Torchvision — An Introduction

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Sunday 2nd February, 2020 11:05

Preamble

Deep learning owes its success (as measured by performance evaluation over datasets consisting of millions of images) only partly to the architectures of the networks.

The other reason for that success is data augmentation.

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 imge will undergo significant transformations.

So how can the neural network used by the vehicle be trained so that it would work well despite these transformations? The answer: Through data augmentation of the sort that is presented here.

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. Given its importance, the goal of this lecture is to develop a working level familiarity with the different classes of Torchvision.

Outline

Outline

Making the Case for Transformations for Augmenting the Training Data

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 scale issue: A vehicle approaching an intersection would see the stop sign at different scales;

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 also create 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 on a picture mounted on a wall, because of 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 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.]

Homogeneous Coordinates

- Consider a camera taking a photo of a wall mounted picture. What happens to the relationship between the point coordinates on the wall and the pixel coordinates as you view the picture 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 a more complex relationship between the two planes.

Homogeneous Coordinates (contd.)

- We use homogeneous coordinates (HC) to simplify the relationship between the two planes.
- In 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 $\mathbf{x}' = \mathbf{H}\mathbf{x}$ where $\mathbf{H}$ is the homography.

Homography and the Distance Between the Camera and the Object

- 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.
- The homography shown on the previous slide is the Projective homography. The next slide talks about the Affine Homography.
- A Projective Homography always maps straight lines into straight lines.

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

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

- An affine homography always maps straight lines in a scene into straight lines in the image. **Additionally, the parallel lines remain parallel.**

Affine vs. Projective Distortions

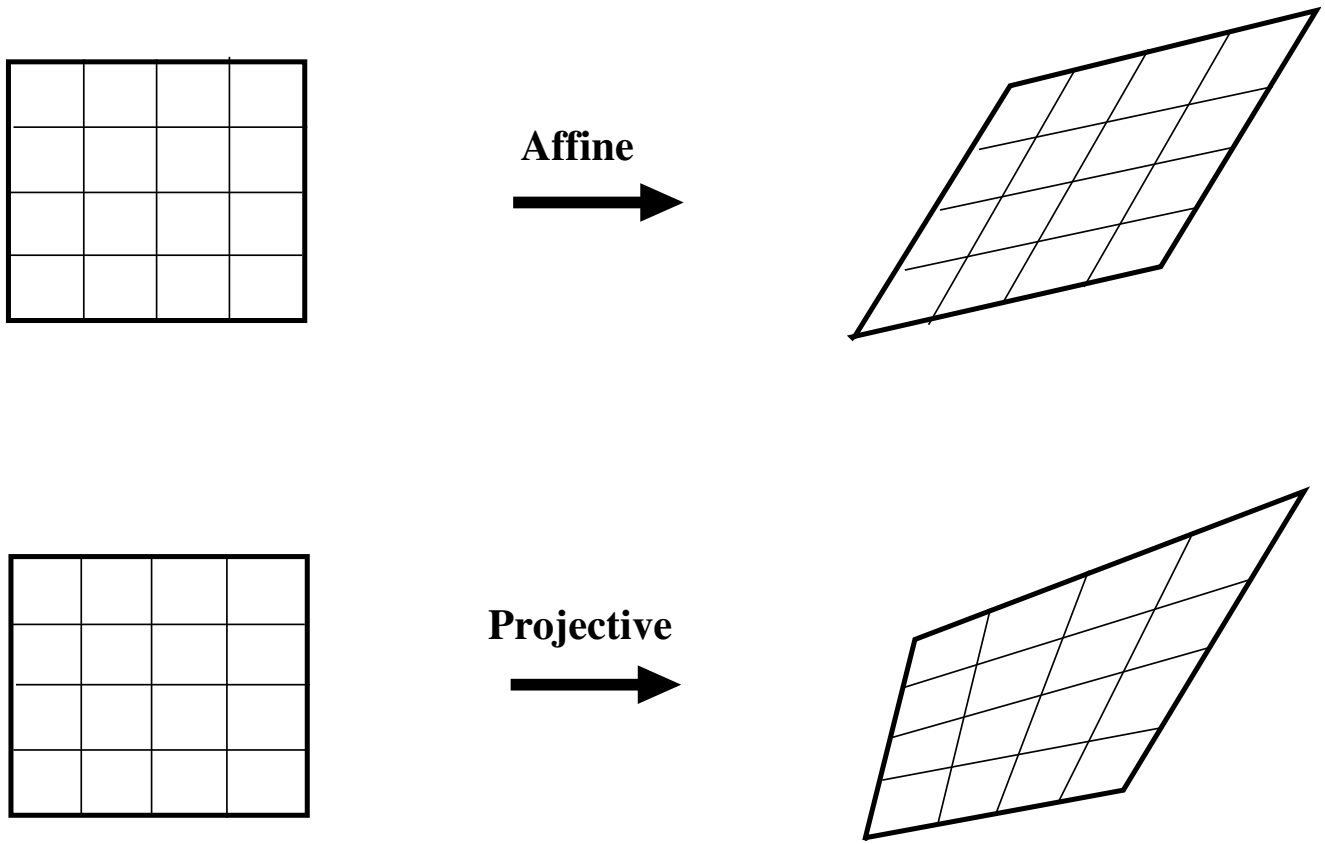

Figure: Affine: Straight lines remain straight and parallel lines remain parallel. Projective: Straight lines remain straight.

Affine Transformation Functionality in torchvision.transforms

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

    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 a camera 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 different kinds of illumination properties is much more complex and data augmentation with respect to these effects still not a part of DL frameworks.

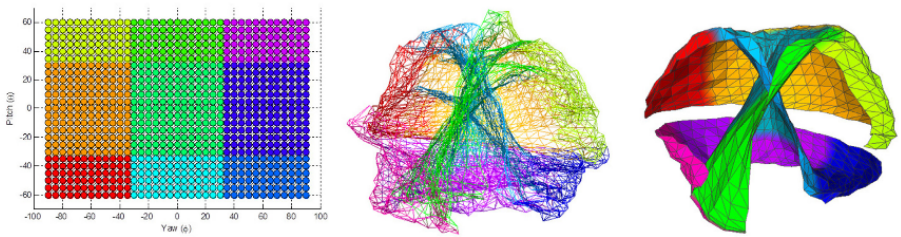

Figure: The face image data for three different human subjects resides on the manifolds shown. The depiction is based on the first three eigenvectors of the data.

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

Greyscale and Color Transformations

- An image is always an array of pixels. In a B&W image, each pixel is represented by an 8-bit integer. What that means is that the grayscale value at each pixel will be an integer value between 0 and 127.
- On the other hand, a pixel in a color image typically has three values associated with it, each an integer between 0 and 127. These three values stand for the three color channels.
- Frequently, the purpose of grayscale and color transformation is to normalize the values so that they possess a specific mean (most frequently zero), a specific range (more frequently -1.0 to 1.0) and a specific standard deviation (more frequently 0.5).

Grayscale and Color Normalization in torchvision.transforms

```
class
torchvision.transforms.Normalize(mean, std, inplace=False)

    Normalize a tensor image with mean and standard deviation. Given mean: (M1,...,Mn) and
    std: (S1,..,Sn) for  n channels, this transform will normalize each channel of the
    input torch.*Tensor i.e. input[channel] = (input[channel] - mean[channel]) / std[channel]

    Parameters
       mean (sequence)  Sequence of means for each channel.
       std (sequence)  Sequence of standard deviations for each channel.
       inplace (bool,optional)  Bool to make this operation in-place.
```

Image Normalization

- Note that when transforming the integer representation in a PIL image into a tensor, all the channel values will be converted from [0,255] ints to [0,1] floats. It is on such a tensor that you would call

```
transforms.Normalize((mean, mean, mean), (std, std, std))])
```

- The call shown above is used most commonly with the following values for the parameters:

```
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

  These parameter values convert each channel values to the (-1,1) range of floats. The normalization of the values in each channel is carried out using the formula:

```
pixel_val = (pixel_val - mean) / std
```

  The parameters (mean, std) are passed as (0.5, 0.5) in the call shown above. This will normalize the image in the range [-1,1]. For example, the minimum value 0 will be converted to (0-0.5)/0.5=-1, the maximum value of 1 will be converted to (1-0.5)/0.5=1.

- In order to take the image back to the PIL form, you would first need to change the range back to [0,1] by $pixel\_val = ((pixel\_val * std) + mean)$

Transforming the Color Space

- Most commonly, the images are based on the RGB color channels. However, that is not always the best strategy.
- Sometimes, the discriminations you want to carry out are better implemented in other color spaces such as HSI/V/L and L*a*b.

Does Torchvision Contain the Transforms for Different Color Spaces

- The answer is: No.
- However, you can use the PIL's `convert()` function for converting an image from one color space to another. For example, if `im` is a PIL image in RGB and you want to convert it into the HSV color space, you would call `x.convert('HSV')`.
- If you want this color-space transformation, you would need to define your color-space transformer class as:

```
def convert_to_hsv(x):
    return x.convert("HSV")
```

But note that "x" has to be a PIL Image object. You can then use the `Lambda` class in `torchvision.transforms` to create your own transform class that will be treated like the native transform classes. This you can do by defining:

```
my_color_xform = torchvision.transforms.Lambda(lambda x: convert_to_hsv(x))
```

Transforming the Color Space (contd.)

- Subsequently, you would want to embed the `my_color_xform` in a randomizer so that it changes the images in each batch in each epoch. That way, the same image would be seen through different color spaces in different epochs.

- This can easily be done by calling on the `RandomApply` class as follows:

```
random_colour_transform = torchvision.transforms.RandomApply([colour_transform], p)
```

  where the second argument, `p`, is set to the probability with which you would want your color transformer to be invoked. Its default value is 0.5.

Transforming the Color Space (contd.)

Here is another data augmentation function from `torchvision.transforms` that can be used to randomly change the brightness, contrast, saturation, and hue levels in an image:

```
class
torchvision.transforms.ColorJitter(brightness=0, contrast=0, saturation=0, hue=0)

    Randomly change the brightness, contrast and saturation of an image.
    Parameters
       brightness (python:float or tuple of python:float (min, max))  How much to jitter
             brightness. brightness_factor is chosen uniformly from [max(0, 1 - brightness),
             1 + brightness] or the given [min, max]. Should be non negative numbers.

       contrast (python:float or tuple of python:float (min, max))  How much to jitter
             contrast. contrast_factor is chosen uniformly from [max(0, 1 - contrast), 1 + contrast]
             or the given [min, max]. Should be non negative numbers.

       saturation (python:float or tuple of python:float (min, max))  How much to jitter saturation.
             saturation_factor is chosen uniformly from [max(0, 1 - saturation), 1 + saturation]
             or the given [min, max]. Should be non negative numbers.

       hue (python:float or tuple of python:float (min, max))  How much to jitter hue.
             hue_factor is chosen uniformly from [-hue, hue] or the given [min, max].
             Should have 0<= hue <= 0.5 or -0.5 <= min <= max <= 0.5.
```

Converting a Tensor Back to a PIL Image

Say that the output of a calculation is a tensor that represents an image. At some point you would want to display the tensor. For that, you have to first convert the tensor to a PIL image. This can be done by calling on the `torchvision.transforms.ToPILImage()` method presented here:

```
class
torchvision.transforms.ToPILImage(mode=None)
    Convert a tensor or an ndarray to PIL Image.
    Converts a torch.*Tensor of shape C x H x W or a numpy ndarray of shape H x W x C to a PIL Image while pres
    Parameters
        mode (PIL.Image mode)
        color space and pixel depth of input data (optional). If mode is None (default) there are some assumpti
                If the input has 4 channels, the mode is assumed to be RGBA.
                If the input has 3 channels, the mode is assumed to be RGB.
                If the input has 2 channels, the mode is assumed to be LA.
                If the input has 1 channel, the mode is determined by the data type (i.e int, float, short).
```

If Color is Important, Why Not Texture?

- Yes, there is much information in texture also, although most published papers on CNN based image recognition do not mention texture at all.
- See the following paper that talks about the role texture has played in the results that researchers have obtained with CNNs:

  `https://arxiv.org/pdf/1811.12231.pdf`
- You might also want to see the following tutorial by me on color and texture: `https://engineering.purdue.edu/kak/Tutorials/TextureAndColor.pdf`

Outline

Using the Compose Class as a Container of Transformations

`torchvision.transforms` provides you with a convenient class called `Compose` that serves as a "container" for all the transformations you want to apply to your images before they are fed into a neural network. Consider the following three different invocations of `Compose`:

```
resize_xform = tvt.Compose( [ tvt.Resize((64,64)) ] )                          ## (A)

gray_and_resize = tvt.Compose( [tvt.Grayscale(num_output_channels = 1),
                                tvt.Resize((64,64)) ] )                        ## (B)

gray_resize_normalize = tvt.Compose( [tvt.Grayscale(num_output_channels = 1),
                                      tvt.Resize((64,64)),
                                      tvt.ToTensor(),
                                      tvt.Normalize(mean=[0.5], std=[0.5]),
                                      tvt.ToPILImage() ] )                     ## (C)
```

What you see on the LHS for each of the three cases is an instance of the `Compose` class — although a callable instance.

Applying the "Composed" Transformations to the Images

The `torchvision.transforms` functionally is designed to work on PIL (Python Image Library) images. So you must first open your disk-stored image file as follows:

```
im_pil = Image.open(image_file)
```

Subsequently, you could invoke any of the following commands:

```
img = resize_xform( im_pil )

img = gray_and_resize( im_pil )

img = gray_resize_normalize( im_pil )
```

depending on what it is that you want to do to the image.

Converting a PIL Image Into a Tensor

Given a PIL image, `img`, you can convert into into a [0,1.0]-range tensor by the following transformation:

```
img_tensor = tvt.Compose([tvt.ToTensor()])
img_data = img_tensor(img)
```

Note again that we must first construct a callable instance of `Compose` and then invoke it on the image that needs to be converted to a tensor. Now we can apply, say, a comparison operator to it as follows:

```
img_data = img_data > 0.5
```

which returns a tensor of booleans (which, for obvious reasons, cannot be displayed directly in the form of an image). To create an image from the booleans, we can use the following ploy:

```
img_data = img_data.float()
to_image_xform = tvt.Compose([tvt.ToPILImage()])
img = to_image_xform(img_data)
```

You can see this code implemented in the function `graying_resizing_binarizing()` of my `RegionProposalGenerator` module.

Outline

Accessing the Color Channels

Shown below are examples of statements you'd need to make if you want to access the individual color channels in an image. I have lifted these statements from the `RegionProposalGenerator` module that, by the way, you can download from

`https://pypi.org/project/RegionProposalGenerator/1.0.4/`

```
im_pil = Image.open(image_file)

image_to_tensor_converter = tvt.ToTensor()                    ## for conversion to [0,1] range
image_as_tensor = image_to_tensor_converter(im_pil)

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]

gray_tensor = 0.4 * image_as_tensor[0]  +   0.4 * image_as_tensor[1]   + 0.2 * image_as_tensor[2]
```

Working with Color Spaces

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')
hsv_arr = np.asarray(hsv_image)
np.save("hsv_arr.npy", hsv_arr)
image_to_tensor_converter = tvt.ToTensor()
hsv_image_as_tensor = image_to_tensor_converter( hsv_image )
```

This code is implemented in the function `working_with_hsv_color_space` of my `RegionProposalGenerator` class.

Histogramming an Image

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

```
im_pil = Image.open(image_file)
image_to_tensor_converter = tvt.ToTensor()
color_image_as_tensor = image_to_tensor_converter( im_pil )

r_tensor = color_image_as_tensor[0]
g_tensor = color_image_as_tensor[1]
b_tensor = color_image_as_tensor[2]

hist_r = torch.histc(r_tensor, bins = 10, min = 0.0, max = 1.0)
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 counts in each sum to 1.
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 `RegionProposalGenerator` module.

Otsu Segmentation of an Image

See the function `histogramming_and_thresholding()` of the `RegionProposalGenerator` module to see how you can carry out figure-ground separation with the Otsu algorithm.