COMPUTER ASSIGNMENT 04

U-net for image segmentation

In class, we talked about U-net for image segmentation. This assignment is intended to

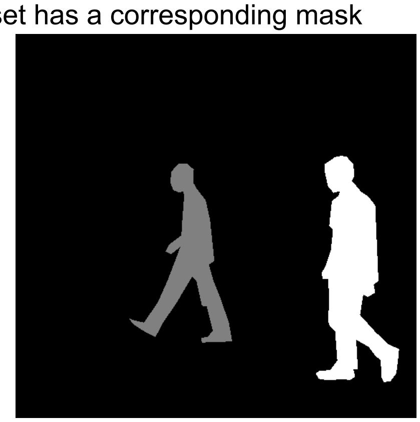
- help you better understand the concept of U-net for image segmentation
- help you get started with designing networks in pytorch including loading data, network design, loss function, training and testing.

For this assignment, you will attempt to segment pedestrians, which is a challenge hosted on Kaggle. You could download from the [Kaggle (https://www.kaggle.com/jiweiliu/pennfudanped?select=readme.txt)](https://www.kaggle.com/jiweiliu/pennfudanped?select=readme.txt) Website.

You should create a folder 'data/' in the current folder and unzip the data into the folder.

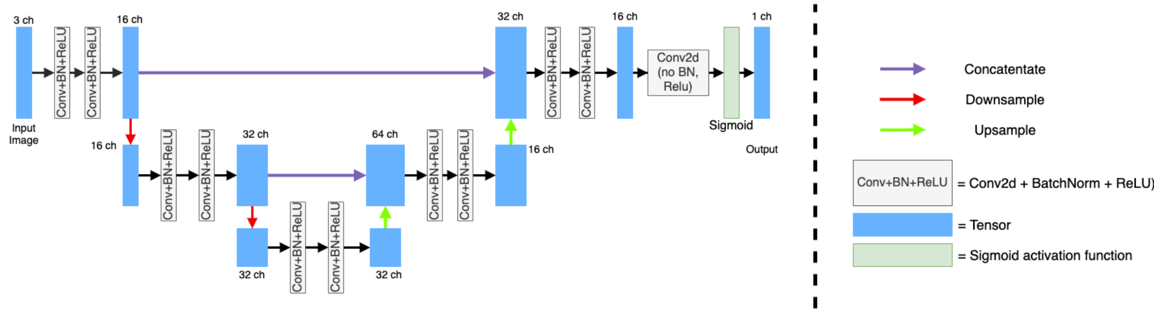
Let's first take a look at the images in the training dataset

Each image in the training dataset has a corresponding mask



You should

- Implement the U-net of the following architechture. Note that for the 2 consecutive Conv+BN+ReLU layers, the output channels of the first layer equals to the output channels of the second.



- Write function dice_coeff(input, target) for evaluation.
- Based on the definition of soft dice, write loss function SoftDICELoss().
- Load training dataset and testing dataset. Notice that you should rescale the images to a smaller size (for example 64x64/96x96/128x128). Otherwise it takes too long to train on cpu.
- Train your network for a few epochs, evaluate your network on validation dataset after each epoch.
- Test your network on testing dataset and report the DICE coefficient of it.
- Plot training loss and validation loss. Comment on the difference between them and when to stop training is appropriate based on your observation.
- Test your model by feeding in a new image in testing dataset. Plot your result of the original image, the mask and the segmented image.

In [907]:
```
import random
import sys
import os
import numpy as np
import matplotlib.pyplot as plt
import glob
import pickle
from tqdm import tqdm
import cv2

%matplotlib inline
```

In [908]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch import optim
from torchvision import transforms

from torch.utils.data import Dataset
```

In [909]:
```
# Use the GPU if you have one
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

print(device)
```

```
cpu
```

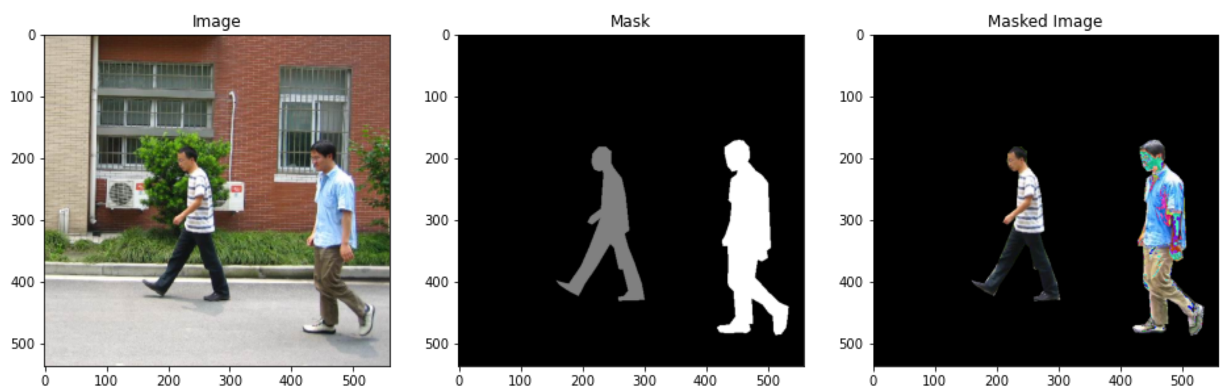
Let's first check what the data is look like!

In [910]:
```
img_list = sorted(glob.glob("./data/PNGImages/*.png"))
label_list = sorted(glob.glob("./data/PedMasks/*.png"))
assert len(img_list) == len(label_list)
print ("Collected {} images".format(len(img_list)))
```

```
Collected 170 images
```

In [911]:
```
img = cv2.imread(img_list[0])
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
mask = cv2.imread(label_list[0],cv2.IMREAD_GRAYSCALE)

plt.figure(figsize = (16,48))
plt.subplot(1,3,1)
plt.imshow(img)
plt.title('Image')
plt.subplot(1,3,2)
plt.imshow(mask,cmap='gray')
plt.title('Mask')
plt.subplot(1,3,3)
plt.imshow((img*mask[:,:,None]))
plt.title('Masked Image')
```

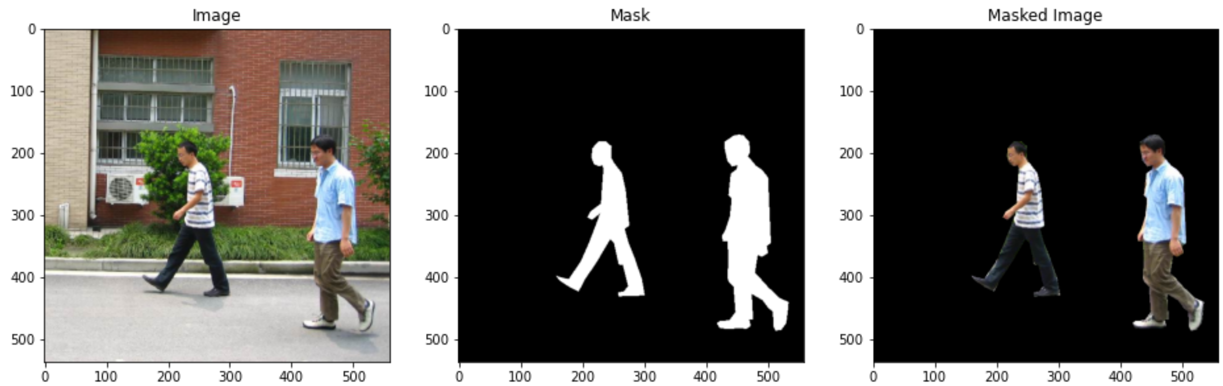
Out[911]:
```
Text(0.5, 1.0, 'Masked Image')
```



Looks a bit weird? This is because the mask is annotated with ID

In [912]:
```
mask = mask>0   # mask = 1 or 0     if mask==1: mask=Ture else: mask=False

plt.figure(figsize = (16,48))
plt.subplot(1,3,1)
plt.imshow(img)
plt.title('Image')
plt.subplot(1,3,2)
plt.imshow(mask,cmap='gray')
plt.title('Mask')
plt.subplot(1,3,3)
plt.imshow((img*mask[:,:,None]))
plt.title('Masked Image')
```

Out[912]:
```
Text(0.5, 1.0, 'Masked Image')
```



[TODO 1] First define following layer to be used later

- **Conv2d + BatchNorm2d + ReLu** as **single_conv layer** ,
- **down layer**: use **Maxpool2d** to downsample by a factor of 2
- **up layer**: takes two inputs of different dimensions. First use **nn.Upsample** to upsample the smaller input to be the same size as the larger, then concatenate the two along the channel dimension
- **outconv layer**: **Conv2d** followed by **sigmoid activation** to generate probability for each pixel

You can check out the documentation in this link to understand how to use the methods called in the provided template:

[https://pytorch.org/docs/stable/nn.html (https://pytorch.org/docs/stable/nn.html)](https://pytorch.org/docs/stable/nn.html)

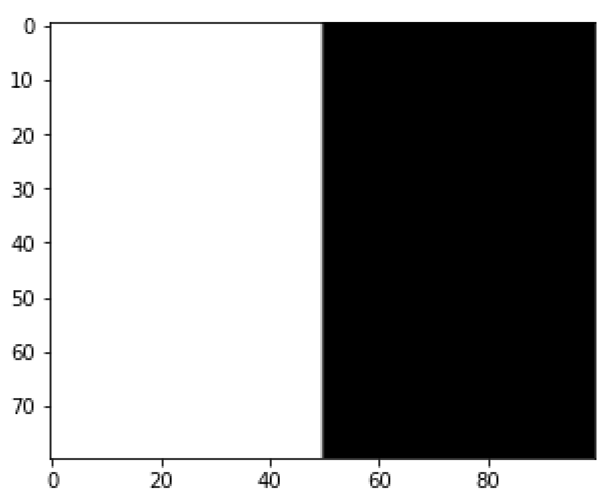
In [913]:
```
############################################### [TODO] ##########################
# DEFINE SINGLE_CONV CLASS
class single_conv(nn.Module):
    '''(conv => BN => ReLU) '''
    def __init__(self, in_ch, out_ch):
        super(single_conv, self).__init__()
        # Define the layers here
        # Note: for conv, use a padding of (1,1) so that size is maintained
        self.conv = nn.Conv2d(in_ch, out_ch, 3, padding=1)
        self.bn = nn.BatchNorm2d(out_ch)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        # define forward operation using the layers above
        x = self.conv(x)
        x = self.bn(x)
        x = self.relu(x)
        return x

############################################### [TODO] ##########################
# DEFINE DOWN CLASS
class down_layer(nn.Module):
    def __init__(self,in_ch, out_ch):
        super(down_layer, self).__init__()
        self.down = nn.MaxPool2d(kernel_size=2, stride=2) # use nn.MaxPool2d( )


    def forward(self, x):
        x = self.down(x)
#         x = self.conv(x)
        return x

############################################### [TODO] ##########################
# DEFINE UP CLASS
# Note that this layer will not only upsample x1, but also concatenate up-sampled

class up_layer(nn.Module):
    def __init__(self,in_ch, out_ch):
        super(up_layer, self).__init__()
        self.up = nn.Upsample(scale_factor=2)  # use nn.Upsample() with mode bili



    # Note: after up, we also concatenate with previously saved feature x2
    def forward(self, x1, x2): # Takes in smaller x1 and larger x2
        # First we upsample x1 to be same size as x2
        x1 = self.up(x1)
        # Now we concatenat x2 and x1 along channel dimension
        # Note pytorch tensor shape correspond to: (batchsize, channel, x_dim, y_
        x = torch.cat((x1,x2),1)

        return x

############################################### [TODO] ##########################
# DEFINE OUTCONV CLASS
class outconv(nn.Module):
```

```
    def __init__(self, in_ch, out_ch):
        super(outconv, self).__init__()
        # 1 conv layer
        self.conv = nn.Conv2d(in_ch, out_ch, 3, padding=1)

    def forward(self, x):
        # Forward conv layer + sigmoid
        x = self.conv(x)
        x = torch.sigmoid(x)
        return x
```

In [914]:
```
################################################ [TODO] #########################
# Build your network with predefined classes: single_conv, down_layer, up_layer,
# The number of input and output channels should follow the U-Net Structure shown
import torch.nn.functional as F
#warnings.warn("nn.functional.sigmoid is deprecated. Use torch.sigmoid instead.")
#import torch.sigmoid as F
class UNet(nn.Module):
    def __init__(self, n_channels_in, n_channels_out):
        super(UNet, self).__init__()
        self.conv1 = single_conv(n_channels_in, 16) # conv2d +  batchnorm + relu
        self.conv2 = single_conv(16,16)

        self.down1 = down_layer(16, 16)         # maxpool2d + conv2d + batchnorm

        self.conv3 = single_conv(16,32)
        self.conv4 = single_conv(32,32)

        self.down2 = down_layer(32, 32)         # maxpool2d + conv2d + batchnorm

        self.conv5 = single_conv(32,32)
        self.conv6 = single_conv(32,32)

        self.up1 = up_layer(32, 64)             # upsample + pad + conv2d + batch

        self.conv7 = single_conv(64,16)
        self.conv8 = single_conv(16,16)

        self.up2 = up_layer(16, 32)             # upsample + pad + conv2d + batch

        self.conv9 = single_conv(32,16)
        self.conv10 = single_conv(16,16)

        self.output = outconv(16, 1)


    def forward(self, x):
        # Define forward pass
        ## Go down to lower dimension
        x1 = self.conv2(self.conv1(x))
        x2 = self.conv4(self.conv3(self.down1(x1)))
        x3 = self.conv6(self.conv5(self.down2(x2)))

        ## Go up back to original dimension
        x = self.conv8(self.conv7(self.up1(x3, x2)))
        x = self.conv10(self.conv9(self.up2(x, x1)))
        x = self.output(x)

        return x

#Test Unet
net = UNet(n_channels_in=3, n_channels_out=1)
net = net.double()
#print(net)
test_net_img=[[0.2]*50+[0.1]*50]*80 #image
test_net_img_batch=np.array([[test_net_img,test_net_img,test_net_img]]) # R,G,B
test_net_img_batch_tensor = torch.from_numpy(test_net_img_batch).type(torch.Doubl
```

```
print(test_net_img_batch_tensor)
plt.imshow(test_net_img_batch[0][0], cmap='gray')
plt.show()
net.train()
print(net(test_net_img_batch_tensor))
```

```
tensor([[[[0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000],
          [0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000],
          [0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000],
          ...,
          [0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000],
          [0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000],
          [0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000]],

         [[0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000],
          [0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000],
          [0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000],
          ...,
          [0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000],
          [0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000],
          [0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000]],

         [[0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000],
          [0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000],
          [0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000],
          ...,
          [0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000],
          [0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000],
          [0.2000, 0.2000, 0.2000,  ..., 0.1000, 0.1000, 0.1000]]]],
       dtype=torch.float64)
```




```
tensor([[[[0.4822, 0.3334, 0.7350,  ..., 0.4441, 0.5492, 0.5531],
          [0.6477, 0.4271, 0.6384,  ..., 0.6293, 0.5716, 0.5211],
          [0.6155, 0.3503, 0.3976,  ..., 0.5724, 0.6051, 0.5059],
          ...,
          [0.5551, 0.5271, 0.5415,  ..., 0.5357, 0.5918, 0.6378],
          [0.4818, 0.4908, 0.5730,  ..., 0.5382, 0.5586, 0.7059],
          [0.5716, 0.3188, 0.4018,  ..., 0.4468, 0.4220, 0.5769]]]],
       dtype=torch.float64, grad_fn=<SigmoidBackward>)
```

[TODO 2] Define evaluation function and loss function:

- **Evaluation function**
  Dice coefficient is defined as
  $$DICE = \frac{2 \times |A \cap B|}{|A| + |B|} = \frac{2 \sum_i^N p_i g_i}{\sum_i^N p_i + \sum_i^N g_i}$$
  For the case of evaluating a Dice coefficient on predicted segmentation masks, we can approximate intersection of A and B as the element-wise multiplication between the prediction and target mask, and then sum the resulting matrix.

  In order to quantify the cardinality of A and B, we can use the simple sum of prediction and target mask.
- **Loss function**
  Soft dice is defined as
  $$SoftDICE = \frac{2 \sum_i^N p_i g_i}{\sum_i^N p_i^2 + \sum_i^N g_i^2}$$
  Here, p_i is the probability value of pixel i, not a binary value, which is different from the one in evaluation function. For the denominator, some researchers use the simple sum whereas other researchers prefer to use the squared sum for this calculation. You can use either way.

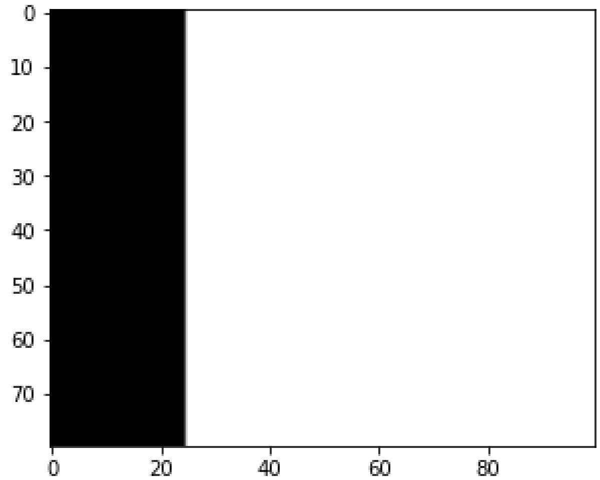
  Remember SoftDICE loss = 1 - SoftDICE = $1 - \frac{2 \sum_i^N p_i g_i}{\sum_i^N p_i^2 + \sum_i^N g_i^2}$.

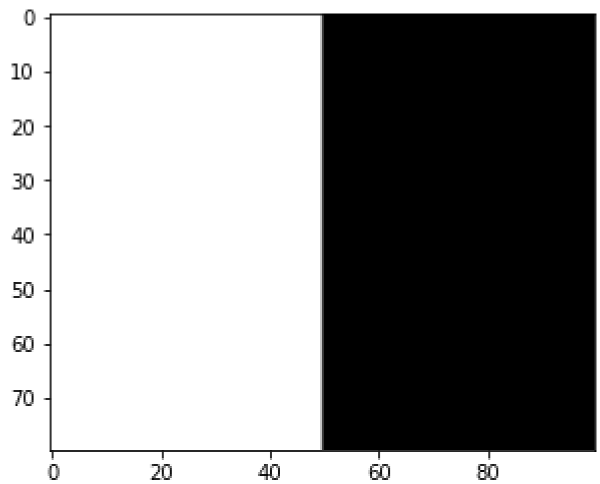
Note:remember to add a Laplace smooth to both the evaluation function and the loss function for better math stability, e.g. $\frac{\text{numerator}+\epsilon}{\text{denominator}+\epsilon}$, where $\epsilon = 1$ typically, also you can try other values.

Now let's randomly split the dataset for training/validation/test

This part has been done for you. But please read through so that you learn the general processing steps.

In [915]:
```
############################################### [TODO] ########################
# define dice coefficient
def dice_coeff(pred, target):
    smooth = 1
    # First let's flatten the matrix to [Batch Size, -1]
    # The flatten operation does not afftect the computation of the above equatio
    num = pred.size(0)
    m1 = pred.view(num, -1).float()  # Flatten
    m2 = target.view(num, -1).float()  # Flatten
    # Then we compute the intersection and the sum of cardinality
    intersection = 2 * torch.sum(m1*m2)
    cardinality = torch.sum(m1) + torch.sum(m2)
    # in case union = 0
    if cardinality == 0:
        cardinality = cardinality + 0.0001
    # Followed by Dice
    dice = intersection / cardinality
    return dice.mean()

# define SoftDICE loss as 1 - SoftDICE
class SoftDICELoss(nn.Module):
    def __init__(self, smooth = 1):
        super(SoftDICELoss, self).__init__()
        self.smooth = smooth
    def forward(self, pred, target):
        num = pred.size(0)
        m1 = pred.view(num, -1).float()  # Flatten
        m2 = target.view(num, -1).float()  # Flatten
        loss = 1 - 2 * torch.sum(m1*m2) / (torch.sum(m1**2)+torch.sum(m2**2))
        return loss.mean()


#Dice Check
#Test image size 100(x)*80(y)
test_mask_img1=np.array([[0]*25+[1]*75]*80)
test_mask_img2=np.array([[1]*50+[0]*50]*80)
test_mask_img_tensor1 = torch.from_numpy(test_mask_img1).type(torch.DoubleTensor)
test_mask_img_tensor2 = torch.from_numpy(test_mask_img2).type(torch.DoubleTensor)

plt.imshow(test_mask_img1, cmap='gray')
plt.show()
plt.imshow(test_mask_img2, cmap='gray')
plt.show()

print(dice_coeff(test_mask_img_tensor1, test_mask_img_tensor2).numpy())
print(SoftDICELoss().forward(pred = test_mask_img_tensor1, target = test_mask_img
```






```
0.4
0.6
```

In [916]:
```
from random import shuffle

def shuffle_two_lists(listA, listB):
    temp = list(zip(listA, listB))
    shuffle(temp)
    return zip(*temp)

train_split_ratio = 0.8
test_split_ratio = 0.1

num_samples = len(img_list)
train_size = int(num_samples*train_split_ratio)
test_size = int(num_samples*test_split_ratio)
val_size = num_samples-train_size-test_size

img_list_shuffled, label_list_shuffled = shuffle_two_lists(img_list, label_list)

train_img_list, train_label_list = img_list_shuffled[:train_size], label_list_shu
val_img_list, val_label_list = img_list_shuffled[train_size:train_size+val_size],
test_img_list, test_label_list = img_list_shuffled[train_size+val_size:], label_l

print ("Training set size: {}".format(len(train_img_list)))
print ("Validation set size: {}".format(len(val_img_list)))
print ("Test set size: {}".format(len(test_img_list)))

#print(train_img_list)
#print(val_img_list)
#print(test_img_list)
```

```
Training set size: 136
Validation set size: 17
Test set size: 17
```

Now let's implement a custom PyTorch dataset!

The important function in a Dataset object includes:

- def __len__(self): get the length of the dataset
- def __getitem__(self,idx): get the image-label pair given the index

You should implement those functions and also add random augmentation functions into the __getitem__() so the data could be augmented on the fly

In [917]:
```
class PedDataset(Dataset):
    def __init__(self, img_path_list, label_path_list,
                 res=(96,96), IF_TRAIN=False):
        self.img_path_list = img_path_list
        self.label_path_list = label_path_list
        self.res = res
        self.IF_TRAIN = IF_TRAIN
        self.scale_factor = 0.1

        self.img_list, self.mask_list = self.preprocess()

    def __len__(self):
        return len(self.img_list)

    def preprocess(self):
        # ------------------------------------------------------------
        # In preprocess(), we 1) read the images, 2) process the masks
        #                     3) resize the images and masks jointly
        # ------------------------------------------------------------
        img_list, mask_list = [], []
        for idx in tqdm(range(len(self.label_path_list))):
            img = cv2.imread(self.img_path_list[idx])
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            mask = cv2.imread(self.label_path_list[idx], cv2.IMREAD_GRAYSCALE)
            mask[mask>=1] = 255

            img = cv2.resize(img, self.res)
            mask = cv2.resize(mask, self.res)

            img_list.append(img)
            mask_list.append(mask)
        return img_list, mask_list


    def __getitem__(self, idx):
        img = self.img_list[idx]
        mask = self.mask_list[idx]

        # Random Augmentation here
        if self.IF_TRAIN:
            # ------------------------------------------------------------
            # Implement random scaling here, with 0.7 prob we rescale the img
            # ------------------------------------------------------------
            if random.random()<0.7:
                scale = np.random.randn()*self.scale_factor+1
                new_res = (int(self.res[0]*scale), int(self.res[1]*scale))
                img_candidate = cv2.resize(img, new_res)
                mask_candidate = cv2.resize(mask, new_res)
                if scale >= 1:
                    x_begin = (new_res[0]-self.res[0])//2
                    y_begin = (new_res[1]-self.res[1])//2
                    img = img_candidate[x_begin:x_begin+self.res[0],y_begin:y_beg
                    mask = mask_candidate[x_begin:x_begin+self.res[0],y_begin:y_b
                else:
                    img = np.zeros_like(img)
                    mask = np.zeros_like(mask)
```

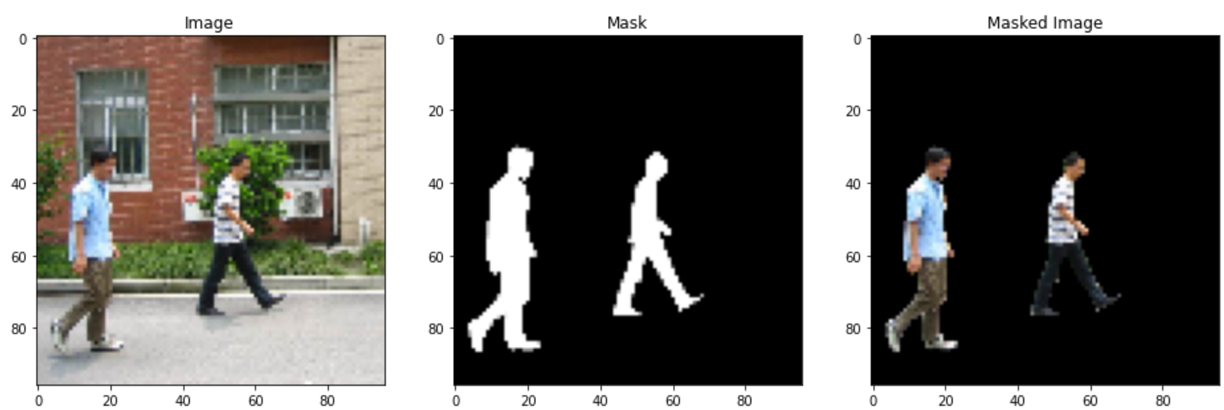
```
                x_begin = (self.res[0]-new_res[0])//2
                y_begin = (self.res[1]-new_res[1])//2
                img[x_begin:x_begin+new_res[0],y_begin:y_begin+new_res[1]] =
                mask[x_begin:x_begin+new_res[0],y_begin:y_begin+new_res[1]] =
        # ------------------------------------------------------------
        # Implement random flip here, with 0.5 prob we flip the image
        # -----------------------------------------------------------
        if random.random()<0.5:
            img = np.array(img[:,::-1,:])
            mask = np.array(mask[:,::-1])

    # ------------------------------------------------------------
    # Need to:
    #       convert img and mask from numpy array to torch tensor
    #       switch the channel dimension to the first dimension
    #       normalize image and maskes
    # ------------------------------------------------------------
    img = torch.Tensor(img).permute(2,0,1)
    label = torch.Tensor(mask).unsqueeze(0)

    return img/255., label/255.
```

Let's check if the dataset is correct!

In [918]:
```
#img_list = sorted(glob.glob("./data/PNGImages/*.png"))
#label_list = sorted(glob.glob("./data/PedMasks/*.png"))
#print (img_list)
#print (label_list)
dataset = PedDataset(img_list, label_list, IF_TRAIN=True)
img, label = dataset[0]
img = img.permute(1,2,0)
label = label.permute(1,2,0)
plt.figure(figsize = (16,48))
plt.subplot(1,3,1)
plt.imshow(img.cpu().numpy())
plt.title('Image')
plt.subplot(1,3,2)
plt.imshow(label.squeeze(),cmap='gray')
plt.title('Mask')
plt.subplot(1,3,3)
plt.imshow((img*label).cpu().numpy())
plt.title('Masked Image')
```

```
100%|██████████| 170/170 [00:00<00:00, 182.33it/s]
```

Out[918]:
```
Text(0.5, 1.0, 'Masked Image')
```



[TODO 3] Start training your network

Okay... It's time to generate the actual dataset!

In [919]:
```
# Use the predefined PedDataset to create train, val and test datasets
#print(train_img_list)
#print(val_img_list)
#print(test_img_list)

train_dataset = PedDataset(train_img_list, train_label_list, IF_TRAIN=True)
val_dataset = PedDataset(val_img_list, val_label_list, IF_TRAIN=True)
test_dataset = PedDataset(test_img_list, test_label_list, IF_TRAIN=True)
```

```
100%|██████████| 136/136 [00:00<00:00, 180.26it/s]
100%|██████████| 17/17 [00:00<00:00, 170.61it/s]
100%|██████████| 17/17 [00:00<00:00, 210.45it/s]
```

Let's prepare some helper functions and define a few parameters for training and evaluation

Note: since the dataset is quite small, the trained model may be prone to overfitting. Please be careful when setting hyper parameters, for example, it is recommended to choose a large batch size and to choose a relatively small number of epochs to stop training before overfitting occurs.

In [946]:
```
################################################ [TODO] ########################
# Specify number of epochs, image scale factor, batch size and learning rate
NUM_EPOCH = 300 # e.g. 40
BATCH_SIZE = 8 # e.g. 8
LR = 0.001 # e.g. 0.001
SAVE_PATH = "./model_final20/"
```

In [947]:
```
train_loader = torch.utils.data.DataLoader(train_dataset,
                                           batch_size=BATCH_SIZE,
                                           shuffle=True,
                                           num_workers=0)
val_loader = torch.utils.data.DataLoader(val_dataset,
                                         batch_size=BATCH_SIZE,
                                         shuffle=False,
                                         num_workers=0)
test_loader = torch.utils.data.DataLoader(test_dataset,
                                         batch_size=BATCH_SIZE,
                                         shuffle=False,
                                         num_workers=0)
```

In [948]:
```
############################################### [TODO] #########################
# Define the training process and return the average loss
def train_epoch(net, data_loader, optimizer, criterion, epoch):
    """
    Input:
    net: The UNet model you defined
    data_loader: a data loader object. Here you should use data loader constructe
    optimizer: The optimizer, preferreablely ADAM
    criterion: The criterion to compute loss
    epoch: The number of current epoch
    """
    # [TODO]: Set model in train mode
    net.train(True)

    loss_stat = []
    for i, img_mask in enumerate(data_loader):

        img, mask = img_mask

        # [TODO]: Send data to device
        img_data = img.type(torch.DoubleTensor)
        mask_data = mask.type(torch.DoubleTensor)

        # [TODO]: Feed data to model to get predictions
        mask_pred = net(img_data)

        # [TODO]: Compute loss and perform update of gradients
        mask_pred_flat = mask_pred.view(mask_pred.shape[0], -1)
        mask_data_flat = mask_data.view(mask_data.shape[0], -1)
        loss = criterion(mask_pred_flat, mask_data_flat)
        loss_stat += [loss.item()]*img.shape[0]

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()


#         print(i)
#         batch_count = 0
#         plt.imshow(img[batch_count][0].numpy())   # i th batch img [batch_count]
#         plt.show()
#         plt.imshow(mask[batch_count][0].numpy())   # i th batch mask [batch_cour
#         plt.show()
#         plt.imshow(mask_pred[batch_count][0].detach().numpy()) # i th batch mask
#         plt.show()

    print ("Epoch {}: [{}/{}] Loss: {:.3f}".format(epoch, len(data_loader), len(d
    return np.mean(loss_stat)

#Test
optimizer = optim.SGD(net.parameters(), lr=LR, momentum=0.9, weight_decay=0.0005)
criterion = nn.BCELoss()
epoch = 1
print(train_epoch(net, test_loader, optimizer, criterion, epoch))
```

```
Epoch 1: [3/3] Loss: 0.244
```

```
0.24361757459420197
```

In [949]:

```
################################################ [TODO] ########################
# Define the training process and return the average loss
# Comparing to training, you don't need to compute gradients
def eval_epoch(net, data_loader, metric, criterion, epoch):
    # [TODO]: set model in eval mode to avoid updating BN layer
    net.eval()

    metric_stat = []
    val_loss_stat = []
    for i, img_mask in enumerate(data_loader):
        img, mask = img_mask
        # [TODO]: send data to device
        img_data = img.type(torch.DoubleTensor)
        mask_data = mask.type(torch.DoubleTensor)

        # [TODO]: feed data to the model. No need to compute grad.
        with torch.no_grad():
            pred = net(img_data)
            val_loss = criterion(pred.view(1, -1),mask_data.view(1, -1))

        # [TODO]: eval the results using DICE function as the metric
        # Convert probability to prediction mask
        pred = pred>0.5
        err = metric(pred.view(1, -1).float(), mask_data.view(1, -1).float())
        metric_stat += [err.item()]*img.shape[0]
        val_loss_stat += [val_loss.item()]*img.shape[0]

    print ("Dice: {:.3f}  Val Loss: {:.3f} ".format(np.mean(metric_stat), np.mear

    return np.mean(val_loss_stat), np.mean(metric_stat)

#Test
print(eval_epoch(net, test_loader, dice_coeff, SoftDICELoss(), 1))
```

```
Dice: 0.725  Val Loss: 0.199
(0.19877573672462912, 0.7252089801956626)
```

In [950]:

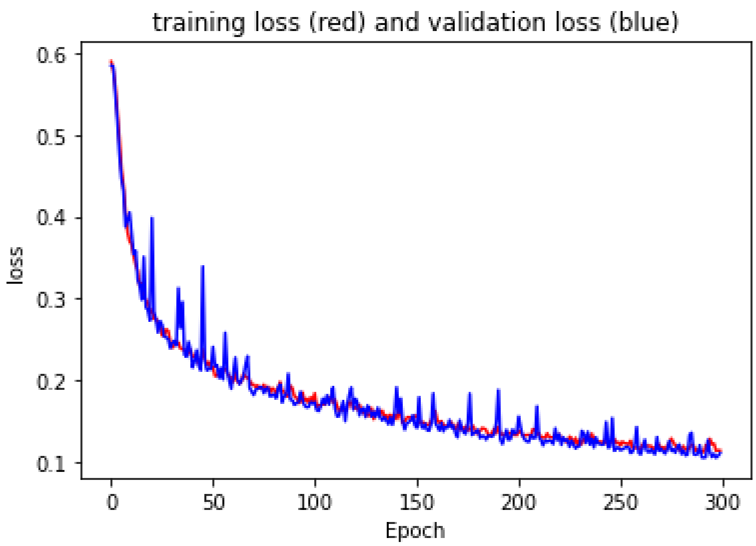
```
################################################ [TODO] ########################
# Create a UNET object. Input channels = 3, output channels = 1
net = UNet(n_channels_in=3, n_channels_out=1)
net = net.double()
net.to(device)  # run net.to(device) if using GPU
#print(net)

# If continuing from previously saved model, run
# net.load_state_dict(torch.load('PATH TO SAVED MODEL FILE'))

# This shows the number of parameters in the network
n_params = sum(p.numel() for p in net.parameters() if p.requires_grad)
print('Number of parameters in network: ', n_params)
```

```
Number of parameters in network:  54241
```

In [951]:
```
############################################### [TODO] #########################
# Define an optimizer for your model.
# Pytorch has built-in package called optim. Most commonly used methods are alrea
# Here we use ADAM as the optimizer
# For usage of ADAM, you can read https://www.programcreek.com/python/example/926
# Also you can use SGD as the optimizer
# For usage of SGD, you can read https://pytorch.org/docs/stable/_modules/torch/o

optimizer = optim.SGD(net.parameters(), lr=LR, momentum=0.9, weight_decay=0.0005)

# The loss function we use here is SoftDICE loss
# If your SoftDICE loss doesn't work, you can use nn.BCELoss()
#criterion = nn.BCELoss()
criterion = SoftDICELoss()

############################################### [TODO] #########################
# Lists used for plotting loss
train_loss_list = []
val_loss_list = []
# Start training
for epoch in range(NUM_EPOCH):
    loss = train_epoch(net, train_loader, optimizer, criterion, epoch)
    val_loss, dice = eval_epoch(net, train_loader, dice_coeff, criterion, epoch)

    # Record losses for each epoch
    train_loss_list.append(loss)
    val_loss_list.append(val_loss)

    # Save the model after each epoch
    if os.path.isdir(SAVE_PATH):
        torch.save(net.state_dict(),SAVE_PATH + 'PedSegEpoch{}.pth'.format(epoch
    else:
        os.makedirs(SAVE_PATH, exist_ok=True)
        torch.save(net.state_dict(),SAVE_PATH + 'PedSegEpoch{}.pth'.format(epoch
    print('Checkpoint {} saved to {}'.format(epoch + 1, SAVE_PATH + 'PedSegEpoch{
```

```
Checkpoint 294 saved to ./model_final20/PedSegEpoch294.pth
Epoch 294: [17/17] Loss: 0.128
Dice: 0.819  Val Loss: 0.112
Checkpoint 295 saved to ./model_final20/PedSegEpoch295.pth
Epoch 295: [17/17] Loss: 0.122
Dice: 0.825  Val Loss: 0.106
Checkpoint 296 saved to ./model_final20/PedSegEpoch296.pth
Epoch 296: [17/17] Loss: 0.122
Dice: 0.837  Val Loss: 0.110
Checkpoint 297 saved to ./model_final20/PedSegEpoch297.pth
Epoch 297: [17/17] Loss: 0.114
Dice: 0.825  Val Loss: 0.105
Checkpoint 298 saved to ./model_final20/PedSegEpoch298.pth
Epoch 298: [17/17] Loss: 0.114
Dice: 0.840  Val Loss: 0.107
Checkpoint 299 saved to ./model_final20/PedSegEpoch299.pth
Epoch 299: [17/17] Loss: 0.114
Dice: 0.813  Val Loss: 0.111
Checkpoint 300 saved to ./model_final20/PedSegEpoch300.pth
```

In [960]:
```
# Plot training loss and validation loss
#print(train_loss_list)
#print(val_loss_list)

plt.plot(np.arange(len(train_loss_list)),train_loss_list, label = "train_loss_lis
plt.plot(np.arange(len(val_loss_list)), val_loss_list, label = "val_loss_list", c

#plt.scatter(train_loss_list,val_loss_list)
plt.xlabel('Epoch')
plt.ylabel('loss')
plt.title('training loss (red) and validation loss (blue)')
plt.show()

#y-axis train_loss,val_loss_list
#x-axis epoch
```



DISCUSSION ABOUT RESULTS

- Discuss your observations on plots of training loss and validation loss and when to stop training is appropriate based on your observations.
- **Fill in your response in the cell below**

Discuss results here (modify this cell)

The results shows that after 300 times of data training. The loss and validation loss finally drop from 0.6 to around 0.3.

You should get a DICE score of at least 0.7

In [953]:
```
# Test on test dataset
for i in range (5):
     test_loss, test_dice = eval_epoch(net, test_loader, dice_coeff, criterion, e
```

```
Dice: 0.770  Val Loss: 0.157
Dice: 0.761  Val Loss: 0.168
Dice: 0.744  Val Loss: 0.181
Dice: 0.749  Val Loss: 0.172
Dice: 0.759  Val Loss: 0.166
```

[Optional] If your model doesn't perform well, you can load a check point

In [954]:
```
#checkpoint_path = './model_final/PedSegEpoch40.pth'
#net = UNet(3,1)
#net.load_state_dict(torch.load(checkpoint_path))
#net.eval()
```

[TODO 4] load one image from testing dataset and plot output mask

In [955]:
```
################################################ [TODO] ########################
# Define a function for prediction/testing
def predict_img(net, img, res, out_threshold=0.5):
    # set the mode of your network to evaluation
    net.eval()

    # convert from Height*Width*Channel TO Batch*Channel*Height*Width(Batch=1) ar
    img = cv2.resize(img, res)/255
    img = np.transpose(img, axes=[2, 0, 1])
    img = torch.from_numpy(np.array([img.tolist()])).type(torch.DoubleTensor)   #
    img = img.to(device)

    with torch.no_grad():
        ################################################ [TODO] ################
        # predict the masks
        pred = net(img)
        pred = pred.squeeze(0).squeeze(0)
        # threshold the probability to generate mask: mask=1 if prob > out_thresh
        pred = torch.round(pred)

    return pred
```

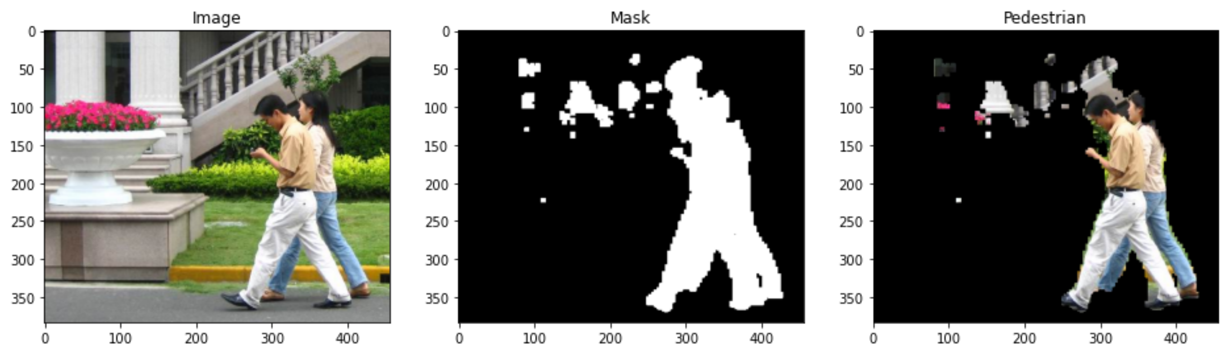
In [961]:
```
################################################ [TODO] ##########################
# Load an image from testing dataset
#idx = ?
img_path, label_path = './data/PNGImages/FudanPed00014.png','./data/PedMasks/Fuda
#img_path = './data/test4.jpg'
img = cv2.imread(img_path)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
label = cv2.imread(label_path)

################################################ [TODO] ##########################

# Predict the mask
mask = predict_img(net=net,
                   img=img,
                   res=(128,128),
                   out_threshold=0.5)

# Rescale the mask back to original image size
mask = mask.numpy().tolist()
mask = (((cv2.resize(np.transpose([mask,mask,mask], axes=[1, 2, 0]), (img.shape[1

#mask = mask
#mask = [mask,mask,mask]
#mask = np.transpose(mask, axes=[1, 2, 0]) # H, W, C
#mask = cv2.resize(mask, (img.shape[1],img.shape[0]))
#mask = mask>0.1
#mask = mask*1
```

In [962]:
```
# Extract the pedestrian from the image using the predicted mask
img_seg = img*mask
```

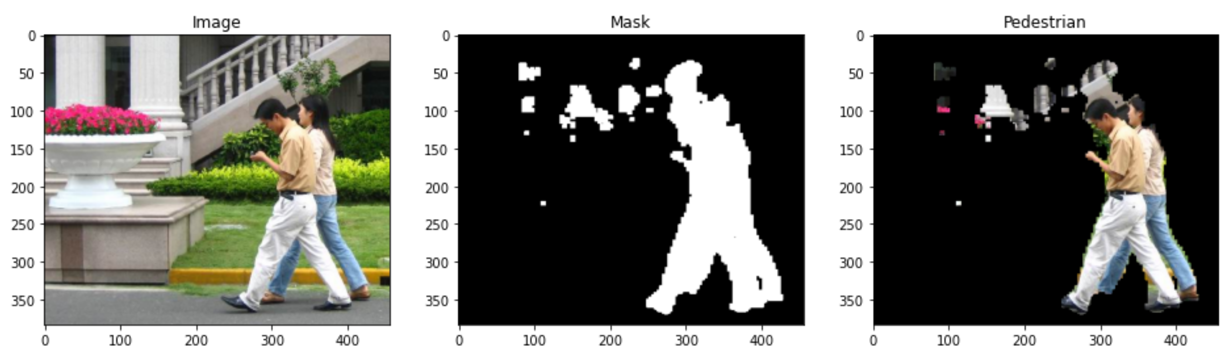
Plot original image and mask image

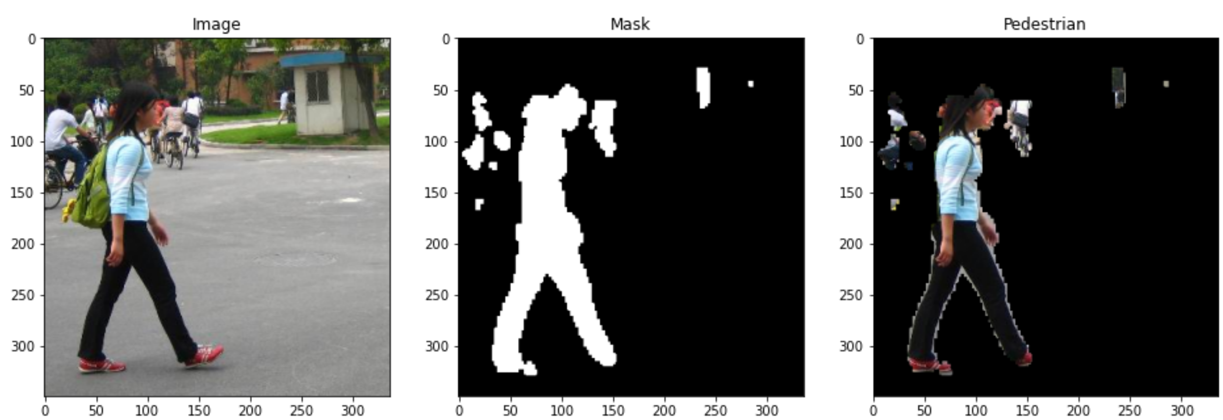
In [963]:
```
################################################ [TODO] ########################
# Plot original images and masks and segmented image
plt.figure(figsize = (16,48))
plt.subplot(1,3,1)
plt.imshow(img)
plt.title('Image')
plt.subplot(1,3,2)
plt.imshow(mask*255)
plt.title('Mask')
plt.subplot(1,3,3)
plt.imshow(img_seg)
plt.title('Pedestrian')
```

Out[963]:
```
Text(0.5, 1.0, 'Pedestrian')
```



In [959]:
```
################################################ [TODO] ##########################
# Display 5 more of test samples
def display(img):
    mask = predict_img(net=net,img=img,res=(128,128),out_threshold=0.5)
    mask = mask.numpy().tolist()
    mask = (((cv2.resize(np.transpose([mask,mask,mask], axes=[1, 2, 0]), (img.sha
    img_seg = img*mask

    plt.figure(figsize = (16,48))
    plt.subplot(1,3,1)
    plt.imshow(img)
    plt.title('Image')
    plt.subplot(1,3,2)
    plt.imshow(mask*255)
    plt.title('Mask')
    plt.subplot(1,3,3)
    plt.imshow(img_seg)
    plt.title('Pedestrian')

img_path_display = ['./data/PNGImages/FudanPed00014.png', './data/PNGImages/Fudar


for i in range(5):
    img = cv2.imread(img_path_display[i])
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    display(img)
```





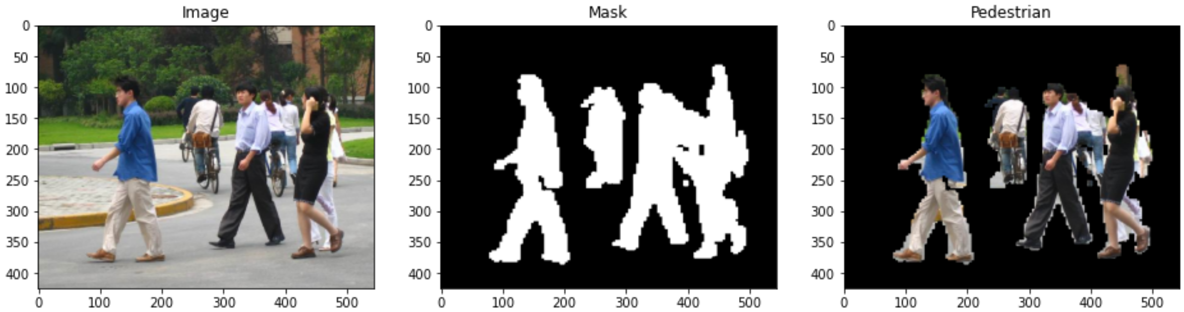


Image
Mask
Pedestrian
0 50 100 150 200 250 300 350 400
0 100 200 300 400 500

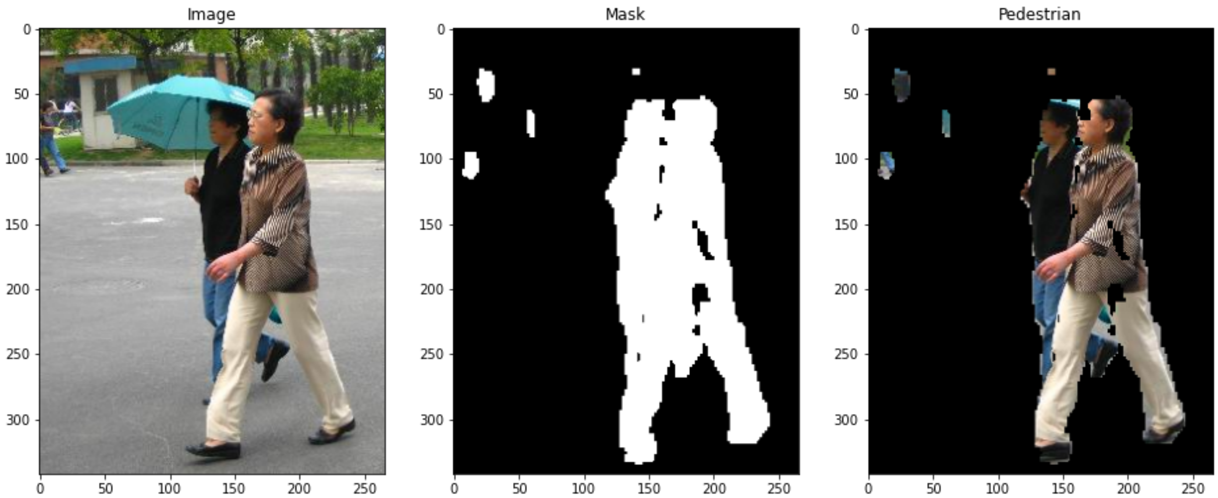


Image
Mask
Pedestrian
0 50 100 150 200 250 300
0 50 100 150 200 250

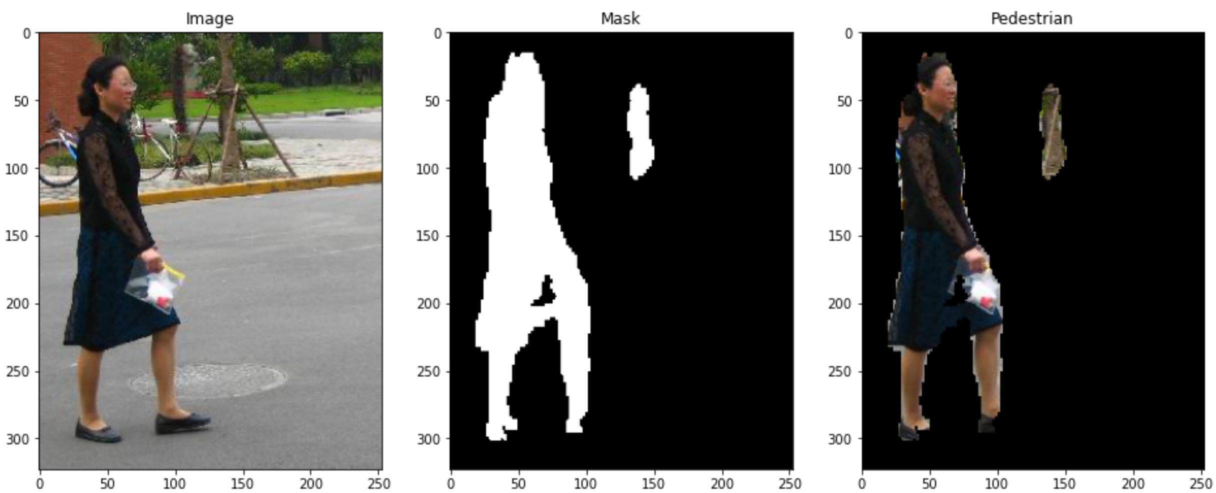


Image
Mask
Pedestrian
0 50 100 150 200 250 300
0 50 100 150 200 250

In [ ]: