CSC420 A3
Zhihong Wang
1002095207
wangz154

1.1. Implement UNet

Binary Cross Entropy Loss:

```
def CELoss(y, y_hat, cw):

    y, y_hat, cw = y.numpy(), y_hat.numpy(), cw.numpy()
    n, c, d1, d2 = y_hat.shape
    loss = np.empty_like(y)
    L=[]

    for i in range(n):
        for k in range(d1):
            for l in range(d2):
                clas = y[i,k,l]
                num = np.exp(y_hat[i,clas,k,l])
                denom = np.exp(y_hat[i,:,k,l]).sum()
                ce = -np.log(num/denom)*cw[clas]
                L.append(ce)

    loss = np.asarray(L).reshape(y.shape)

    return loss
```

Dice Loss:

```
class DiceLoss(nn.Module):

    def __init__(self):

        super(DiceLoss, self).__init__()

    def forward(self, input, target):

        smooth = 1.0

        iflat = input.view(-1)
        tflat = target.view(-1)

        iflat_square = iflat * iflat
        tflat_square = tflat * tflat

        intersection = (iflat * tflat).sum()

        return 1 - ( (2.0 * intersection + smooth) / (iflat_square.sum() + tflat_square.sum() + smooth) )
```

UNet:

```
class DoubleConv(nn.Module):

    def __init__(self, in_ch, out_ch):

        super(DoubleConv,self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(in_ch,out_ch,3,padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace = True),

            nn.Conv2d(out_ch,out_ch,3,padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace = True)
            )

    def forward(self,x):
        return self.conv(x)
```

```python
class UNet(nn.Module):

    def __init__(self,in_ch=3,out_ch=1):

        super(UNet,self).__init__()

        self.conv1 = DoubleConv(in_ch,64)
        self.pool1 = nn.MaxPool2d(2)

        self.conv2 = DoubleConv(64,128)
        self.pool2 = nn.MaxPool2d(2)

        self.conv3 = DoubleConv(128,256)
        self.pool3 = nn.MaxPool2d(2)

        self.conv4 = DoubleConv(256,512)
        self.pool4 = nn.MaxPool2d(2)

        self.conv5 = DoubleConv(512,1024)

        self.up6 = nn.ConvTranspose2d(1024,512,2,stride=2)
        self.conv6 = DoubleConv(1024,512)

        self.up7 = nn.ConvTranspose2d(512,256,2,stride=2)
        self.conv7 = DoubleConv(512,256)

        self.up8 = nn.ConvTranspose2d(256,128,2,stride=2)
        self.conv8 = DoubleConv(256,128)

        self.up9 = nn.ConvTranspose2d(128,64,2,stride=2)
        self.conv9 = DoubleConv(128,64)

        self.conv10 = nn.Conv2d(64,out_ch,1)
```

```python
    def forward(self,x):

        c1 = self.conv1(x)
        p1 = self.pool1(c1)

        c2 = self.conv2(p1)
        p2 = self.pool2(c2)

        c3 = self.conv3(p2)
        p3 = self.pool3(c3)

        c4 = self.conv4(p3)
        p4 = self.pool4(c4)

        c5 = self.conv5(p4)

        up_6 = self.up6(c5)
        merge6 = torch.cat([up_6,c4],dim=1)
        c6 = self.conv6(merge6)

        up_7 = self.up7(c6)
        merge7 = torch.cat([up_7,c3],dim=1)
        c7 = self.conv7(merge7)

        up_8 = self.up8(c7)
        merge8 = torch.cat([up_8,c2],dim=1)
        c8 = self.conv8(merge8)

        up_9 = self.up9(c8)
        merge9 = torch.cat([up_9,c1],dim=1)
        c9 = self.conv9(merge9)

        c10 = self.conv10(c9)

        out = nn.Sigmoid()(c10)

        return out
```

Read Data:

```python
transform = transforms.Compose([transforms.ToPILImage(),
                                transforms.Resize((128, 128)),
                                transforms.ToTensor()])
```

```python
def load_transform_dataset(input_path, mask_path, transform):
    images = []
    masks = []
    size = 0

    for imagepath in input_path:
        size = size + 1
        color_img = cv2.imread(str(imagepath))
        color_img = transform(color_img)
        images.append(color_img)

    for imagepath in mask_path:
        img = cv2.imread(str(imagepath))
        mask_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        mask_img = transform(mask_img)
        masks.append(mask_img)

    return images, masks, size
```

```python
class MyDataset:

    def __init__(self, inputs, masks, size):
        self.inputs = inputs
        self.masks = masks
        self.len = size

    def __len__(self):
        return self.len

    def __getitem__(self, idx):
        return self.inputs[idx], self.masks[idx].float()
```

```python
train_inputs, train_thres, train_size = load_transform_dataset(train_input_path, train_mask_path, transform)
test_inputs, test_thres, test_size = load_transform_dataset(test_input_path, test_mask_path, transform)

train_data = MyDataset(train_inputs, train_thres, train_size)
test_data = MyDataset(test_inputs, test_thres, test_size)

trainloader = torch.utils.data.DataLoader(train_data, batch_size=5, shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=1, shuffle=True)
```

Train:

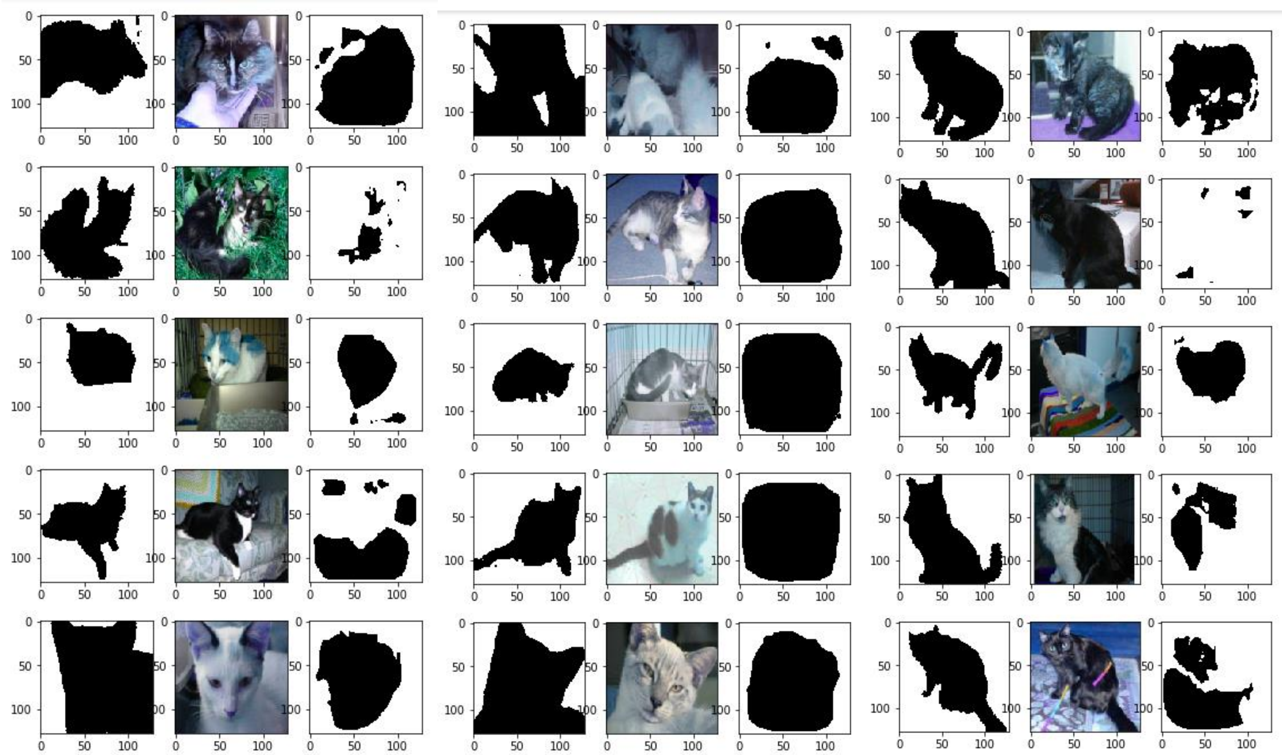
```python
def train(net, trainloader, epochs=10, lr=0.003, loss_fun='dice'):

    if torch.cuda.is_available():
        net = net.cuda()

    criterion = loss_function(loss_fun)
    optimizer = torch.optim.Adam(net.parameters())
    result = []

    for e in range(epochs):
        net.train()
        running_loss = 0

        for images, labels in tqdm(trainloader):
            if torch.cuda.is_available():
                images = images.cuda()
                labels = labels.cuda()

            log_ps = net(images)
            loss = criterion(log_ps, labels)

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

            running_loss += loss.item()
            result.append([log_ps, images, labels])

        else:
            print(f"Training loss: {running_loss/len(trainloader)}")

    return result
```

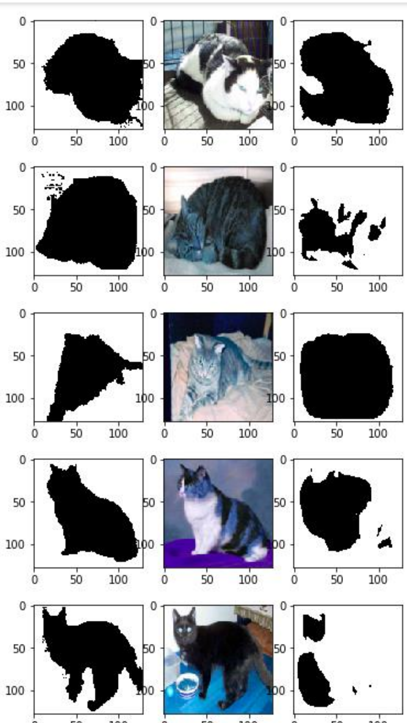
Test:

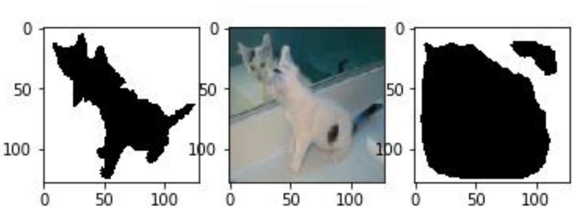
```python
def test(net, testloader, loss_fun='dice'):

    net.eval()
    accuracy = 0
    result = []
    criterion = loss_function(loss_fun)

    for images, labels in testloader:
        if torch.cuda.is_available():
            images = images.cuda()
            labels = labels.cuda()

        log_ps = net(images)
        loss = criterion(log_ps, labels)

        accuracy += 1 - loss
        result.append([log_ps, images, labels])

    print("Test Accuracy: {:.3f}".format(accuracy / len(testloader)))

    return result
```

Binary Cross Entropy Loss Performance:

```
100%|██████████| 15/15 [00:03<00:00,  4.47it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.7127752105394999
100%|██████████| 15/15 [00:03<00:00,  4.46it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.6726645867029826
100%|██████████| 15/15 [00:03<00:00,  4.46it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.6455337603886923
100%|██████████| 15/15 [00:03<00:00,  4.49it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.6150294144948324
100%|██████████| 15/15 [00:03<00:00,  4.46it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.6187748591105143
100%|██████████| 15/15 [00:03<00:00,  4.47it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.5956886331240336
100%|██████████| 15/15 [00:03<00:00,  4.45it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.5800564229488373
100%|██████████| 15/15 [00:03<00:00,  4.44it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.5566850284735362
100%|██████████| 15/15 [00:03<00:00,  4.45it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.5507455865542094
100%|██████████| 15/15 [00:03<00:00,  4.44it/s]
Training loss: 0.545043925444285
Test Accuracy: 0.656
```

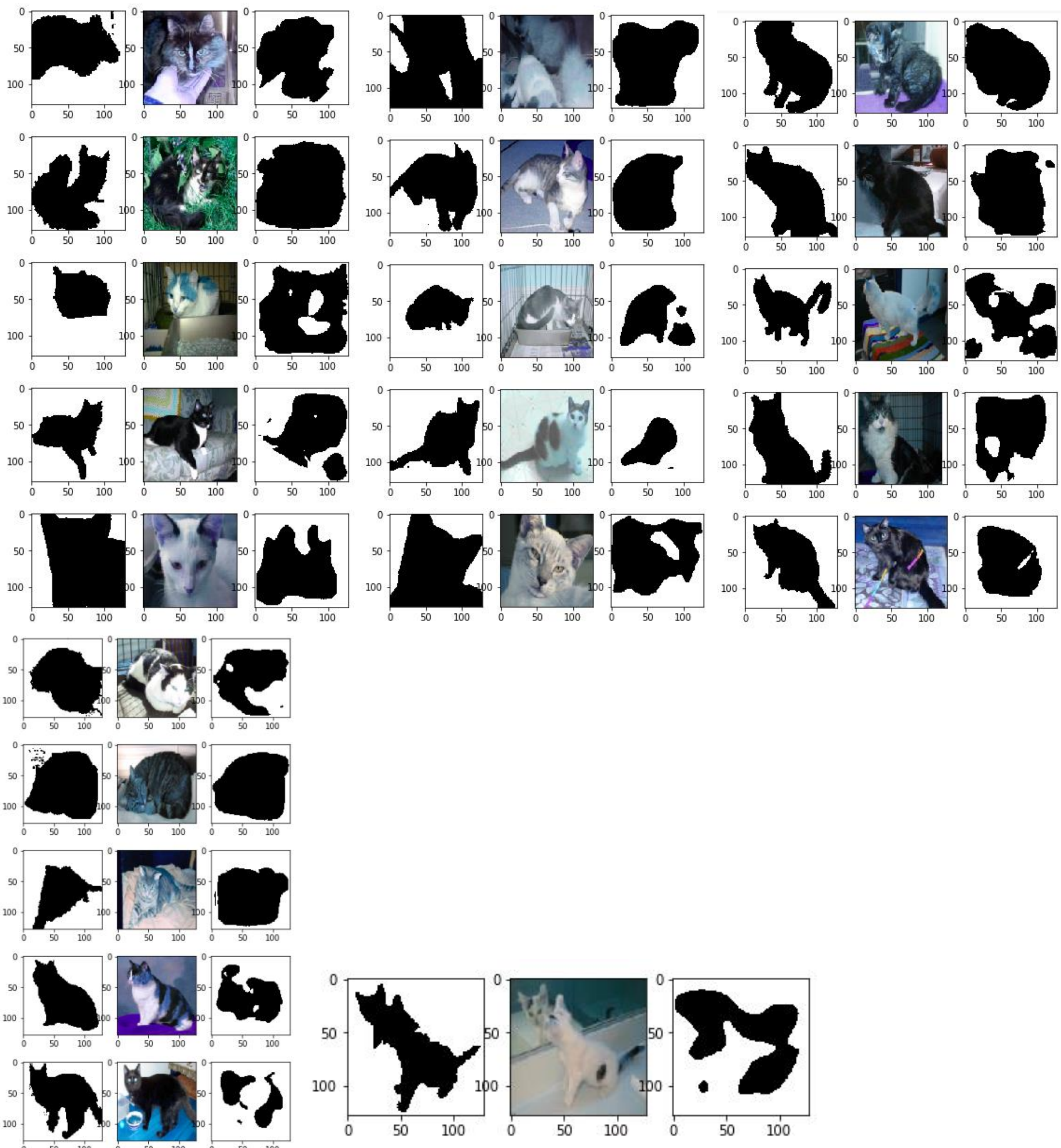






Dice Loss Performance:

```
100%|██████████| 15/15 [00:03<00:00,  4.47it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.3616743922233582
100%|██████████| 15/15 [00:03<00:00,  4.45it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.32314998706181847
100%|██████████| 15/15 [00:03<00:00,  4.44it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.26556370258331297
100%|██████████| 15/15 [00:03<00:00,  4.46it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.2352707068125407
100%|██████████| 15/15 [00:03<00:00,  4.43it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.2460876742998759
100%|██████████| 15/15 [00:03<00:00,  4.44it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.22407633066177368
100%|██████████| 15/15 [00:03<00:00,  4.43it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.22416775226593016
100%|██████████| 15/15 [00:03<00:00,  4.45it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.22270710865656534
100%|██████████| 15/15 [00:03<00:00,  4.42it/s]
  0%|          | 0/15 [00:00<?, ?it/s]Training loss: 0.20345031817754108
100%|██████████| 15/15 [00:03<00:00,  4.43it/s]
Training loss: 0.20759664376576742
Test Accuracy: 0.742
```



Why dice loss is better than bce loss:

Because the actual goal of dice coefficient is to maximize those metrics, and cross-entropy is just a proxy which is easier to maximize using backpropagation. In addition, Dice coefficient performs better at class imbalanced problems by design.

1.2. Data Augmentation

I think there are two different ways to do the data augmentation. First, we can augment the images when loading the them. Second, we can augment the data when building the dataset (i.e. do the augmentation in "getitem"). I tried both and the second way is better in performance improvement. I used four different augmentation techniques: flip, changing the brightness, rotation, affine by randomly applying one of these to the image and mask synchronously when calling "getitem" from the dataset class. The reason of second way is better is because it can randomly assign one of the augmentations to one of the images of loaded data for every epoch (i.e. we can get the unique augmented training data for each epoch). However, if we use the first way (i.e. augmented images when loading the them), then even we randomly assign the augmentations, the images will stay the same as they loaded when training the data, which is not random enough. However, the performance is just better than before for a bit (0.783 compare to 0.779). I think it is because the double the data size is not enough. Larger data may provide even better performance.

```python
def augment_options(image, index):

    if index == 0:
        image = cv2.flip(image, 1)
        image = transform(image)

    if index == 1:
        image = np.uint8(np.clip((1.5 * image + 10), 0, 255))
        image = transform(image)

    if index == 2:
        (h,w) = image.shape[:2]
        center = (w / 2,h / 2)
        M = cv2.getRotationMatrix2D(center,30,1)
        image = cv2.warpAffine(image,M,(w,h))
        image = transform(image)

    if index == 3:
        M = np.float32([[1,0,25],[0,1,50]])
        image = cv2.warpAffine(image,M,(image.shape[1],image.shape[0]))
        image = transform(image)

    elif index != 0 or 1 or 2 or 3:
        image = transform(image)

    return image
```

```python
def load_dataset(input_path, mask_path):
    images = []
    masks = []
    size = 0

    for imagepath in input_path:
        size = size + 1
        color_img = cv2.imread(str(imagepath))
        images.append(color_img)

    for imagepath in mask_path:
        img = cv2.imread(str(imagepath))
        mask_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        masks.append(mask_img)

    return images, masks, size
```

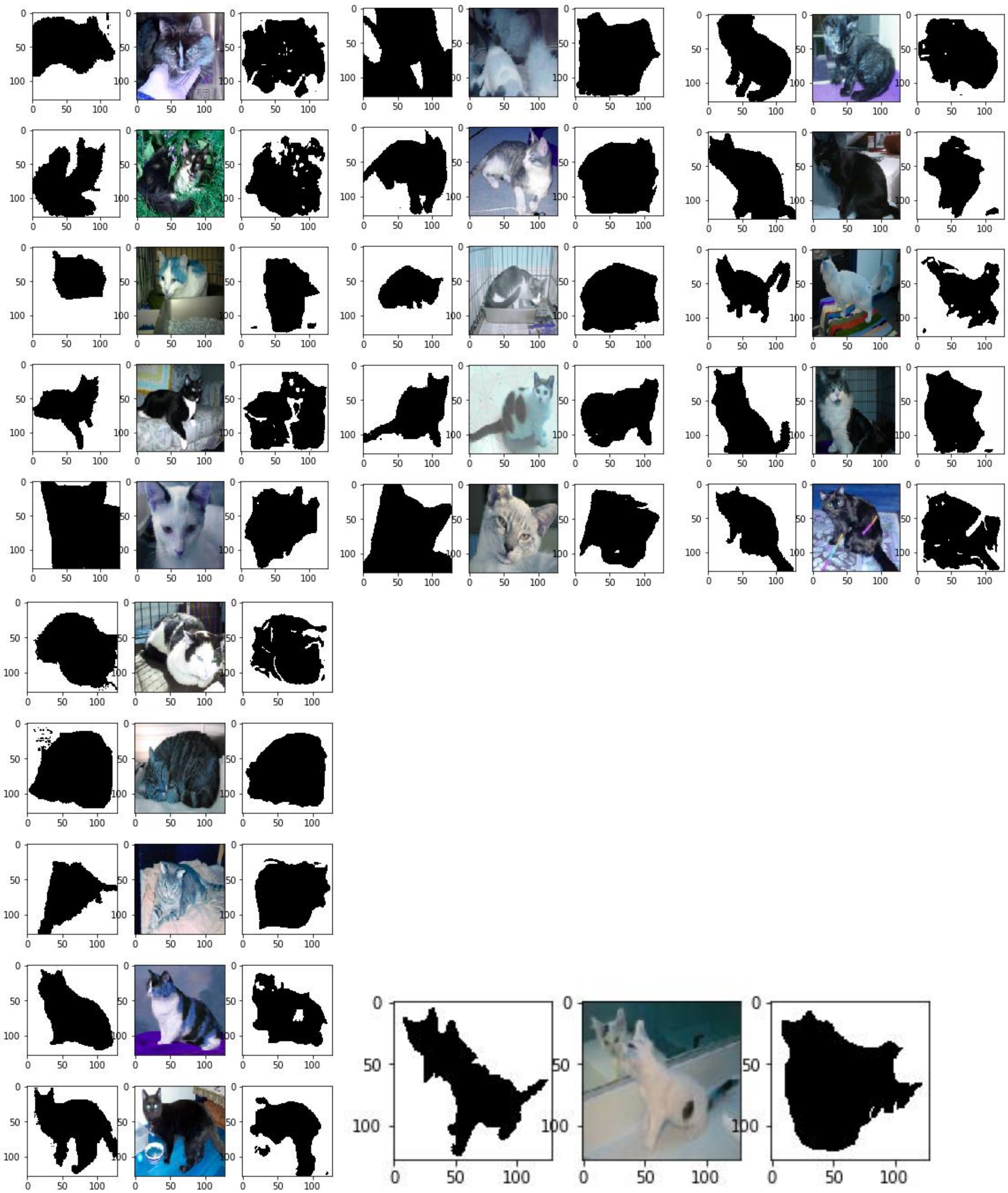
```python
class MyAugmentDataset:

    def __init__(self, inputs, masks, size):
        self.inputs = inputs
        self.masks = masks
        self.len = size

    def __len__(self):
        return self.len

    def __getitem__(self, idx):
        image = self.inputs[idx]
        mask = self.masks[idx]
        seed = np.random.randint(0, 999999)
        random_index = np.random.randint(0, 4)
#         print("random index = ", random_index)
        np.random.seed(seed)
        image = augment_options(image, random_index)
        np.random.seed(seed)
        mask = augment_options(mask, random_index)
        return image, mask.float()

```

I doubled the data size:

```python
train_inputs, train_thres, train_size = load_dataset(train_input_path, train_mask_path)
test_inputs, test_thres, test_size = load_transform_dataset(test_input_path, test_mask_path, transform)

train_data = MyAugmentDataset(train_inputs + train_inputs, train_thres + train_thres, train_size + train_size)
test_data = MyDataset(test_inputs, test_thres, test_size)

augment_trainloader = torch.utils.data.DataLoader(train_data, batch_size=5, shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=1)
```

Performance:

```
100%|██████████| 24/24 [00:07<00:00,  3.42it/s]
  0%|          | 0/24 [00:00<?, ?it/s]Training loss: 0.3007693663239479
100%|██████████| 24/24 [00:07<00:00,  3.44it/s]
  0%|          | 0/24 [00:00<?, ?it/s]Training loss: 0.25509018699328107
100%|██████████| 24/24 [00:06<00:00,  3.44it/s]
  0%|          | 0/24 [00:00<?, ?it/s]Training loss: 0.23498573899269104
100%|██████████| 24/24 [00:07<00:00,  3.42it/s]
  0%|          | 0/24 [00:00<?, ?it/s]Training loss: 0.2229666213194529
100%|██████████| 24/24 [00:07<00:00,  3.41it/s]
  0%|          | 0/24 [00:00<?, ?it/s]Training loss: 0.22957989821831384
100%|██████████| 24/24 [00:07<00:00,  3.44it/s]
  0%|          | 0/24 [00:00<?, ?it/s]Training loss: 0.2115496868888537
100%|██████████| 24/24 [00:07<00:00,  3.44it/s]
  0%|          | 0/24 [00:00<?, ?it/s]Training loss: 0.20879298945267996
100%|██████████| 24/24 [00:07<00:00,  3.42it/s]
  0%|          | 0/24 [00:00<?, ?it/s]Training loss: 0.20033962527910867
100%|██████████| 24/24 [00:07<00:00,  3.42it/s]
  0%|          | 0/24 [00:00<?, ?it/s]Training loss: 0.2058428352077802
100%|██████████| 24/24 [00:07<00:00,  3.38it/s]
Training loss: 0.18970023343960443
Test Accuracy: 0.783
```



1.3. Transfer Learning (Oxford IIIT Pet Dataset: https://www.robots.ox.ac.uk/~vgg/data/pets/)
My trained net (the weight file): https://drive.google.com/open?id=1my7gqC9yoBw9VP94AP84IJAqGZw6GNGr

How I perform the transfer learning:

Step 0: Download the data from the website, and separate data reasonably.
Step 1. Find data path:

```
transfer_train_input_path = glob.glob("./images/*.jpg")
transfer_train_input_path.sort()
transfer_train_mask_path = glob.glob("./annotations/trimaps/*.png")
transfer_train_mask_path.sort()
```

Step 2. Load transfer data, transform them to tensor and handle corrupted data:

```
transform = transforms.Compose([transforms.ToPILImage(),
                                transforms.Resize((128, 128)),
                                transforms.ToTensor()])
def load_transfer_dataset(input_path, mask_path, transform):
    images = []
    masks = []
    size = 0
    corrupted_index = []
    print("original number of images = ", len(input_path), ", original number of masks = ", len(mask_path))

    for imagepath in input_path:
        size = size + 1
        color_img = cv2.imread(str(imagepath))
        if color_img is None:
            index = input_path.index(imagepath)
            corrupted_index.append(index)
            continue
        color_img = transform(color_img)
        images.append(color_img)

    for imagepath in mask_path:
        mask = Image.open(imagepath).convert("L")
        mask = np.array(mask)
        mask = (mask == 1)
        mask = (mask * 255).astype("uint8")
        mask_img = transform(mask)
        masks.append(mask_img)

    for index in sorted(corrupted_index, reverse=True):
        del masks[index]

    size = size - len(corrupted_index)

    print("number of images = ", len(images), ", number of masks = ", len(masks), ", size = ",  size)

    return images, masks, size
```

Step 3. Build basic dataset:

```python
class MyDataset:

    def __init__(self, inputs, masks, size):
        self.inputs = inputs
        self.masks = masks
        self.len = size

    def __len__(self):
        return self.len

    def __getitem__(self, idx):
        return self.inputs[idx], self.masks[idx].float()
```

Step 4. Use the same UNet, loss function and train/test function from 1.1 and 1.2. Execute and save the trained net:

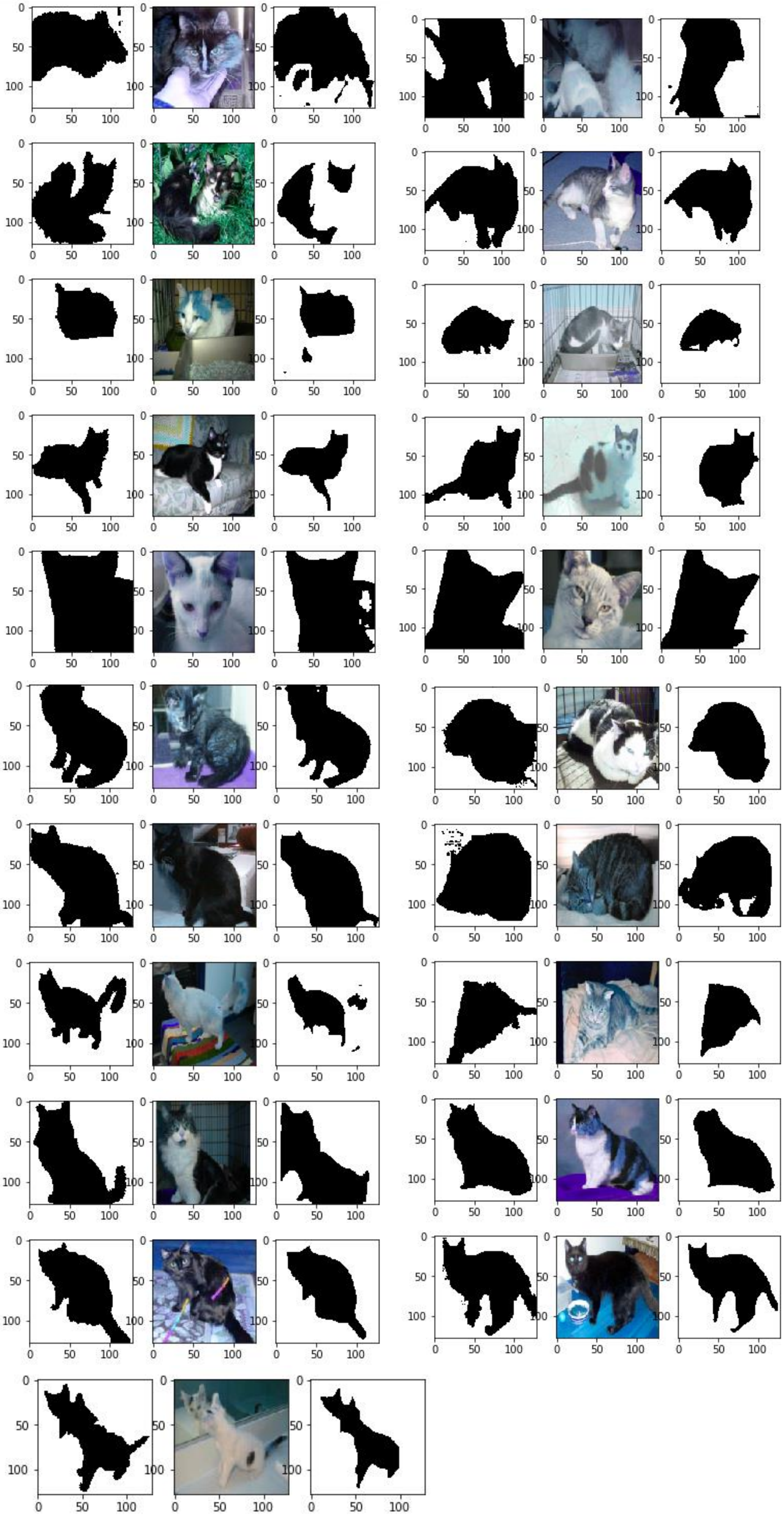
```python
if __name__ == '__main__':

    torch.backends.cudnn.benchmark = True
    torch.cuda.empty_cache()
    # os.environ['CUDA_VISIBLE_DEVICES'] = '0, 1, 2'

    # train_inputs, train_masks, train_size = load_transform_dataset(train_input_path, train_mask_path, transform)
    # transfer_train_inputs, transfer_train_masks, transfer_train_size = load_transform_dataset(transfer_train_input_path, transfer_train_mask_path, transform)
    transfer_train_inputs, transfer_train_masks, transfer_train_size = load_transfer_dataset(transfer_train_input_path,
                                                                                              transfer_train_mask_path,
                                                                                              transform)
    test_inputs, test_masks, test_size = load_transform_dataset(test_input_path, test_mask_path, transform)

    # train_inputs, train_masks, train_size = load_dataset(train_input_path, train_mask_path)
    # train_data = MyAugmentDataset(train_inputs, train_masks, train_size, all_transform)

    # train_data = MyDataset(train_inputs, train_mask, train_size)
    train_data = MyDataset(transfer_train_inputs, transfer_train_masks, transfer_train_size)
    test_data = MyDataset(test_inputs, test_masks, test_size)

    trainloader = torch.utils.data.DataLoader(train_data, batch_size=64, shuffle=True, pin_memory=True)
    testloader = torch.utils.data.DataLoader(test_data, batch_size=1, shuffle=True)

    transfer_learning_net = UNet()
    if torch.cuda.is_available():
        transfer_learning_net = transfer_learning_net.cuda()
    # transfer_learning_net.load_state_dict(torch.load("./tf_model_post.pt"))

    # net = UNet()
    # if torch.cuda.is_available():
    #     net = net.cuda()

    # train(net, trainloader)
    # te_result = test(net, testloader)

    train(transfer_learning_net, trainloader)
    torch.save(transfer_learning_net.state_dict(), "./my_net.pt")
    te_result = test(transfer_learning_net, testloader)
```

Step 5. Check the performance, the training set is the transfer data from Oxford, the test set is from a3 cat_data

```
number of images =  7384 , number of masks =  7384 , size =  7384
original number of images =  21 , original number of masks =  21
number of images =  21 , number of masks =  21 , size =  21
 99%|█████████▉| 115/116 [00:48<00:00,  2.82it/s]Training loss: 0.24215494712878918
100%|██████████| 116/116 [00:54<00:00,  2.14it/s]
 99%|█████████▉| 115/116 [00:41<00:00,  2.65it/s]Training loss: 0.16631031087760267
100%|██████████| 116/116 [00:41<00:00,  2.78it/s]
100%|██████████| 116/116 [00:41<00:00,  2.81it/s]
  0%|          | 0/116 [00:00<?, ?it/s]Training loss: 0.1374676489624484
100%|██████████| 116/116 [00:41<00:00,  2.80it/s]
Training loss: 0.1150483694569818
 99%|█████████▉| 115/116 [00:41<00:00,  2.78it/s]Training loss: 0.10264479394616752
100%|██████████| 116/116 [00:41<00:00,  2.80it/s]
100%|██████████| 116/116 [00:41<00:00,  2.79it/s]
  0%|          | 0/116 [00:00<?, ?it/s]Training loss: 0.09419669776127257
100%|██████████| 116/116 [00:41<00:00,  2.79it/s]
  0%|          | 0/116 [00:00<?, ?it/s]Training loss: 0.087798109342312
 99%|█████████▉| 115/116 [00:41<00:00,  2.77it/s]Training loss: 0.07970565300563286
100%|██████████| 116/116 [00:41<00:00,  2.79it/s]
 99%|█████████▉| 115/116 [00:41<00:00,  2.71it/s]Training loss: 0.07418641139721048
100%|██████████| 116/116 [00:41<00:00,  2.77it/s]
 99%|█████████▉| 115/116 [00:41<00:00,  2.77it/s]Training loss: 0.0730496783708704
100%|██████████| 116/116 [00:41<00:00,  2.78it/s]
100%|██████████| 116/116 [00:41<00:00,  2.78it/s]
  0%|          | 0/116 [00:00<?, ?it/s]Training loss: 0.06982231705353178
 99%|█████████▉| 115/116 [00:41<00:00,  2.77it/s]Training loss: 0.06446692604443123
100%|██████████| 116/116 [00:41<00:00,  2.78it/s]
100%|██████████| 116/116 [00:41<00:00,  2.78it/s]
  0%|          | 0/116 [00:00<?, ?it/s]Training loss: 0.06139969517444742
100%|██████████| 116/116 [00:41<00:00,  2.78it/s]
Training loss: 0.05790502089878608
 99%|█████████▉| 115/116 [00:42<00:00,  2.68it/s]Training loss: 0.05404536426067352
100%|██████████| 116/116 [00:42<00:00,  2.71it/s]
Test Accuracy: 0.867
```

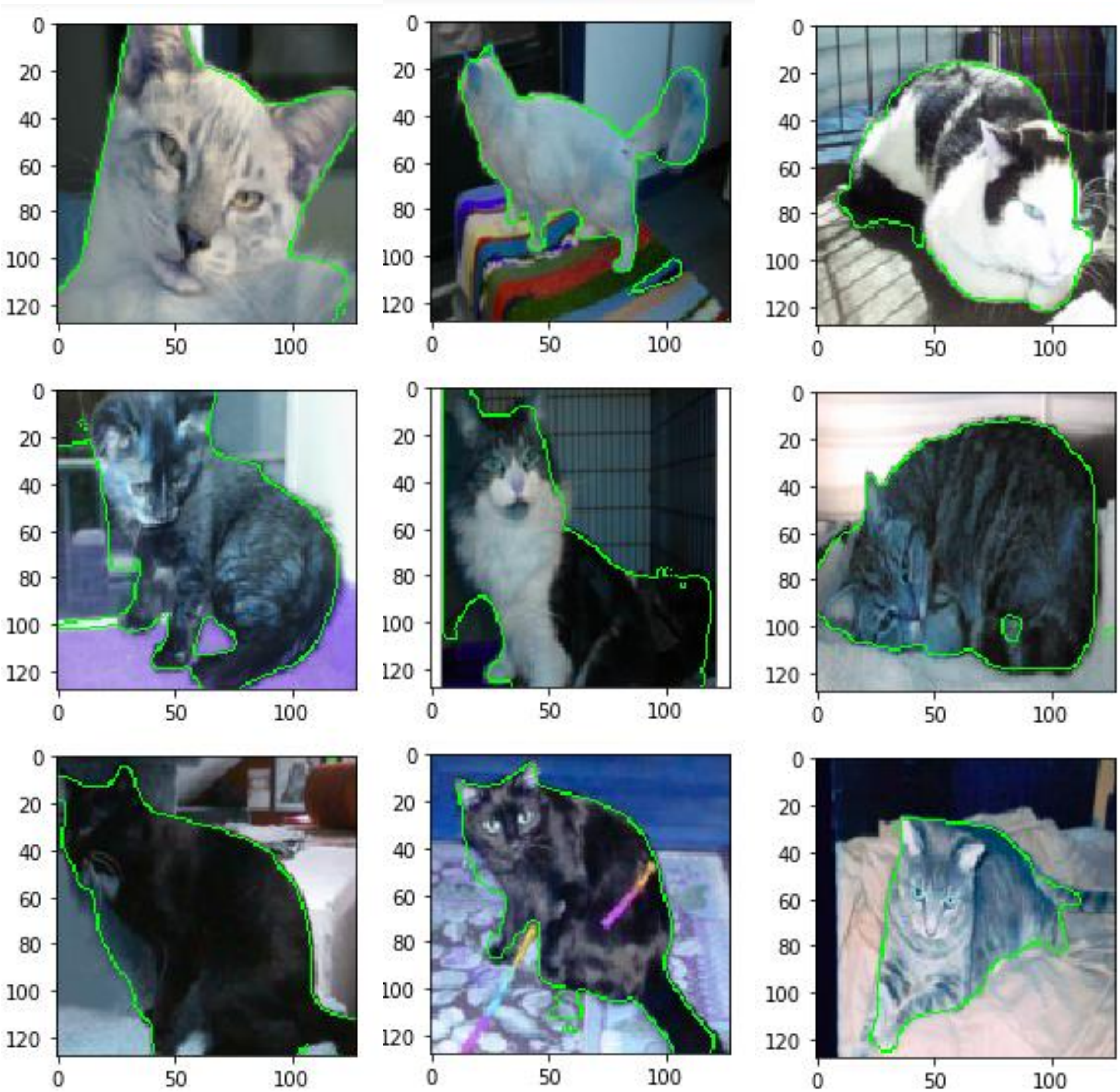
```python
transfer_learning_net = UNet()
if torch.cuda.is_available():
    transfer_learning_net = transfer_learning_net.cuda()

transfer_learning_net.load_state_dict(torch.load("./drive/My Drive/csc420/my_net_3.pt"))
transfer_te_result = test(transfer_learning_net, testloader)
```

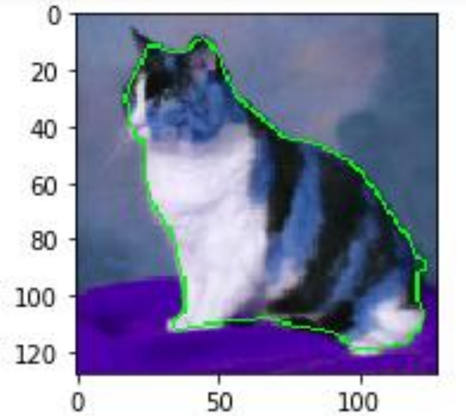
```
Test Accuracy: 0.911
```

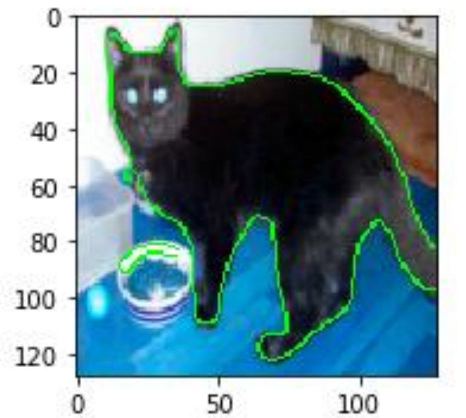


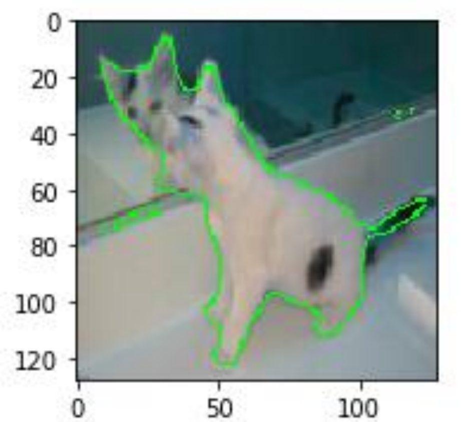
1.4. Visualizing segmentation predictions:

```
def visualize(result, index):
    ret, binary_pred = cv2.threshold(result[index][0].cpu().squeeze().detach().numpy(),0.05,1,0)
    outputs = np.array(binary_pred)*255
    images = np.array(result[index][1].cpu().squeeze().permute(1,2,0))

    result = np.copy(images)
    outputs_canny = cv2.Canny(np.uint8(outputs), 20, 250)
    result[outputs_canny == 255] = [0,1,0]
    plt.subplot(1, 2, 1)
    plt.imshow(result)
    plt.tight_layout()

visualize(transfer_te_result, 18)
```


0 20 40 60 80 100 120
0 50 100
0 20 40 60 80 100 120
0 50 100
0 20 40 60 80 100 120
0 50 100
0 20 40 60 80 100 120
0 50 100
0 20 40 60 80 100 120
0 50 100
0 20 40 60 80 100 120
0 50 100
0 20 40 60 80 100 120
0 50 100
0 20 40 60 80 100 120
0 50 100
0 20 40 60 80 100 120
0 50 100


0 20 40 60 80 100 120
0 50 100


0 20 40 60 80 100 120
0 50 100


0 20 40 60 80 100 120
0 50 100

2.1. Problem definition
Input and output of my neural network:
Input: The original image generated by function noisy_circle from main.py.
Output: The prediction image of the circle.

```
def train(net, trainloader, epochs=20, loss_fun='iou'):
    if torch.cuda.is_available():
        net.cuda()

    criterion = loss_function(loss_fun)
    optimizer = torch.optim.Adam(net.parameters())

    for e in range(epochs):
        net.train()
        running_loss = 0
        for images, labels in tqdm(trainloader):
            if torch.cuda.is_available():
                images = images.cuda()
                labels = labels.cuda()
            log_ps = net(images)
            loss = criterion(log_ps, labels)

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

            running_loss += loss.item()
        else:
            print(f"Training loss: {running_loss / len(trainloader)}")
```

Loss function: Dice Loss

```
class DiceLoss(nn.Module):
    def __init__(self):
        super(DiceLoss, self).__init__()

    def forward(self, input, target):
        smooth = 1.

        iflat = input.view(-1)
        tflat = target.view(-1)
        intersection = (iflat * tflat).sum()

        return 1 - ((2. * intersection + smooth) /
                    ((iflat * iflat).sum() + (tflat * tflat).sum() + smooth))
```

Why I represent the problem this way:
I want to use the similar process from question 1 to question 2 (i.e. using UNet for question 2). UNet is a good neural network for image segmentation. For question 2, when we generating the noisy image with the circle, we can also generate the “mask” image synchronously, which is the original generated circle image without the noise. Then, we can apply the similar technique from Q1 to Q2: using the noisy circle images and the corresponding “masks” (i.e. clear circle images) to train the net and predict the “masks” (i.e. clear circle images) for random inputs.

2.2. Implementation

My neural network: UNet with less layers

```python
class DoubleConv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(DoubleConv, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(out_ch),
            nn.Conv2d(out_ch, out_ch, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(out_ch)
        )

    def forward(self, x):
        return self.conv(x)
```

```python
class UNet(nn.Module):
    def __init__(self, in_ch=1, out_ch=1):
        super(UNet, self).__init__()

        self.conv1 = DoubleConv(in_ch, 64)
        self.pool1 = nn.MaxPool2d(2)

        self.conv2 = DoubleConv(64, 128)
        self.pool2 = nn.MaxPool2d(2)

        # self.conv3 = DoubleConv(128,256)
        # self.pool3 = nn.MaxPool2d(2)

        # self.conv4 = DoubleConv(256,512)
        # self.pool4 = nn.MaxPool2d(2)

        self.conv5 = DoubleConv(128, 256)

        # self.up6 = nn.ConvTranspose2d(1024,512,2,stride=2)
        # self.conv6 = DoubleConv(1024,512)

        # self.up7 = nn.ConvTranspose2d(512,256,2,stride=2)
        # self.conv7 = DoubleConv(512,256)

        self.up8 = nn.ConvTranspose2d(256, 128, 2, stride=2)
        self.conv8 = DoubleConv(256, 128)

        self.up9 = nn.ConvTranspose2d(128, 64, 2, stride=2)
        self.conv9 = DoubleConv(128, 64)

        self.conv10 = nn.Conv2d(64, out_ch, 1)
```

```python
    def forward(self, x):
        c1 = self.conv1(x)
        p1 = self.pool1(c1)
        c2 = self.conv2(p1)
        p2 = self.pool2(c2)

        # c3 = self.conv3(p2)
        # p3 = self.pool3(c3)
        # c4 = self.conv4(p3)
        # p4 = self.pool4(c4)

        c5 = self.conv5(p2)

        # up_6 = self.up6(c5)
        # merge6 = torch.cat([up_6,c4],dim=1)
        # c6 = self.conv6(merge6)
        # up_7 = self.up7(c5)
        # merge7 = torch.cat([up_7,c3],dim=1)
        # c7 = self.conv7(merge7)

        up_8 = self.up8(c5)
        merge8 = torch.cat([up_8, c2], dim=1)
        c8 = self.conv8(merge8)
        up_9 = self.up9(c8)
        merge9 = torch.cat([up_9, c1], dim=1)
        c9 = self.conv9(merge9)

        c10 = self.conv10(c9)

        out = nn.Sigmoid()(c10)
        return out
```

Visualization of network (includes Parameters):

```python

model = UNet()
if torch.cuda.is_available():
    model.cuda()
summary(model, (1, 200, 200))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 200, 200]             640
              ReLU-2         [-1, 64, 200, 200]               0
       BatchNorm2d-3         [-1, 64, 200, 200]             128
            Conv2d-4         [-1, 64, 200, 200]          36,928
              ReLU-5         [-1, 64, 200, 200]               0
       BatchNorm2d-6         [-1, 64, 200, 200]             128
        DoubleConv-7         [-1, 64, 200, 200]               0
         MaxPool2d-8         [-1, 64, 100, 100]               0
            Conv2d-9        [-1, 128, 100, 100]          73,856
             ReLU-10        [-1, 128, 100, 100]               0
      BatchNorm2d-11        [-1, 128, 100, 100]             256
           Conv2d-12        [-1, 128, 100, 100]         147,584
             ReLU-13        [-1, 128, 100, 100]               0
      BatchNorm2d-14        [-1, 128, 100, 100]             256
       DoubleConv-15        [-1, 128, 100, 100]               0
        MaxPool2d-16          [-1, 128, 50, 50]               0
           Conv2d-17          [-1, 256, 50, 50]         295,168
             ReLU-18          [-1, 256, 50, 50]               0
      BatchNorm2d-19          [-1, 256, 50, 50]             512
           Conv2d-20          [-1, 256, 50, 50]         590,080
             ReLU-21          [-1, 256, 50, 50]               0
      BatchNorm2d-22          [-1, 256, 50, 50]             512
       DoubleConv-23          [-1, 256, 50, 50]               0
  ConvTranspose2d-24        [-1, 128, 100, 100]         131,200
           Conv2d-25        [-1, 128, 100, 100]         295,040
             ReLU-26        [-1, 128, 100, 100]               0
      BatchNorm2d-27        [-1, 128, 100, 100]             256
           Conv2d-28        [-1, 128, 100, 100]         147,584
             ReLU-29        [-1, 128, 100, 100]               0
      BatchNorm2d-30        [-1, 128, 100, 100]             256
       DoubleConv-31        [-1, 128, 100, 100]               0
  ConvTranspose2d-32         [-1, 64, 200, 200]          32,832
           Conv2d-33         [-1, 64, 200, 200]          73,792
             ReLU-34         [-1, 64, 200, 200]               0
      BatchNorm2d-35         [-1, 64, 200, 200]             128
           Conv2d-36         [-1, 64, 200, 200]          36,928
             ReLU-37         [-1, 64, 200, 200]               0
```


```
      BatchNorm2d-30        [-1, 128, 100, 100]             256
       DoubleConv-31        [-1, 128, 100, 100]               0
  ConvTranspose2d-32         [-1, 64, 200, 200]          32,832
           Conv2d-33         [-1, 64, 200, 200]          73,792
             ReLU-34         [-1, 64, 200, 200]               0
      BatchNorm2d-35         [-1, 64, 200, 200]             128
           Conv2d-36         [-1, 64, 200, 200]          36,928
             ReLU-37         [-1, 64, 200, 200]               0
```


```
      BatchNorm2d-38         [-1, 64, 200, 200]             128
       DoubleConv-39         [-1, 64, 200, 200]               0
           Conv2d-40          [-1, 1, 200, 200]              65
================================================================
Total params: 1,864,257
Trainable params: 1,864,257
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.15
Forward/backward pass size (MB): 481.26
Params size (MB): 7.11
Estimated Total Size (MB): 488.53
----------------------------------------------------------------
```

Explain of architecture:

As we already know from Q1, U-Net is a convolutional neural network for biomedical image segmentation. The network is based on the fully convolutional network and its architecture was modified and extended to work with fewer training images and to yield more precise segmentations. The main idea is to supplement a usual contracting network by successive layers, where pooling operations are replaced by up-sampling operators. Hence these layers increase the resolution of the output. What's more, a successive convolutional layer can then learn to assemble a precise output based on this information. U-Net has a large number of feature channels in the up-sampling part, which allow the network to propagate context information to higher resolution layers.

As I said before, I want to use the similar process from question 1 to question 2 (i.e. using U-Net for question 2). U-Net is a good neural network for image segmentation. For question 2, when we generating the noisy image with the circle, we can also generate the "mask" image synchronously, which is the original generated circle image without the noise. Then, we can apply the similar technique from Q1 to Q2: using the noisy circle images and the corresponding "masks" (i.e. clear circle images) to train the net and predict the "masks" (i.e. clear circle images) for random inputs.

Please notice that I changed the "noisy_circle" function from main.py slightly, by returning an extra result "img" as the "mask" (i.e. clear circle image without noise). The original "img" is called "noisy_img" now.

```python
def noisy_circle(size, radius, noise):
    img = np.zeros((size, size), dtype=np.float)

    # Circle
    row = np.random.randint(size)
    col = np.random.randint(size)
    rad = np.random.randint(10, max(10, radius))
    draw_circle(img, row, col, rad)

    noisy_img = img.copy()

    # Noise
    noisy_img += noise * np.random.rand(*img.shape)
    return (row, col, rad), noisy_img, img
```

Training set:

```python
transform = transforms.Compose([transforms.ToTensor()])

def load_transform_train_dataset(transform):
    images = []
    masks = []
    size = 0
    while size < 100:
        params, noisy_img, mask_img = noisy_circle(200, 50, 2)
        noisy_img = transform(noisy_img)
        mask_img = transform(mask_img)
        images.append(noisy_img)
        masks.append(mask_img)
        size = size + 1
    return images, masks, size
```

Testing set:

```python
def load_transform_test_dataset(transform):
    images = []
    masks = []
    size = 0
    while size < 10:
        params, noisy_img, mask_img = noisy_circle(200, 50, 2)
        noisy_img = transform(noisy_img)
        mask_img = transform(mask_img)
        images.append(noisy_img)
        masks.append(mask_img)
        size = size + 1
    return images, masks, size
```

Find Circle:

```python
def find_circle(img,mask):
    # Fill in this function
    noisy_img = [transform(img)]
    mask_img = [transform(mask)]
    size = 1

    test_data = MyDataset(noisy_img, mask_img, size)
    testloader = torch.utils.data.DataLoader(test_data, batch_size=1)

    find_circle_net = UNet()
    if torch.cuda.is_available():
        find_circle_net = find_circle_net.cuda()
    find_circle_net.load_state_dict(torch.load("./q2_net_iou_loss.pt"))

    test_result = test(find_circle_net, testloader)
    center, radius = detect_circle(test_result[0][0].cpu().squeeze().detach().numpy())
    print("prediction: ", center, radius)
    print("==================================================================================================")
    return center[1], center[0], radius
```

Detect Circle:

```python
def detect_circle(predict):
    # print(np.size(predict,0), np.size(predict,1))
    predict = predict[1:(np.size(predict,0)-1),1:(np.size(predict,1)-1)]
    # print(np.size(predict,0), np.size(predict,1))
    ret, binary = cv2.threshold(predict,0.04,1,0)
    # print(binary)

    circle_indexs = np.where(binary==np.max(binary))
    # print(circle_indexs)

    min_x = circle_indexs[0][0]
    y_index_for_min_x = np.where(circle_indexs[0]==min_x)[0]
    y_for_min_x_array = circle_indexs[1][y_index_for_min_x]
    y_for_min_x = np.mean(y_for_min_x_array)
    # print('min row = ', min_x, y_for_min_x)

    max_x = circle_indexs[0][-1]
    y_index_for_max_x = np.where(circle_indexs[0]==max_x)[0]
    y_for_max_x_array = circle_indexs[1][y_index_for_max_x]
    y_for_max_x = np.mean(y_for_max_x_array)
    # print('max row = ', max_x, y_for_max_x)

    min_y = min(circle_indexs[1])
    x_index_for_min_y = np.where(circle_indexs[1]==min_y)[0]
    x_for_min_y_array = circle_indexs[0][x_index_for_min_y]
    x_for_min_y = np.mean(x_for_min_y_array)
    # print('min col = ', min_y, x_for_min_y)

    max_y = max(circle_indexs[1])
    x_index_for_max_y = np.where(circle_indexs[1]==max_y)[0]
    x_for_max_y_array = circle_indexs[0][x_index_for_max_y]
    x_for_max_y = np.mean(x_for_max_y_array)
    # print('max col = ', max_y, x_for_max_y)
```

```python
    point_1 = (y_for_min_x, min_x)
    point_2 = (y_for_max_x, max_x)
    point_3 = (min_y, x_for_min_y)
    point_4 = (max_y, x_for_max_y)
    # print(point_1, point_2, point_3, point_4)

    d_x_1 = point_1[0] - point_2[0]
    d_y_1 = point_1[1] - point_2[1]
    d_1 = math.sqrt(d_x_1**2 + d_y_1**2)
    # print("d1 = ", d_1)

    d_x_2 = point_3[0] - point_4[0]
    d_y_2 = point_3[1] - point_4[1]
    d_2 = math.sqrt(d_x_2**2 + d_y_2**2)
    # print("d2 = ", d_2)

    diameter = max(d_1, d_2)
    # print("diameter = ", diameter)
    radius = int(round(max(d_1, d_2)/2))
    # print("radius = ", radius)

    if diameter == d_1:
        x = int(round((point_1[0] + point_2[0] + 2) / 2))
        y = int(round((point_1[1] + point_2[1] + 2) / 2))
        center = (x, y)

    elif diameter == d_2:
        x = int(round((point_3[0] + point_4[0] + 2) / 2))
        y = int(round((point_3[1] + point_4[1] + 2) / 2))
        center = (x, y)

    # print("center = ", center)

    return center, radius
```

Dice performance of my test (train data size = 1000, epoch = 15):

```
 97%|█████████▋| 31/32 [00:11<00:00,  3.04it/s]Training loss: 0.958402244374156
100%|██████████| 32/32 [00:11<00:00,  2.84it/s]
100%|██████████| 32/32 [00:10<00:00,  3.10it/s]
  0%|          | 0/32 [00:00<?, ?it/s]Training loss: 0.9005990009754896
 97%|█████████▋| 31/32 [00:10<00:00,  3.04it/s]Training loss: 0.8042261581867933
100%|██████████| 32/32 [00:10<00:00,  3.11it/s]
 97%|█████████▋| 31/32 [00:10<00:00,  3.03it/s]Training loss: 0.6240626201033592
100%|██████████| 32/32 [00:10<00:00,  3.10it/s]
 97%|█████████▋| 31/32 [00:10<00:00,  3.04it/s]Training loss: 0.4704882763326168
100%|██████████| 32/32 [00:10<00:00,  3.10it/s]
100%|██████████| 32/32 [00:10<00:00,  3.10it/s]
  0%|          | 0/32 [00:00<?, ?it/s]Training loss: 0.23173785209655762
100%|██████████| 32/32 [00:10<00:00,  3.10it/s]
  0%|          | 0/32 [00:00<?, ?it/s]Training loss: 0.1427176669239998
100%|██████████| 32/32 [00:10<00:00,  3.09it/s]
  0%|          | 0/32 [00:00<?, ?it/s]Training loss: 0.126893674954772
100%|██████████| 32/32 [00:10<00:00,  3.09it/s]
Training loss: 0.10626217536628246
100%|██████████| 32/32 [00:10<00:00,  3.09it/s]
  0%|          | 0/32 [00:00<?, ?it/s]Training loss: 0.13154328987002373
 97%|█████████▋| 31/32 [00:10<00:00,  3.03it/s]Training loss: 0.08447357639670372
100%|██████████| 32/32 [00:10<00:00,  3.09it/s]
100%|██████████| 32/32 [00:10<00:00,  3.09it/s]
  0%|          | 0/32 [00:00<?, ?it/s]Training loss: 0.06870537996292114
100%|██████████| 32/32 [00:10<00:00,  3.09it/s]
  0%|          | 0/32 [00:00<?, ?it/s]Training loss: 0.07320396602153778
100%|██████████| 32/32 [00:10<00:00,  3.09it/s]
Training loss: 0.05407358519732952
 97%|█████████▋| 31/32 [00:10<00:00,  3.01it/s]Training loss: 0.0493210032582283
100%|██████████| 32/32 [00:10<00:00,  3.09it/s]
Test Accuracy: 0.919
```

Performance of main.py test:

```
Test Accuracy: 0.965
prediction:  (68, 139) 19
==============================
True circle:  (105, 98) 25
Test Accuracy: 0.940
prediction:  (106, 98) 26
==============================
True circle:  (60, 55) 38
Test Accuracy: 0.936
prediction:  (60, 53) 39
==============================
True circle:  (175, 185) 48
Test Accuracy: 0.918
prediction:  (162, 171) 46
==============================
True circle:  (40, 118) 45
Test Accuracy: 0.953
prediction:  (40, 118) 46
==============================
True circle:  (106, 195) 29
Test Accuracy: 0.881
prediction:  (104, 197) 32
==============================
True circle:  (97, 33) 37
Test Accuracy: 0.964
prediction:  (98, 32) 38
==============================
True circle:  (155, 192) 16
Test Accuracy: 0.897
prediction:  (154, 191) 17
==============================
True circle:  (85, 6) 34
Test Accuracy: 0.950
prediction:  (85, 4) 35
==============================
True circle:  (101, 101) 38
Test Accuracy: 0.973
prediction:  (100, 102) 39
==============================
iou mean =  0.904
```

My trained net (the weight file):
https://drive.google.com/open?id=188nbv9hnsKAVfPUKU-SC9te4uhMotTkm

2.3. IOU Optimization

```
100%|██████████| 2/2 [00:00<00:00,  2.59it/s]
  0%|          | 0/2 [00:00<?, ?it/s]Training loss: 0.9478632211685181
100%|██████████| 2/2 [00:00<00:00,  2.61it/s]
  0%|          | 0/2 [00:00<?, ?it/s]Training loss: 0.9453335106372833
100%|██████████| 2/2 [00:00<00:00,  2.64it/s]
  0%|          | 0/2 [00:00<?, ?it/s]Training loss: 0.9424173831939697
100%|██████████| 2/2 [00:00<00:00,  2.59it/s]
Training loss: 0.9396699368953705
Test Accuracy: 0.001
```

The result was really bad. The loss decreased very slow (barely moved).
This is because the IOU is not differentiable.

2.4. Visualization and error analysis

Visualization Code: (Please notice that I changed the “noisy_circle” function from main.py slightly, by returning an extra result “img” as the “mask” (i.e. clear circle image without noise). The original “img” is called “noisy_img” now.)

```python
def vis(detected, noisy_img, params):
    true_row, true_col, true_radius = params
    row, col, radius = detected
    img_seg = np.dstack([noisy_img*255, noisy_img*255, noisy_img*255]).astype("uint8")
    mask = np.zeros(noisy_img.shape, dtype=np.float)
    draw_circle(mask, row, col, radius)
    draw_circle(mask, true_row, true_col, true_radius)
    img_seg[mask > 0.5] = [255, 255, 0]
    return img_seg
```

```python
def noisy_circle(size, radius, noise):
    img = np.zeros((size, size), dtype=np.float)

    # Circle
    row = np.random.randint(size)
    col = np.random.randint(size)
    rad = np.random.randint(10, max(10, radius))
    draw_circle(img, row, col, rad)

    noisy_img = img.copy()

    # Noise
    noisy_img += noise * np.random.rand(*img.shape)
    return (row, col, rad), noisy_img, img

```

```python
def find_circle(img,mask):
    # Fill in this function
    noisy_img = [transform(img)]
    mask_img = [transform(mask)]
    size = 1

    test_data = MyDataset(noisy_img, mask_img, size)
    testloader = torch.utils.data.DataLoader(test_data, batch_size=1)

    find_circle_net = UNet()
    if torch.cuda.is_available():
        find_circle_net = find_circle_net.cuda()
    find_circle_net.load_state_dict(torch.load("./drive/My Drive/csc420/q2_net_dice_1000_15.pt"))

    test_result = test(find_circle_net, testloader)
    center, radius = detect_circle(test_result[0][0].cpu().squeeze().detach().numpy())
    print("prediction: ", center, radius)
    # print("==========================================================================================")
    return center[1], center[0], radius

```

```python
def main():
    results = []
    for _ in range(10):
        params, noisy_img, mask = noisy_circle(200, 50, 2)
        print("====================================================
        print("True circle: ", (params[1], params[0]), params[2])
        detected = find_circle(noisy_img, mask)
        seg = vis(detected, noisy_img, params)
        plt.imshow(seg, cmap="gray")
        plt.show()
        print("iou = ", iou(params, detected))
        print("====================================================
        results.append(iou(params, detected))
    results = np.array(results)
    print("iou mean = ", (results > 0.7).mean())

main()
```

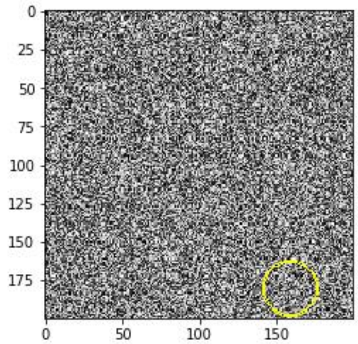
Results:

Notes:

1.The (0,0) point is at the top-left corner.

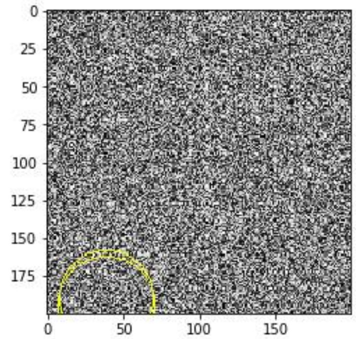
2.I printed the centers and radiuses of true and predicted circles as reference.

```
True circle:  (159, 180) 17
prediction:  (159, 180) 18
```



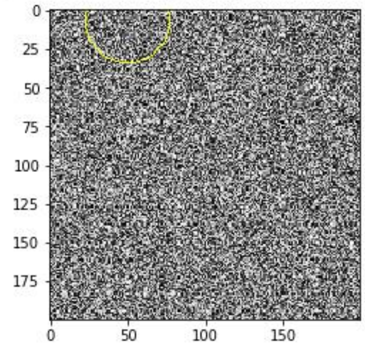
```
iou =  0.8919753086419752
```

```
True circle:  (39, 189) 31
prediction:  (39, 194) 32
```



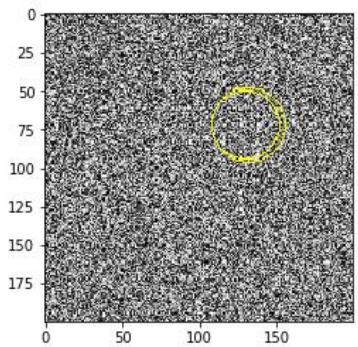
```
iou =  0.8131989065575698
```

```
True circle:  (50, 7) 27
prediction:  (50, 7) 27
```



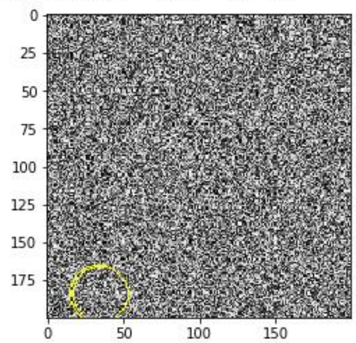
```
iou =  1.0
```

```
True circle:  (130, 72) 22
prediction:  (132, 72) 24
```



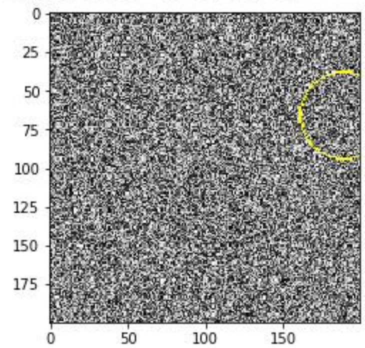
```
iou =  0.8402777777777778
```

```
True circle:  (35, 184) 18
prediction:  (34, 184) 19
```



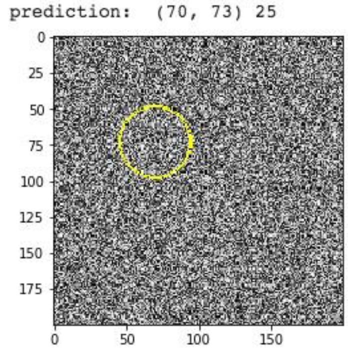
```
iou =  0.8975069252077565
```

```
True circle:  (189, 66) 28
prediction:  (190, 66) 28
```



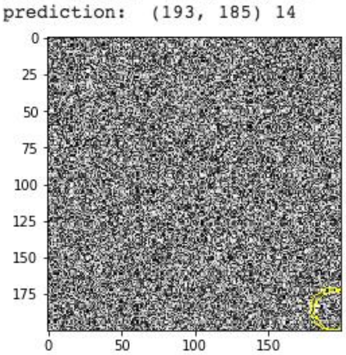
```
iou =  0.955487251646524
```

```
True circle:  (70, 73) 24
prediction:  (70, 73) 25
```



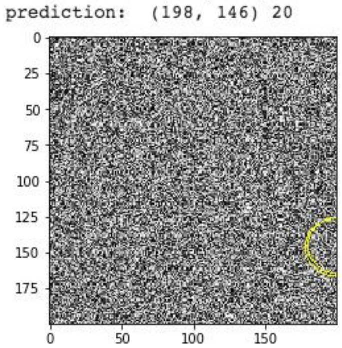
```
iou =  0.9215999999999998
```

```
True circle:  (195, 186) 13
prediction:  (193, 185) 14
```



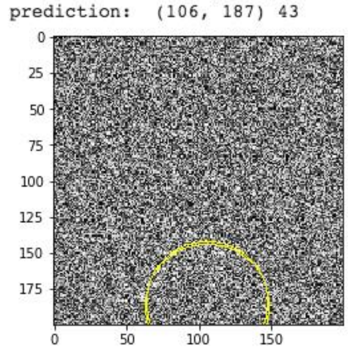
```
iou =  0.7921017625069104
```

```
True circle:  (199, 145) 19
prediction:  (198, 146) 20
```



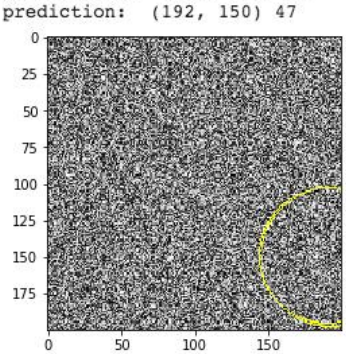
```
iou =  0.8898558326044305
```

```
True circle:  (106, 184) 42
prediction:  (106, 187) 43
```



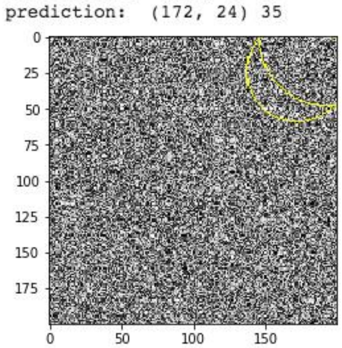
```
iou =  0.9093599389412192
```

```
True circle:  (190, 149) 46
prediction:  (192, 150) 47
```



```
iou =  0.9347081309046523
```

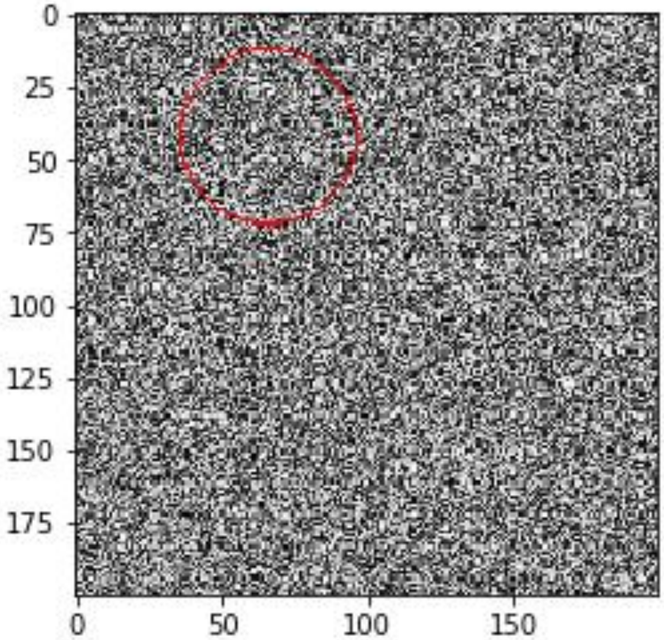
```
True circle:  (193, 1) 47
prediction:  (172, 24) 35
```



```
iou =  0.33314843194337573
```

This is an example of perfect detection.

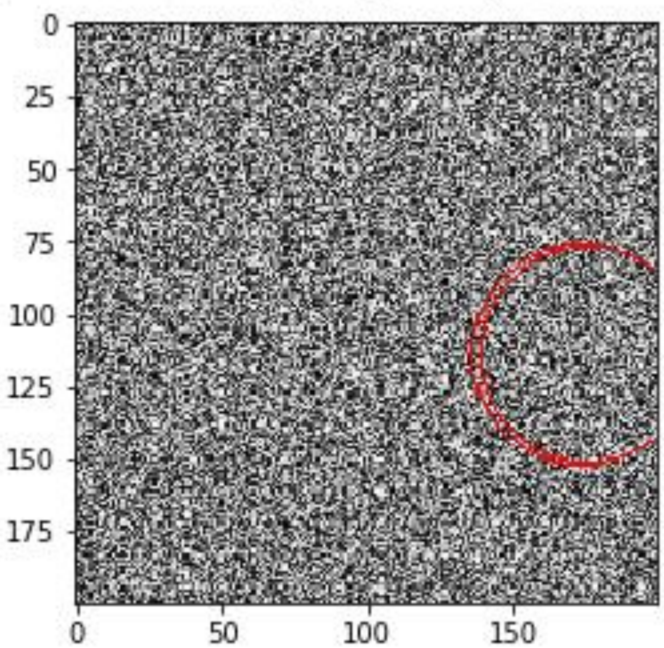
```
==================================
True circle:  (66, 42) 30
prediction:  (66, 42) 30
```




```
iou =  1.0
==================================
```

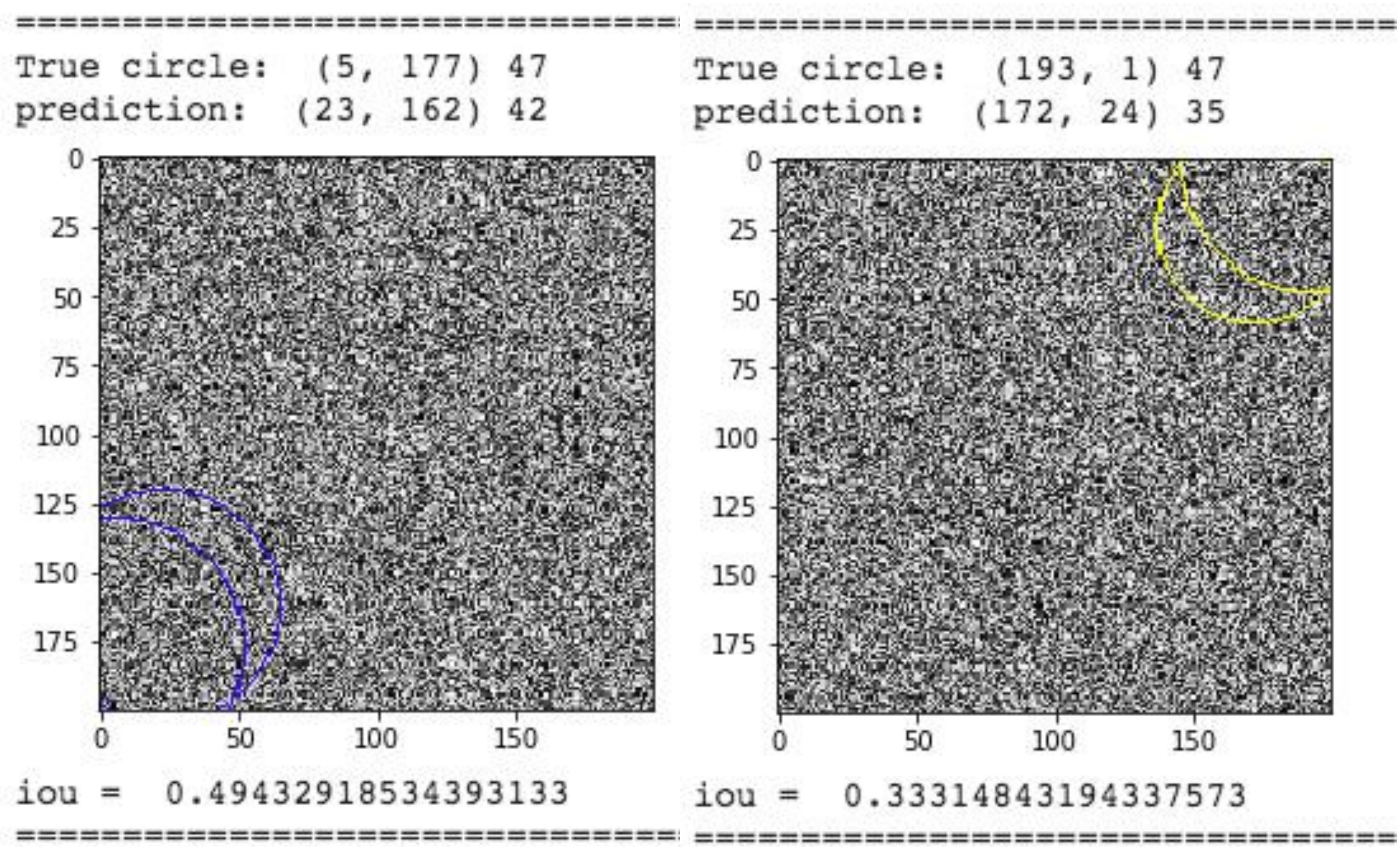
This is an example of almost perfect detection. Even though the mismatch is little, you can see the predicted center is not perfectly match the true center, and the predicted radius is not equal to the real radius too. This is because when I was implementing my “detect_circle” function, I applied the “round()” and “int()” to center points and radius. Doing this can help me return a reasonable integer result. But sometime it may lead to a mismatch.

```
==================================
True circle:  (176, 114) 37
prediction:  (174, 114) 38
```




```
iou =  0.9262091897853147
==================================
```

These are examples of bad detection. My detection method is hard to detect the circle which is not completed (i.e. semi-circles that were cut by edges). My “detect_circle” is designed to find the extreme points of the circle to determine the diameter and the center. For these cases, the extreme points on the circles cannot find the correct diameters, which leads to the wrong centers too.



3. Hot Dog or Not Hot Dog

The output (i.e. a single scalar c) is a boundary (i.e. threshold) of the given binary classifier.
After the training, the model will generate a single scalar c. Let’s assume that “larger or equal to c” means “it is a hot dog”, “less than c” means “it is not a hot dog” (this is designed by the programmer of the net). When input your picture to the model, if the output of your image is larger or equal to c, then it will be classified to “hot dog “by the classifier. If your output is less than c, then it will be classified to “not hot dog”.
So the value of c doesn’t matter. The key is we need to know the input will be larger, equal to, or less than c.