Exercise 1 - Binarization

General remarks

- Pay attention to correctly calculate the metrics used in the paper by Su et al. (see Lecture 3: Slides 46 - 48)
- You can use a library for a sanity check of your results, e.g. DoxaPy (~ 18.5 dB PSNR)

U-Net Binarization

Training

- Train a basic U-Net with one class on patches (e.g. 256x256). The final image is obtained by thresholding or a proper activation function (e.g. Sigmoid)
- Use data augmentation such as vertical/horizontal flips, gaussian blur or color jitter, e.g.

```
if random.random() > 0.5:
        img = transforms.functional.hflip(img)
        mask = transforms.functional.hflip(mask)
if random.random() > 0.5:
        img = transforms.functional.vflip(img)
        mask = transforms.functional.vflip(mask)
if random.random() > 0.5:
        gaussian = transforms.GaussianBlur(3, sigma=(0.1, 2.0))
        img = gaussian(img)

augmentation_pipeline = torchvision.transforms.Compose([
         transforms.ColorJitter(brightness=0.25, contrast=0.25, saturation=0.25, hue = 0.25),
         transforms.ToTensor()
])
```

- Since the images included in DIBCO2009 are grayscale, it is also worth-trying to only train on grayscale images.
- Use as much data available as possible, e.g. DIBCO 2010,2011,2012,2017 might be a good starting point.
- We recommend to combine a segmentation-based loss, e.g. Dice loss and a pixel-based loss, e.g. BCELoss

```
loss = bceloss(pred, mask) + dice_loss(mask, pred)
```

Evaluation

- Although training is done on patch-level, the metrics should be calculated on the full images. Therefore, you need to implement a split and merge algorithm
  1. Split the images in NxN patches
  2. U-Net inference
  3. Merge the patches
  4. Metrics calculation
- Useful - but not the most efficient :) - code might be:

```
# Calculate the new dimensions that are divisible by 48
pad_H = patch_size - (H % patch_size)
pad_W = patch_size - (W % patch_size)

img = torch.nn.functional.pad(img, (0, pad_W, 0, pad_H), mode='constant', value=1.0)
```

```
# patch extraction and save its height and width ids
 for h_idx in range(0, H+pad_H, patch_size):
        for w_idx in range(0, W+pad_W, patch_size):
            patch = img[:, h_idx:h_idx+patch_size, w_idx:w_idx+patch_size]
                        # ...

### binarize patches via U-Net

# Loop over the patches and insert them into the reconstructed image
for i, (idx, patch) in enumerate(zip(idxs, patches)):
        x1 = idx[0]*patch_size
        x2 = (idx[0]+1)*patch_size
        y1 = idx[1]*patch_size
        y2 = (idx[1]+1)*patch_size
```

- There are also third-party libraries, e.g. Patchify for creating patches of your images.
- It is not necessary to outperform the algorithm by Su et al., but you should provide proper evaluation and report (dis-)advantages of the methods used. Our U-Net achieves a PSNR of about 19.43.