Data Augmentation with PyTorch: Image Segmentation

Augmenting imagery for segmenting tasks in PyTorch is more nuanced than for categorizing tasks because it requires the enhancement of both the primary image and its corresponding segmentation overlay. This is achieved by creating a dataset class that ingests two distinct directories: one for the primary images and another for the segmentation overlays. Correspondence between files is crucial, either through identical naming or identical alphabetical sequencing. The dataset constructor is designed to incorporate two distinct augmentation processes, one each for the primary image and the overlay. This distinction is essential as some augmentations, like color alterations, are inappropriate for overlays. Additionally, the system ensures that any spatial modifications to the primary image are mirrored in the overlay, achieved by synchronizing the random number generators for both processes. An extra parameter is also available for integrating additional seed-setting functions, potentially from other libraries.

```
!pip3 install 'torch==0.4.0'
!pip3 install 'torchvision==0.2.1'
!pip3 install --no-cache-dir -I 'pillow==5.1.0'
!pip install git+https://github.com/aleju/imgaug
import numpy as np
from skimage.io import imshow, imread
import torchvision
from torchvision import transforms
from torchvision.utils import make_grid

import matplotlib.pyplot as plt
import matplotlib as mpl
%matplotlib inline
mpl.rcParams['axes.grid'] = False
mpl.rcParams['image.interpolation'] = 'nearest'
mpl.rcParams['figure.figsize'] = 15, 10

def show(img):
    npimg = img.numpy()
    plt.figure()
    plt.imshow(np.transpose(npimg, (1, 2, 0)), interpolation='nearest')
import os
import random

import torch
import torch.utils.data as data
```

```
from PIL import Image

class SegmentationDataset(data.Dataset):
    IMG_EXTENSIONS = ['.jpg', '.jpeg', '.png', '.ppm', '.bmp', '.pgm',
'.tif']

    @staticmethod
    def _isimage(image, ends):
        return any(image.endswith(end) for end in ends)

    @staticmethod
    def _load_input_image(path):
        with open(path, 'rb') as f:
            img = Image.open(f)
            return img.convert('RGB')

    @staticmethod
    def _load_target_image(path):
        with open(path, 'rb') as f:
            img = Image.open(f)
            return img.convert('L')

    def __init__(self, input_root, target_root, transform_input=None,
                 transform_target=None, seed_fn=None):
        assert bool(transform_input) == bool(transform_target)

        self.input_root = input_root
        self.target_root = target_root
        self.transform_input = transform_input
        self.transform_target = transform_target
        self.seed_fn = seed_fn

        self.input_ids = sorted(img for img in os.listdir(self.input_root)
                                if self._isimage(img, self.IMG_EXTENSIONS))

        self.target_ids = sorted(img for img in
os.listdir(self.target_root)
                                 if self._isimage(img,
self.IMG_EXTENSIONS))

        assert(len(self.input_ids) == len(self.target_ids))

    def _set_seed(self, seed):
```

```
        random.seed(seed)
        torch.manual_seed(seed)
        if self.seed_fn:
            self.seed_fn(seed)

    def __getitem__(self, idx):
        input_img = self._load_input_image(
            os.path.join(self.input_root, self.input_ids[idx]))
        target_img = self._load_target_image(
            os.path.join(self.target_root, self.target_ids[idx]))

        if self.transform_input:
            seed = random.randint(0, 2**32)
            self._set_seed(seed)
            input_img = self.transform_input(input_img)
            self._set_seed(seed)
            target_img = self.transform_target(target_img)

        return input_img, target_img, self.input_ids[idx]

    def __len__(self):
        return len(self.input_ids)
```

Let's try with the default torchvision augmentations.

It's important to prioritize geometric augmentations as they are the initial transformations utilizing the seed for the Random Number Generator (RNG). If the augmentation sequence is altered in the input image, it results in a divergence in the random numbers for subsequent augmentations. This discrepancy leads to the creation of a segmentation mask that doesn't align with the modified input image.

```
geometric_augs = [
    transforms.RandomResizedCrop(299),
    transforms.RandomRotation(45),
]

color_augs = [
    transforms.ColorJitter(hue=0.05, saturation=0.4)
]
```

```
def make_tfs(augs):
    return transforms.Compose(augs + [transforms.ToTensor()])

tfs = transforms.Compose(geometric_augs)

ds = SegmentationDataset('../data/segmentation/input/',
'../data/segmentation/masks/',
                         transform_input=make_tfs(geometric_augs +
color_augs),
                         transform_target=make_tfs(geometric_augs))
imgs = [ds[i] for i in range(6)]

show(torchvision.utils.make_grid(torch.stack([img[0] for img in imgs])))
show(torchvision.utils.make_grid(torch.stack([img[1] for img in imgs])))
```

Now let's use an external library: imgaug.

We need to pass a function that sets the seed for imgaug to the seed_fn argument.

```
from imgaug import augmenters as iaa
import imgaug as ia
geometric_augs = [
        iaa.Scale((299, 299)),
        iaa.Fliplr(0.5),
        iaa.Affine(rotate=(-45, 45),
                   translate_percent={"x": (-0.2, 0.2), "y": (-0.2, 0.2)}),
]
color_augs = [
    iaa.AddToHueAndSaturation((-10, 10))
]

def iaug_to_pytorch(augs):
    return transforms.Compose([
        lambda x: np.array(x),
        lambda x: iaa.Sequential(augs).augment_image(x),
        lambda x: Image.fromarray(x),
        transforms.ToTensor(),
    ])

ds2 = SegmentationDataset('../data/segmentation/input/',
'../data/segmentation/masks/',
```

```
                           transform_input=iaug_to_pytorch(geometric_augs +
color_augs),
                           transform_target=iaug_to_pytorch(geometric_augs),
                           seed_fn=lambda x: ia.seed(x % 2**32))
imgs = [ds2[i] for i in range(6)]

show(torchvision.utils.make_grid(torch.stack([img[0] for img in imgs])))
show(torchvision.utils.make_grid(torch.stack([img[1] for img in imgs])))
```

The approach appears effective, yet it's crucial to meticulously sequence the transformations.

We'll adapt the SegmentationDataset class to exclusively accept augmentations from imgaug.

For this, images will be loaded using skimage.io.imread, as imgaug is compatible with numpy arrays.

To selectively deactivate certain augmentations for the segmentation mask, we'll employ hooks (as detailed previously). We'll establish a single transformation parameter and an extra 'input_only' parameter, which will list the names of augmentors to be used solely on input images.

```
class SegmentationDatasetImgaug(data.Dataset):
    IMG_EXTENSIONS = ['.jpg', '.jpeg', '.png', '.ppm', '.bmp', '.pgm',
'.tif']

    @staticmethod
    def _isimage(image, ends):
        return any(image.endswith(end) for end in ends)

    @staticmethod
    def _load_input_image(path):
        return imread(path)

    @staticmethod
    def _load_target_image(path):
        return imread(path, as_gray=True)[..., np.newaxis]

    def __init__(self, input_root, target_root, transform=None,
input_only=None):
        self.input_root = input_root
```

```
        self.target_root = target_root
        self.transform = transform
        self.input_only = input_only

        self.input_ids = sorted(img for img in os.listdir(self.input_root)
                                if self._isimage(img, self.IMG_EXTENSIONS))

        self.target_ids = sorted(img for img in
os.listdir(self.target_root)
                                 if self._isimage(img,
self.IMG_EXTENSIONS))

        assert(len(self.input_ids) == len(self.target_ids))

    def _activator_masks(self, images, augmenter, parents, default):
        if self.input_only and augmenter.name in self.input_only:
            return False
        else:
            return default

    def __getitem__(self, idx):
        input_img = self._load_input_image(
            os.path.join(self.input_root, self.input_ids[idx]))
        target_img = self._load_target_image(
            os.path.join(self.target_root, self.target_ids[idx]))

        if self.transform:
            det_tf = self.transform.to_deterministic()
            input_img = det_tf.augment_image(input_img)
            target_img = det_tf.augment_image(
                target_img,
                hooks=ia.HooksImages(activator=self._activator_masks))

        to_tensor = transforms.ToTensor()
        input_img = to_tensor(input_img)
        target_img = to_tensor(target_img)

        return input_img, target_img, self.input_ids[idx]

    def __len__(self):
        return len(self.input_ids)
augs = iaa.Sequential([
    iaa.Scale((299, 299)),
```

```
    iaa.Fliplr(0.5),
    iaa.Affine(rotate=(-45, 45),
               translate_percent={"x": (-0.2, 0.2), "y": (-0.2, 0.2)}),
    iaa.Add((-40, 40), per_channel=0.5, name="color-jitter")
])


ds3 = SegmentationDatasetImgaug(
    '../data/segmentation/input/', '../data/segmentation/masks/',
    transform=augs,
    input_only=['color-jitter']
)
imgs = [ds3[i] for i in range(6)]

show(torchvision.utils.make_grid(torch.stack([img[0] for img in imgs])))
show(torchvision.utils.make_grid(torch.stack([img[1] for img in imgs])))
```

This solution is more elegant than the previous ones. We can now add different augmentations and apply some of them to the input image only.