Source code for torchvision.transforms

```python
from __future__ import division
import torch
import math
import random
from PIL import Image, ImageOps
try:
    import accimage
except ImportError:
    accimage = None
import numpy as np
import numbers
import types
import collections


class Compose(object):                                                    [docs]
    """Composes several transforms together.

    Args:
        transforms (list of ``Transform`` objects): list of transforms to compose.

    Example:
        >>> transforms.Compose([
        >>>     transforms.CenterCrop(10),
        >>>     transforms.ToTensor(),
        >>> ])
    """

    def __init__(self, transforms):
        self.transforms = transforms

    def __call__(self, img):
        for t in self.transforms:
            img = t(img)
        return img


class ToTensor(object):                                                   [docs]
    """Convert a ``PIL.Image`` or ``numpy.ndarray`` to tensor.

    Converts a PIL.Image or numpy.ndarray (H x W x C) in the range
    [0, 255] to a torch.FloatTensor of shape (C x H x W) in the range [0.0, 1.0].
    """

    def __call__(self, pic):                                              [docs]
        """
        Args:
            pic (PIL.Image or numpy.ndarray): Image to be converted to tensor.

        Returns:
            Tensor: Converted image.
        """
        if isinstance(pic, np.ndarray):
            # handle numpy array
            img = torch.from_numpy(pic.transpose((2, 0, 1)))
            # backward compatibility
            return img.float().div(255)

        if accimage is not None and isinstance(pic, accimage.Image):
            nppic = np.zeros([pic.channels, pic.height, pic.width], dtype=np.float32)
            pic.copyto(nppic)
            return torch.from_numpy(nppic)

        # handle PIL Image
        if pic.mode == 'I':
```

```
            img = torch.from_numpy(np.array(pic, np.int32, copy=False))
        elif pic.mode == 'I;16':
            img = torch.from_numpy(np.array(pic, np.int16, copy=False))
        else:
            img = torch.ByteTensor(torch.ByteStorage.from_buffer(pic.tobytes()))
        # PIL image mode: 1, L, P, I, F, RGB, YCbCr, RGBA, CMYK
        if pic.mode == 'YCbCr':
            nchannel = 3
        elif pic.mode == 'I;16':
            nchannel = 1
        else:
            nchannel = len(pic.mode)
        img = img.view(pic.size[1], pic.size[0], nchannel)
        # put it from HWC to CHW format
        # yikes, this transpose takes 80% of the loading time/CPU
        img = img.transpose(0, 1).transpose(0, 2).contiguous()
        if isinstance(img, torch.ByteTensor):
            return img.float().div(255)
        else:
            return img


class ToPILImage(object):                                                    [docs]
    """Convert a tensor 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 preserving the value range.
    """

    def __call__(self, pic):                                                 [docs]
        """
        Args:
            pic (Tensor or numpy.ndarray): Image to be converted to PIL.Image.

        Returns:
            PIL.Image: Image converted to PIL.Image.

        """
        npimg = pic
        mode = None
        if isinstance(pic, torch.FloatTensor):
            pic = pic.mul(255).byte()
        if torch.is_tensor(pic):
            npimg = np.transpose(pic.numpy(), (1, 2, 0))
        assert isinstance(npimg, np.ndarray), 'pic should be Tensor or ndarray'
        if npimg.shape[2] == 1:
            npimg = npimg[:, :, 0]

            if npimg.dtype == np.uint8:
                mode = 'L'
            if npimg.dtype == np.int16:
                mode = 'I;16'
            if npimg.dtype == np.int32:
                mode = 'I'
            elif npimg.dtype == np.float32:
                mode = 'F'
        else:
            if npimg.dtype == np.uint8:
                mode = 'RGB'
        assert mode is not None, '{} is not supported'.format(npimg.dtype)
        return Image.fromarray(npimg, mode=mode)


class Normalize(object):                                                     [docs]
    """Normalize an tensor image with mean and standard deviation.
```

```
    Given mean: (R, G, B) and std: (R, G, B),
    will normalize each channel of the torch.*Tensor, i.e.
    channel = (channel - mean) / std

    Args:
        mean (sequence): Sequence of means for R, G, B channels respecitvely.
        std (sequence): Sequence of standard deviations for R, G, B channels
            respecitvely.
    """

    def __init__(self, mean, std):
        self.mean = mean
        self.std = std

    def __call__(self, tensor):                                                    [docs]
        """
        Args:
            tensor (Tensor): Tensor image of size (C, H, W) to be normalized.

        Returns:
            Tensor: Normalized image.
        """
        # TODO: make efficient
        for t, m, s in zip(tensor, self.mean, self.std):
            t.sub_(m).div_(s)
        return tensor


class Scale(object):                                                               [docs]
    """Rescale the input PIL.Image to the given size.

    Args:
        size (sequence or int): Desired output size. If size is a sequence like
            (w, h), output size will be matched to this. If size is an int,
            smaller edge of the image will be matched to this number.
            i.e, if height > width, then image will be rescaled to
            (size * height / width, size)
        interpolation (int, optional): Desired interpolation. Default is
            ``PIL.Image.BILINEAR``
    """

    def __init__(self, size, interpolation=Image.BILINEAR):
        assert isinstance(size, int) or (isinstance(size, collections.Iterable) and len(size) == 2)
        self.size = size
        self.interpolation = interpolation

    def __call__(self, img):
        """
        Args:
            img (PIL.Image): Image to be scaled.

        Returns:
            PIL.Image: Rescaled image.
        """
        if isinstance(self.size, int):
            w, h = img.size
            if (w <= h and w == self.size) or (h <= w and h == self.size):
                return img
            if w < h:
                ow = self.size
                oh = int(self.size * h / w)
                return img.resize((ow, oh), self.interpolation)
            else:
                oh = self.size
                ow = int(self.size * w / h)
```

```
                return img.resize((ow, oh), self.interpolation)
        else:
            return img.resize(self.size, self.interpolation)


class CenterCrop(object):                                                    [docs]
    """Crops the given PIL.Image at the center.

    Args:
        size (sequence or int): Desired output size of the crop. If size is an
            int instead of sequence like (h, w), a square crop (size, size) is
            made.
    """

    def __init__(self, size):
        if isinstance(size, numbers.Number):
            self.size = (int(size), int(size))
        else:
            self.size = size

    def __call__(self, img):
        """
        Args:
            img (PIL.Image): Image to be cropped.

        Returns:
            PIL.Image: Cropped image.
        """
        w, h = img.size
        th, tw = self.size
        x1 = int(round((w - tw) / 2.))
        y1 = int(round((h - th) / 2.))
        return img.crop((x1, y1, x1 + tw, y1 + th))


class Pad(object):                                                           [docs]
    """Pad the given PIL.Image on all sides with the given "pad" value.

    Args:
        padding (int or sequence): Padding on each border. If a sequence of
            length 4, it is used to pad left, top, right and bottom borders respectively.
        fill: Pixel fill value. Default is 0.
    """

    def __init__(self, padding, fill=0):
        assert isinstance(padding, numbers.Number)
        assert isinstance(fill, numbers.Number) or isinstance(fill, str) or isinstance(fill, tuple)
        self.padding = padding
        self.fill = fill

    def __call__(self, img):
        """
        Args:
            img (PIL.Image): Image to be padded.

        Returns:
            PIL.Image: Padded image.
        """
        return ImageOps.expand(img, border=self.padding, fill=self.fill)


class Lambda(object):                                                        [docs]
    """Apply a user-defined lambda as a transform.
```

```
    Args:
        lambd (function): Lambda/function to be used for transform.
    """

    def __init__(self, lambd):
        assert isinstance(lambd, types.LambdaType)
        self.lambd = lambd

    def __call__(self, img):
        return self.lambd(img)


class RandomCrop(object):                                                    [docs]
    """Crop the given PIL.Image at a random location.

    Args:
        size (sequence or int): Desired output size of the crop. If size is an
            int instead of sequence like (h, w), a square crop (size, size) is
            made.
        padding (int or sequence, optional): Optional padding on each border
            of the image. Default is 0, i.e no padding. If a sequence of length
            4 is provided, it is used to pad left, top, right, bottom borders
            respectively.
    """

    def __init__(self, size, padding=0):
        if isinstance(size, numbers.Number):
            self.size = (int(size), int(size))
        else:
            self.size = size
        self.padding = padding

    def __call__(self, img):
        """
        Args:
            img (PIL.Image): Image to be cropped.

        Returns:
            PIL.Image: Cropped image.
        """
        if self.padding > 0:
            img = ImageOps.expand(img, border=self.padding, fill=0)

        w, h = img.size
        th, tw = self.size
        if w == tw and h == th:
            return img

        x1 = random.randint(0, w - tw)
        y1 = random.randint(0, h - th)
        return img.crop((x1, y1, x1 + tw, y1 + th))


class RandomHorizontalFlip(object):                                          [docs]
    """Horizontally flip the given PIL.Image randomly with a probability of 0.5."""

    def __call__(self, img):
        """
        Args:
            img (PIL.Image): Image to be flipped.

        Returns:
            PIL.Image: Randomly flipped image.
        """
        if random.random() < 0.5:
```

```
            return img.transpose(Image.FLIP_LEFT_RIGHT)
        return img


class RandomSizedCrop(object):                                                  [docs]
    """Crop the given PIL.Image to random size and aspect ratio.

    A crop of random size of (0.08 to 1.0) of the original size and a random
    aspect ratio of 3/4 to 4/3 of the original aspect ratio is made. This crop
    is finally resized to given size.
    This is popularly used to train the Inception networks.

    Args:
        size: size of the smaller edge
        interpolation: Default: PIL.Image.BILINEAR
    """

    def __init__(self, size, interpolation=Image.BILINEAR):
        self.size = size
        self.interpolation = interpolation

    def __call__(self, img):
        for attempt in range(10):
            area = img.size[0] * img.size[1]
            target_area = random.uniform(0.08, 1.0) * area
            aspect_ratio = random.uniform(3. / 4, 4. / 3)

            w = int(round(math.sqrt(target_area * aspect_ratio)))
            h = int(round(math.sqrt(target_area / aspect_ratio)))

            if random.random() < 0.5:
                w, h = h, w

            if w <= img.size[0] and h <= img.size[1]:
                x1 = random.randint(0, img.size[0] - w)
                y1 = random.randint(0, img.size[1] - h)

                img = img.crop((x1, y1, x1 + w, y1 + h))
                assert(img.size == (w, h))

                return img.resize((self.size, self.size), self.interpolation)

        # Fallback
        scale = Scale(self.size, interpolation=self.interpolation)
        crop = CenterCrop(self.size)
        return crop(scale(img))
```