://docs.lib.purdue.edu/ecetr/8`

Outline

Some Elementary Image Processing Operations

- You have already seen the basic operations of opening an image file and converting the pixel data into a tensor. Shown below are some additional elementary operations related to: (1) determining the type of the PyTorch object you are creating; (2) accessing the individual color channels; and (3) turning a color image into a grayscale image.
- I have lifted these statements from my `YOLOLogic` module that, by the way, you can download from https://engineering.purdue.edu/kak/distYOLO/ .
- Here are the operations:

```
im_pil = Image.open(image_file)
image_to_tensor = tvt.ToTensor()(im_pil)              ## conversion to tensor with [0,1.0] pixel vals
print(type(image_as_tensor))                          ## <class 'torch.Tensor'>
print(image_as_tensor.type())                         ## <class 'torch.FloatTensor'>
print( image_as_tensor.shape) )                       ## (3, 366, 320)
channel_image = image_as_tensor[n]                    ## returns the color channel indexed n
gray_tensor = 0.4 * image_as_tensor[0] +  0.4 * image_as_tensor[1] + 0.2 * image_as_tensor[2]
```

- Note that what `type(tensor)` returns is not the same as what you get with `tensor.type()`. [The first call uses the Python's built-in function `type()`, and the second invokes the method `type()` on the image tensor.]

Some Elementary Image Processing Operations (contd.)

- An example that shows the kinds of calls you'd need to make for color space transformation, etc.

```
im_pil = Image.open(image_file)

hsv_image = im_pil.convert('HSV')                          ## convert RGB to HSV

hsv_arr = np.asarray(hsv_image)                            ## get hold of the numpy array for image

np.save("hsv_arr.npy", hsv_arr)                            ## save the numy array to a disk file

hsv_image_as_tensor = tvt.ToTensor()( hsv_image )          ## convert the PIL image to a tensor
```

- This code is implemented in the function `working_with_hsv_color_space()` of my `YOLOLogic` module.

Some Elementary Image Processing Operations (contd.)

- An example that shows the kinds of calls you'd need to make for histogramming the individual color channels of the image data:

```
im_pil = Image.open(image_file)
color_image_as_tensor = tvt.ToTensor()( im_pil )                    ## only does pixel val scaling

r_tensor = color_image_as_tensor[0]                                 ## red channel
g_tensor = color_image_as_tensor[1]                                 ## green channel
b_tensor = color_image_as_tensor[2]                                 ## blue channel

hist_r = torch.histc(r_tensor, bins = 10, min = 0.0, max = 1.0)     ## call torch.histc() for constructin
hist_g = torch.histc(g_tensor, bins = 10, min = 0.0, max = 1.0)
hist_b = torch.histc(b_tensor, bins = 10, min = 0.0, max = 1.0)

##  Normalizing the channel based hists so that the bin vals look like probs:
hist_r = hist_r.div(hist_r.sum())
hist_g = hist_g.div(hist_g.sum())
hist_b = hist_b.div(hist_b.sum())
```

- See this code implemented in the function `histogramming_the_image()` of my `YOLOLogic` module.

Outline

Constructing Random Tensors

- As mentioned earlier, augmenting the actually recorded training data by applying to it different types of transformations **with randomized parameters** plays an important role in deep learning. Additionally, **you also need random number generators** for creating "artificial" training and testing datasets for neural networks.

- In this section, though, my goal is merely to make you familiar with the functionality that PyTorch provides for creating random tensors. Here is a link to the documentation page where you will find listed all of PyTorch's functions that you can call for creating tensors with random data based on different probability distributions ((uniform, normal, bernoulli, multinomial, poisson, and many more):

  `https://pytorch.org/docs/stable/torch.html#random-sampling`

- At the above link, you will also see functions whose names end in the substring "`_like`". These can be used to create a tensor whose shape corresponds to that of an existing tensor but whose content is drawn from a random process.

Constructing Random Tensors (Contd.)

- In what follows, I'll focus on just three of the functions listed at the link on the previous slide that I find myself using all the time:
  - `torch.randint()` for uniformly distributed integer data in a tensor;
  - `torch.rand()` for filling a tensor with uniformly distributed floating point values in the half-open interval [0,1), and
  - `torch.randn()` for filling a tensor with normally distributed floating-point numbers with mean 0.0 and variance 1.0.
- In line with what was mentioned on the previous slide, you can also call functions with names like `torch.randint_like()`, `torch.rand_like()`, and `torch.randn_like()` for creating duplicates of previously defined tensors that you now want to fill with random data.
- Shown on the next slide is the basic syntax for calling `torch.randint()`.

Constructing a Random Tensor with `torch.randint()`

- What is shown below is how `torch.randint()` is typically called. The function returns integers that are uniformly distributed between the first arg value and the second arg value minus one. For the call shown below, the output integer values will be between 0 and 15, both ends inclusive. **The third arg is a tuple that specifies the shape of the output tensor.** The call shown below says that the tensor returned by the call has a single axis of dimension 30.

```
out_tensor = torch.randint(0,16,(30,))
print(out_tensor)
                  #  tensor([12, 15,  5,  0,  3, 11,  3,  7,  9,  3,  5,  2,  4,  7,  6,  8,
                  #          8, 12, 10,  1,  6,  7,  7, 14,  8,  1,  5,  9, 13,  8])

print(out_tensor.dtype)                                          #  torch.int64
print(out_tensor.type())                                         #  torch.LongTensor
```

- Note the difference in the answers returned by calling `dtype` on the tensor and doing the same with `type()`. [**The function `dtype` returns the type of the data elements in the container and `type()` returns the type of the container itself, which in our case is a tensor. The information returned by invoking `type()` will also give you a clue as to whether the container is meant for a GPU, as you will see on Slide 59.**]

Using `torch.randint()` (Contd.)

- Pay particular attention to the third argument in the call to `torch.randint()` on the previous slide. As mentioned earlier, that argument is set to a tuple that is the shape of the tensor you want `torch.randint()` to return. In the call shown, I have set that to "(30,)".
- You might think that I would get the same answer as shown on the previous slide if I changed the 3rd argument to "(30,0)". But, for what may seem strange at first sight, here is what would be displayed in your terminal by the three print statements shown on the previous slide if you change the 3rd arg in the first statement to "(30,0)":

```
tensor([], size=(30, 0), dtype=torch.int64)
torch.int64
torch.LongTensor
```

As you can see, you get an empty tensor. WHY? [HINT: Using the analogies of "vectors" and "matrices", a vector is NOT a matrix with its row dimension set to 0. Neither is a vector a matrix with its row dimension set to 1. For the former, setting the dimension to 0 is nonsensical — except, perhaps, for an empty object, and, for the latter, when you set the row dimension equal to 1, you end up with a data object that requires two pairs of square brackets to visualize it. A vector should be displayable with a single pair of square brackets that delimit the data. Roughly the same thing happens with numpy arrays if you make the call `numpy.arange(30).reshape(30,0)`, except that now numpy throws an exception saying "cannot reshape array of size 30 into shape (30,0).]

Using `torch.randint()` (Contd.)

- If you wanted to convert on the fly the data elements of the tensor returned by `torch.randint()` so that they are of type `torch.float`, a 32-bit floating-point datatype, the following call would do that:

```
out_tensor = torch.randint(0,16,(30,)).float()
print(out_tensor)
#  tensor([ 9.,  4.,  3.,  0.,  3.,  5., 14., 15., 15.,  0.,  2.,  3.,  8.,  1.,
#           3., 13.,  3.,  3., 14.,  7.,  0.,  1.,  9.,  9., 15.,  0., 15., 10.,
#           4.,  7.])

print(out_tensor.dtype)                                    #  torch.float32
print(out_tensor.type())                                   #  torch.FloatTensor
```

- The following call shows you can create a tensor in a CPU and then transfer it to the GPU by calling `cuda()` on it while setting its `requires_grad` attribute to true:

```
out_tensor = torch.randint(0,16,(30,)).float().requires_grad_(True).cuda()
print(out_tensor)
#  tensor([ 3., 14., 11.,  2.,  7., 12.,  2.,  0.,  0.,  4.,  5.,  5.,  6.,  8.,
#           4.,  1., 15.,  4.,  9., 10., 10., 15.,  8.,  1.,  1.,  7.,  9.,  9.,
#           3.,  6.], device='cuda:0', grad_fn=<CopyBackwards>)

print(out_tensor.dtype)                                    #  torch.float32
print(out_tensor.type())                                   #  torch.cuda.FloatTensor
```

Using `torch.randint()` (Contd.)

- In the second example on the previous slide, note how the tensor type changed from `torch.FloatTensor` **to** `torch.cuda.FloatTensor`. Such a tensor can only reside in the GPU memory.
- It is important to keep in mind that PyTorch makes a distinction between the CPU based tensors and GPU based tensors. A GPU based tensor comes into existence when a CPU based tensor is moved into the GPU memory by invoking `cuda()` as was shown above.
- The following call makes a GPU tensor directly. We now call `torch.randint()` with a specific argument for the parameter `device`:

```
device = torch.device('cuda'if torch.cuda.is_available() else 'cpu')
out_tensor = torch.randint(0,16,(30,), dtype=torch.float, device=device, requires_grad=True)
print(out_tensor)
#  tensor([ 5.,  1.,  7., 12.,  9., 12.,  7., 13.,  2.,  9., 14.,  5., 14., 12.,
#          14.,  0., 12.,  3., 10., 15.,  8.,  1.,  3.,  3.,  6.,  1.,  8.,  2.,
#          15.,  3.], device='cuda:0', requires_grad=True)
```

Using `torch.randint()` (Contd.)

- Shown below is a call to `torch.randint()` that results in the production of double-precision floats (64 bits) for the data elements in a tensor.

```
out_tensor = torch.randint(0,16,(30,)).double().requires_grad_(True).cuda()
print(out_tensor)
#  tensor([ 7., 11., 14.,  2., 11.,  0., 14.,  3.,  5., 12.,  9., 10.,  4., 11.,
#           4.,  6.,  4., 15., 15.,  4.,  3., 12.,  4.,  4.,  8., 14., 15.,  4.,
#           3., 10.], device='cuda:0', dtype=torch.float64, grad_fn=<CopyBackwards>)
print(out_tensor.dtype)                                          #  torch.float64
print(out_tensor.type())                                         #  torch.cuda.DoubleTensor
```

- All the random number generation examples I have shown so far have been using `torch.randint()`. One can construct similar examples for other two functions listed on Slide 56 taking into account the fact that the calling syntax for the functions is different *since the name of the function implies the range of the values taken by the data elements*, as shown on the next slide.

Random Tensors with `torch.rand()` and `torch.randn()`

- Shown below is a call to `torch.rand()` that returns a 1-axis tensor (think vector) whose elements are drawn from the floating-point values that are **uniformly distributed** over the "[0,1)" interval:

```
out_tensor = torch.rand( (15,) )
print(out_tensor)
#  tensor([0.6833, 0.7529, 0.8579, 0.6870, 0.0051, 0.1757, 0.7497, 0.6047, 0.1100,
#          0.2121, 0.9704, 0.8369, 0.2820, 0.3742, 0.0237])
print(out_tensor.dtype)                                          #  torch.float32
print(out_tensor.type())                                         #  torch.FloatTensor
```

- The call to `torch.randn()` shown below returns a 1-axis tensor whose data elements are drawn from the **Standard Normal Distribution**, meaning from a distribution with **zero mean and unit variance**:

```
out_tensor = torch.randn( (15,) )
print("\n\n7th out_tensor:")
print(out_tensor)
#  tensor([ 1.9956, -0.9683, -0.7803, -0.5713, -0.9645, -1.0204,  1.0309,  2.2084,
#           0.1380,  2.1086, -0.2239, -0.2360, -0.3499, -1.9339,  0.3697])
print(out_tensor.dtype)                                          #  torch.float32
print(out_tensor.type())                                         #  torch.FloatTensor
print(out_tensor.shape)                                          #  torch.Size([15])
```

Random Tensors with `torch.rand()` and `torch.randn()` (contd.)

- The previous slide showed us generating 1-axis tensors. Shown below is a call to `torch.rand()` that returns a 2-axis tensor of shape 3×5 whose floating-points elements are drawn from the **uniform distribution** over the "[0,1)" interval:

```
out_tensor = torch.rand( (3, 5) )
print(out_tensor)
##  tensor([[0.4569, 0.6012, 0.8179, 0.9736, 0.8175],
##          [0.9747, 0.4638, 0.0508, 0.2630, 0.8405],
##          [0.4968, 0.2515, 0.1168, 0.0321, 0.0780]])
print(out_tensor.dtype)                                    #  torch.float32
print(out_tensor.type())                                   #  torch.FloatTensor
print(out_tensor.shape)                                    #  torch.Size([3, 5])
```

- The call to `torch.randn()` shown below returns a 2-axis tensor of shape 3×5 whose data elements are drawn from the **Standard Normal Distribution**:

```
out_tensor = torch.randn( (3, 5) )
print(out_tensor)
#  tensor([[-1.2809, -0.7043, -1.0352, -1.5250, -0.3938],
#          [ 2.1491,  1.6389,  1.2827, -0.5575,  0.1922],
#          [ 0.5196,  0.6670, -0.7107,  0.2771,  1.2262]])
print(out_tensor.dtype)                                    #  torch.float32
print(out_tensor.type())                                   #  torch.FloatTensor
print(out_tensor.shape)                                    #  torch.Size([3, 5])
```

Outline

Tensor Shapes Typically Used for Layer I/0 in CNNs

- Let's become more familiar with the shape of a typical tensor at the input and the output of a typical "layer" in a neural network meant for a computer vision application such as classification, object detection, semantic segmentation, etc. Our immediate goal will be to construct random tensors with the same shape.
- As you will see later, when you are training a CNN, a dataloader (which is likely to be an instance of type `torch.utils.data.DataLoader`), extracts a batch of images from your dataset and presents it as a PyTorch tensor at the input to the neural network you are training. By default, the shape of this tensor is

```
(batch_size, C, H, W)
```

  where `H` and `W` correspond to the "height" and "width" of an image and `C` the number of channels. For an RGB image, the `C` would obviously be 3. The `batch_size` can be any number at all, although typically it is likely to be 4 at the low-end and 128 at the high end.

Constructing Batch-Level Histograms for Uniformly Distributed Data

- I'll now use calls to `torch.randint()` in lines (1) and (2) below to create two different tensors, each of shape $(4, 3, 256, 256)$, for representing two different batches of "RGB images". This shape assumes that the batch-size is 4, that each image consists of 3 channels, and is of size 256×256: The reason for constructing two such batches is that later I will be comparing the two batches on the basis of their histograms.

- The calls to `torch.histc()` in lines (3) and (6) will return a 10-bin histogram over the $(0, 255)$ range of values, both ends inclusive. The division in lines (4) and (7) is for histogram normalization.

```
A = torch.randint(0, 256, (4, 3, 256, 256)).float()           ## batch size: 4, channels: 3, H: 256, W: 256     ## (1)
B = torch.randint(0, 256, (4, 3, 256, 256)).float()                                                           ## (2)

histA = torch.histc(A, bins=10, min=0.0, max=256.0)                                                           ## (3)
histA = histA.div(histA.sum())                                ##  normalized histogram                       ## (4)
print(histA)    ## tensor([0.1014, 0.1013, 0.0979, 0.1016, 0.0977, 0.1016, 0.1020, 0.0970, 0.1016, 0.0979])     ## (5)

histB = torch.histc(B, bins=10, min=0.0, max=256.0)                                                           ## (6)
histB = histB.div(histB.sum())                                ##  normalized histogram                       ## (7)
print(histB)    ## tensor([0.1019, 0.1014, 0.0973, 0.1015, 0.0975, 0.1010, 0.1023, 0.0975, 0.1017, 0.0981])     ## (8)
```

Constructing Batch-Level Histograms for Normally Distributed Data

- This slide shows the same thing as on the previous slide except that now the tensor elements are drawn from the Standard Normal Distribution by making calls to `torch.randn()` in lines (1) and (2).
- Since there are no specific upper and lower bounds on the floating-point values for the random numbers generated from a Gaussian distribution, it is not uncommon to limit any histograms to the interval $[-3\sigma, 3\sigma]$ where σ is the standard deviation. That is what is accomplished in lines (3) and (6) when we set the bounds for the 10-bin histogram requested.

```
A = torch.randn( (4, 3, 256, 256) )                                                                                    ## (1)
B = torch.randn( (4, 3, 256, 256) )                                                                                    ## (2)

histA = torch.histc(A, bins=10, min=-3.0, max=3.0)        ## set [-3\sigma, 3\sigma] bounds for hist; \sigma=1        ## (3)
histA = histA.div(histA.sum())                                                                                         ## (4)
print(histA)   ## tensor([0.0070, 0.0281, 0.0794, 0.1596, 0.2263, 0.2267, 0.1592, 0.0791, 0.0279, 0.0068])          ## (5)

histB = torch.histc(B, bins=10, min=-3.0, max=3.0)        ## set [-3\sigma, 3\sigma] bounds for hist; \sigma=1        ## (6)
histB = histB.div(histB.sum())                                                                                         ## (7)
print(histB)   ## tensor([0.0068, 0.0277, 0.0796, 0.1598, 0.2262, 0.2268, 0.1593, 0.0793, 0.0277, 0.0069])          ## (8)
```

Constructing Per-Channel Histograms for Image Tensors

- The script on the next slide shows us calculating per-channel histograms, that is, a separate histogram for each color channel, for each image in two different batches.
- The four images in the first batch are floating-point values drawn from the standard $N(0, 1)$ distribution as yielded by the call to `torch.randn()`. Its histogram is confined to the interval $[-3, +3]$ of the domain.
- And the four images in the second batch are uniformly distributed over the same $[-3, +3]$ interval. For the second batch, we make call to `torch.rand()`. Since this returns uniformly distributed floats over the interval $[0, 1]$, we must translate and scale the data appropriately in order to cover the $[-3, +3]$ interval.

Constructing Per-Channel Histograms (contd.)

```
batch_size = 4                                                                          ## (1)
img_size = 100                                                                          ## (2)
num_bins  = 10                                                                          ## (3)

A = torch.randn( (batch_size, 3, img_size, img_size) )     # from N(0,1) distro         ## (4)
B = torch.rand( (batch_size, 3, img_size, img_size) )      # from U(0,1) distro         ## (5)
B = B * 6.0 - 3.0                                          # scaling B to cover [-3,3] interval  ## (6)

histTensorA = torch.zeros( batch_size, 3, num_bins, dtype=torch.float )                 ## (7)
histTensorB = torch.zeros( batch_size, 3, num_bins, dtype=torch.float )                 ## (8)

for idx_in_batch in range(A.shape[0]):                                                  ## (9)
    color_channels_A = [A[idx_in_batch, ch] for ch in range(3)]                         ## (10)
    color_channels_B = [B[idx_in_batch, ch] for ch in range(3)]                         ## (11)

    histsA = [torch.histc(color_channels_A[ch],bins=num_bins,min=-3.0,max=3.0) for ch in range(3)]  ## (12)
    histsA = [histsA[ch].div(histsA[ch].sum()) for ch in range(3)]                      ## (13)
    histsB = [torch.histc(color_channels_B[ch],bins=num_bins,min=-3.0,max=3.0) for ch in range(3)]  ## (14)
    histsB = [histsB[ch].div(histsB[ch].sum()) for ch in range(3)]                      ## (15)

    for ch in range(3):                                                                 ## (16)
        histTensorA[idx_in_batch,ch] = histsA[ch]                                       ## (17)
        histTensorB[idx_in_batch,ch] = histsB[ch]

print(histTensorA)                          ## of shape:  (batch_size, 3, num_bins)   ## (18)
print(histTensorB)                          ## of shape:  (batch_size, 3, num_bins)   ## (19)
```

Estimating the Wasserstein Distance Between Two Histograms

- During the last couple of years, the Wasserstein Distance has become very popular in the deep learning community in the context of data modeling with Adversarial Learning.
- Later in this class, when we get to Adversarial Learning I'll explain in detail the notion of Wasserstein Distance. For now, let's be content with calling `wasserstein_distance()` from the `scipy.stats` module.
- The code fragment shown on the next slide compares the respective images in the two separate batches that we synthesized in the previous slide on a channel-by-channel basis by invoking `wasserstein_distance()`.

The Wasserstein Distance Between Channel Histograms (contd.)

- Note that the histograms `histTensorA` and `histTensorB` are the same as calculated on Slide 69.

```
from scipy.stats import wasserstein_distance
for idx_in_batch in range(A.shape[0]):
    print("\n\n\nimage index in batch = %d: " % idx_in_batch)
    for ch in range(3):
        dist = wasserstein_distance( torch.squeeze( histTensorA[idx_in_batch,ch] ).cpu().numpy(),
                                     torch.squeeze( histTensorB[idx_in_batch,ch] ).cpu().numpy() )
        print("\n   Wasserstein distance for channel %d: " % ch, dist)

image index in batch = 0:
   Wasserstein distance for channel 0:  0.07382609848864377
   Wasserstein distance for channel 1:  0.07239626231603323
   Wasserstein distance for channel 2:  0.07252646600827574

image index in batch = 1:
   Wasserstein distance for channel 0:  0.07264069300144911
   Wasserstein distance for channel 1:  0.07206300762481986
   Wasserstein distance for channel 2:  0.07379656429402529

image index in batch = 2:
   Wasserstein distance for channel 0:  0.07414598101750017
   Wasserstein distance for channel 1:  0.07164797377772629
   Wasserstein distance for channel 2:  0.07225345838814974

image index in batch = 3:
   Wasserstein distance for channel 0:  0.07221529143862426
   Wasserstein distance for channel 1:  0.07174333869479596
   Wasserstein distance for channel 2:  0.07192279370501638
```

Obtaining Reproducible Results When Working With Randomized Entities

- As you are tracking down bugs and other errors during the development phase of a system that contains randomized entities, seeing your results vary from one run to another can be an unnecessary and often a confounding distraction.
- With **numpy** based code, in order to make your code produce reproducible results from one run to another, all you had to do was to set the seed of the pseudorandom generator by declaring something like `random.seed(0)` and/or `numpy.random.seed(0)` at the top of your script.
- **With PyTorch, achieving reproducible results is bit more involved because the sources of randomness in the GPU part of code execution are different from the sources of randomness in the CPU part of execution.** You could, for example, have the dataloader doing its randomizations in the CPU while the rest of the code is running in the GPU.

Obtaining Reproducible Results (contd.)

- When I seek reproducibility, I place the following declarations at the top of my script:

```
seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)
```