1. The implementation of the UNet, BCE Loss(with logistics), and Sorensen Dice Loss are shown below. I add 1 padding for each convolution to avoid the size to be an odd number.

Related Files:
cat_dataset.py
my_BCE_loss.py
sorensen_dice_coefficient_loss.py
unet.py
train_and_test_1_1.py
train_and_test_util_1.py

UNet

```
class UNet(nn.Module):

    def __init__(self, in_channels=1, out_channels=1):
        super(UNet, self).__init__()
        self.downconv1 = self.build_conv(in_channels, 64)
        self.downconv2 = nn.Sequential(nn.MaxPool2d(2, stride=2), self.build_conv(64, 128))
        self.downconv3 = nn.Sequential(nn.MaxPool2d(2, stride=2), self.build_conv(128, 256))
        self.downconv4 = nn.Sequential(nn.MaxPool2d(2, stride=2), self.build_conv(256, 512))
        self.midconv = nn.Sequential(nn.MaxPool2d(2, stride=2), self.build_conv(512, 1024),
                                     nn.ConvTranspose2d(in_channels=1024, out_channels=512, kernel_size=2,
stride=2),
                                     )

        self.upconv1 = nn.Sequential(self.build_conv(1024, 512),
                                     nn.ConvTranspose2d(in_channels=512, out_channels=256, kernel_size=2,
stride=2),
                                     )
        self.upconv2 = nn.Sequential(self.build_conv(512, 256),
                                     nn.ConvTranspose2d(in_channels=256, out_channels=128, kernel_size=2,
stride=2),
                                     )
        self.upconv3 = nn.Sequential(self.build_conv(256, 128),
                                     nn.ConvTranspose2d(in_channels=128, out_channels=64, kernel_size=2,
stride=2),
                                     )
        self.upconv4 = nn.Sequential(self.build_conv(128, 64),
                                     nn.Conv2d(in_channels=64,
                                               out_channels=out_channels,
                                               kernel_size=1)
                                    )
        self.final = nn.Sigmoid()
```

```
    def forward(self, x):
        self.h1 = self.downconv1(x)
        self.h2 = self.downconv2(self.h1)
        self.h3 = self.downconv3(self.h2)
        self.h4 = self.downconv4(self.h3)
        self.h5 = self.midconv(self.h4)
        self.h6 = self.upconv1(torch.cat([self.h5, self.h4], 1))
        self.h7 = self.upconv2(torch.cat([self.h6, self.h3], 1))
        self.h8 = self.upconv3(torch.cat([self.h7, self.h2], 1))
        self.h9 = self.upconv4(torch.cat([self.h8, self.h1], 1))
        self.out = self.final(self.h9)
        return self.out

    def build_conv(self, in_channels, out_channels):
        return nn.Sequential(nn.Conv2d(in_channels=in_channels,
                                       out_channels=out_channels,
                                       kernel_size=3, padding=1),
                             nn.ReLU(),
                             nn.BatchNorm2d(out_channels),
                             nn.Conv2d(in_channels=out_channels,
                                       out_channels=out_channels,
                                       kernel_size=3, padding=1),
                             nn.ReLU(),
                             nn.BatchNorm2d(out_channels)
                             )
```

BCE Loss

```
class My_BCELoss(nn.Module):

    def __init__(self):
        super(My_BCELoss, self).__init__()

    def forward(self, Y_pred, Y):
        Y_pred_flat = Y_pred.contiguous().view(-1).float()
        Y_flat = Y.contiguous().view(-1).float()

        bceloss = (torch.sum(Y_flat * torch.log(1 + torch.exp(-Y_pred_flat)) \
                + (1-Y_flat) * torch.log(1 + torch.exp(Y_pred_flat))))\
                     / (Y_flat.size()[0])

        return bceloss
```

Sorensen Dice Loss

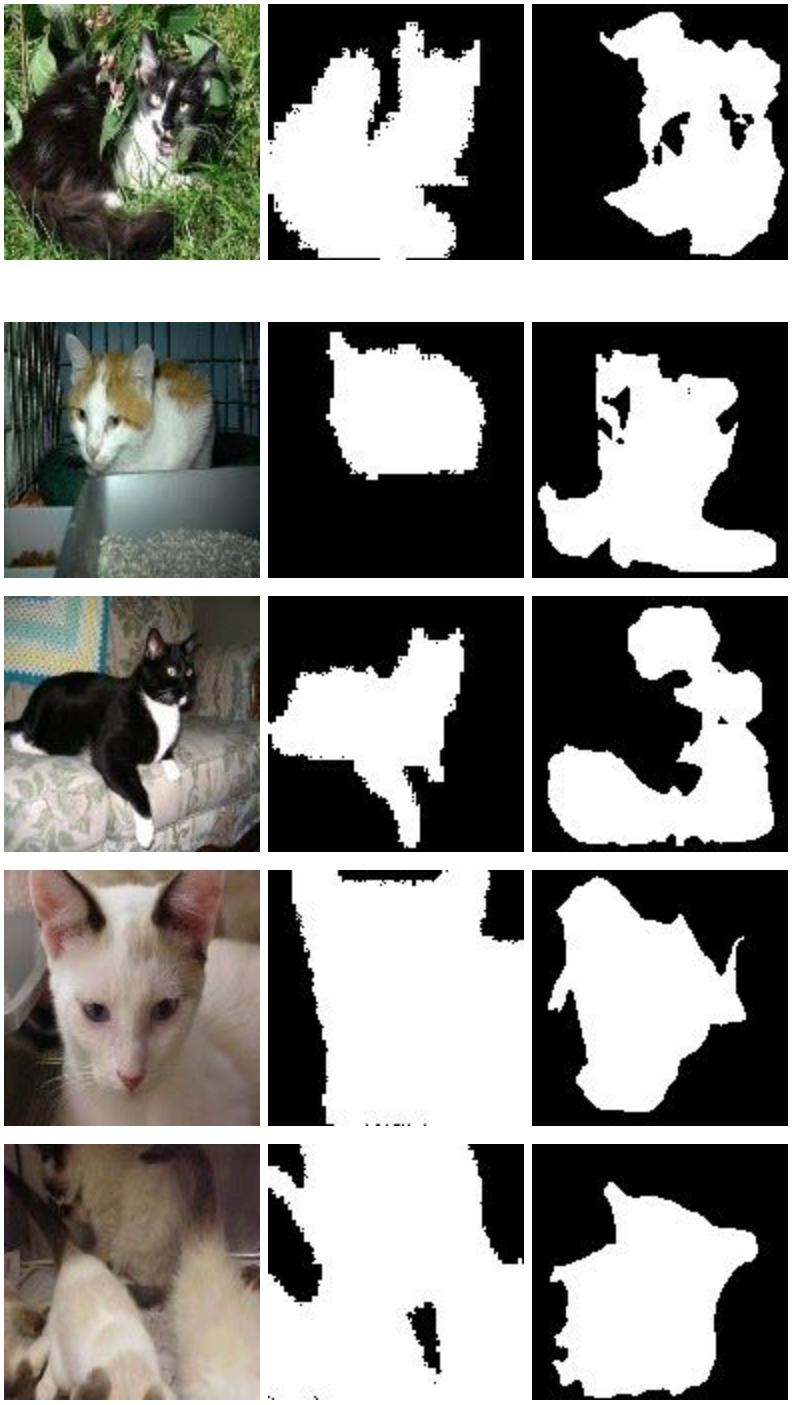
```
class SorensenDiceCoefficientLoss(nn.Module):

    def __init__(self):
        super(SorensenDiceCoefficientLoss, self).__init__()

    def forward(self, Y_pred, Y):
```

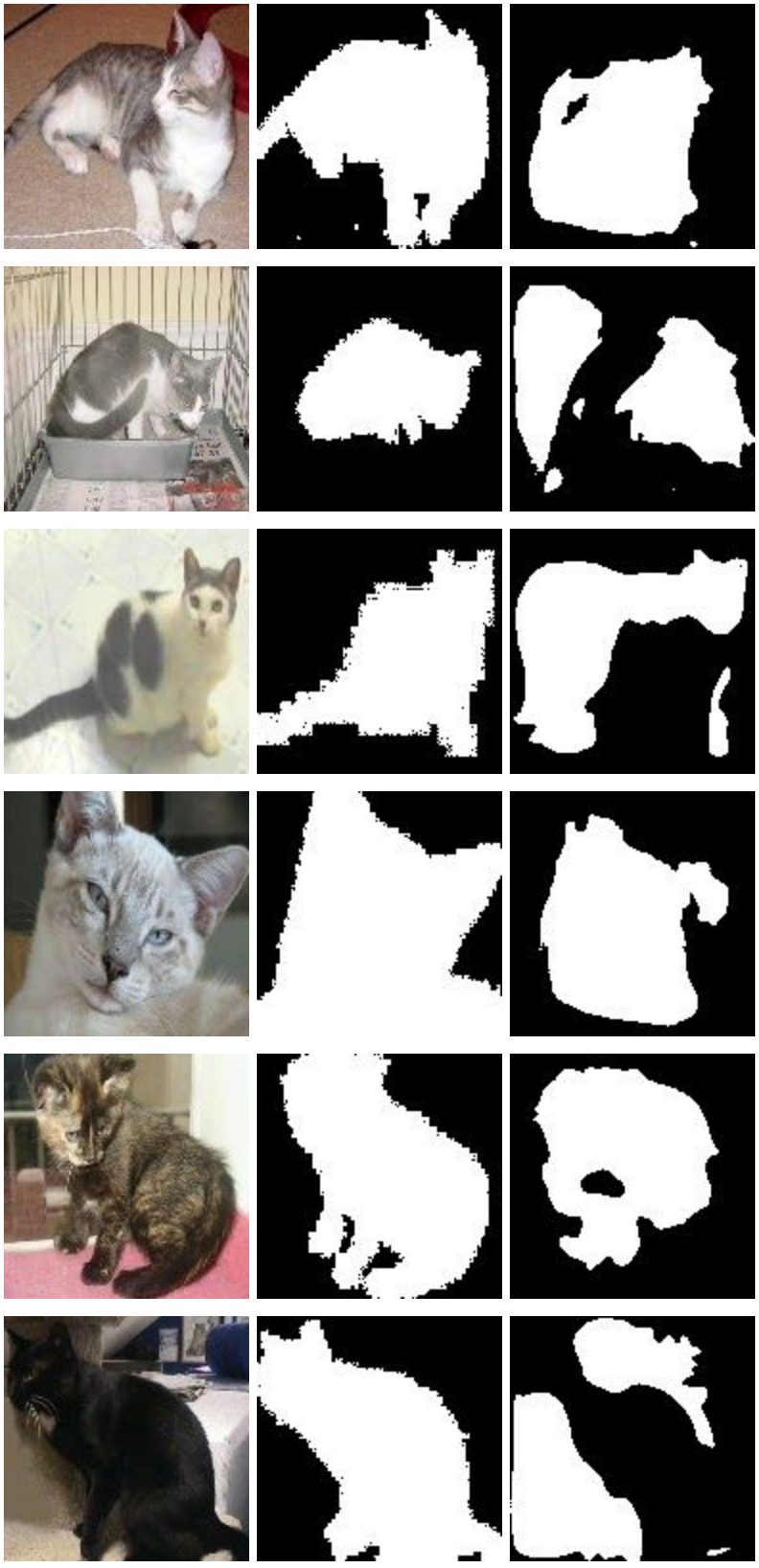
```
        smooth = 1
        Y_pred_flat = Y_pred.contiguous().view(-1).float()
        Y_flat = Y.contiguous().view(-1).float()

        intersection = Y_pred_flat * Y_flat

        A = Y_pred_flat ** 2
        B = Y_flat ** 2

        return 1 - (2 * torch.sum(intersection) + smooth)\
                /(torch.sum(A) + torch.sum(B) + smooth)
```

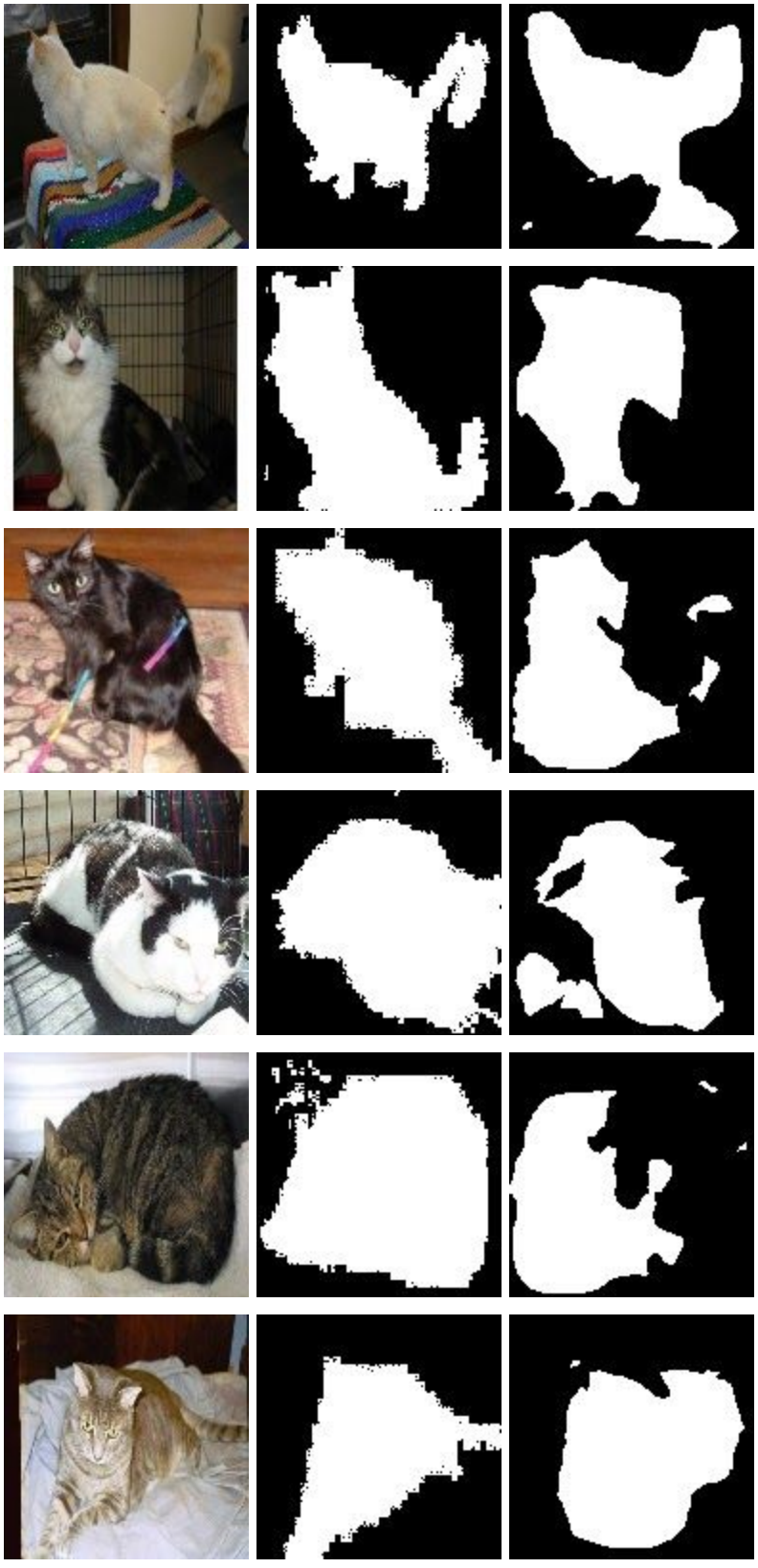
Here are the output of training loss and testing accuracy for BCE Loss, and Sorensen Dice Loss, respectively for 10 epochs with batch size 10, along with original image, true masks, and predicted masks.

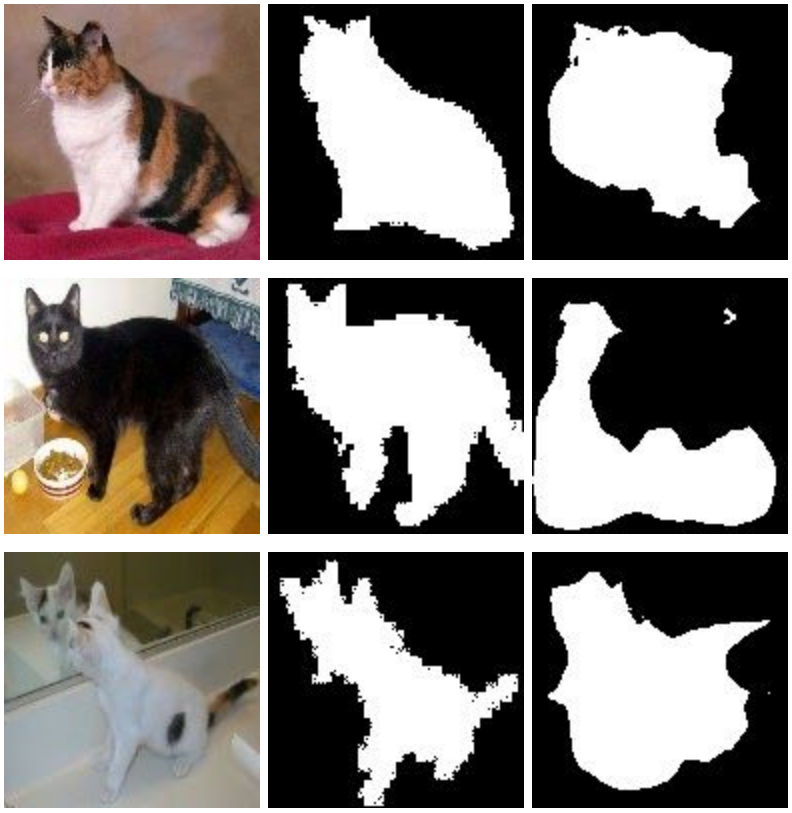
BCE Loss

```
Epoch 0, loss 0.7230752309163412
Epoch 1, loss 0.6801776885986328
Epoch 2, loss 0.6501181324323019
Epoch 3, loss 0.6322247385978699
Epoch 4, loss 0.6194920639197031
Epoch 5, loss 0.6066960493723551
Epoch 6, loss 0.6085568964481354
Epoch 7, loss 0.6069586376349131
Epoch 8, loss 0.6015532612800598
Epoch 9, loss 0.5881412724653879
Test Loss 0.6604264151482355
Test Accuracy 0.3395735848517645
```



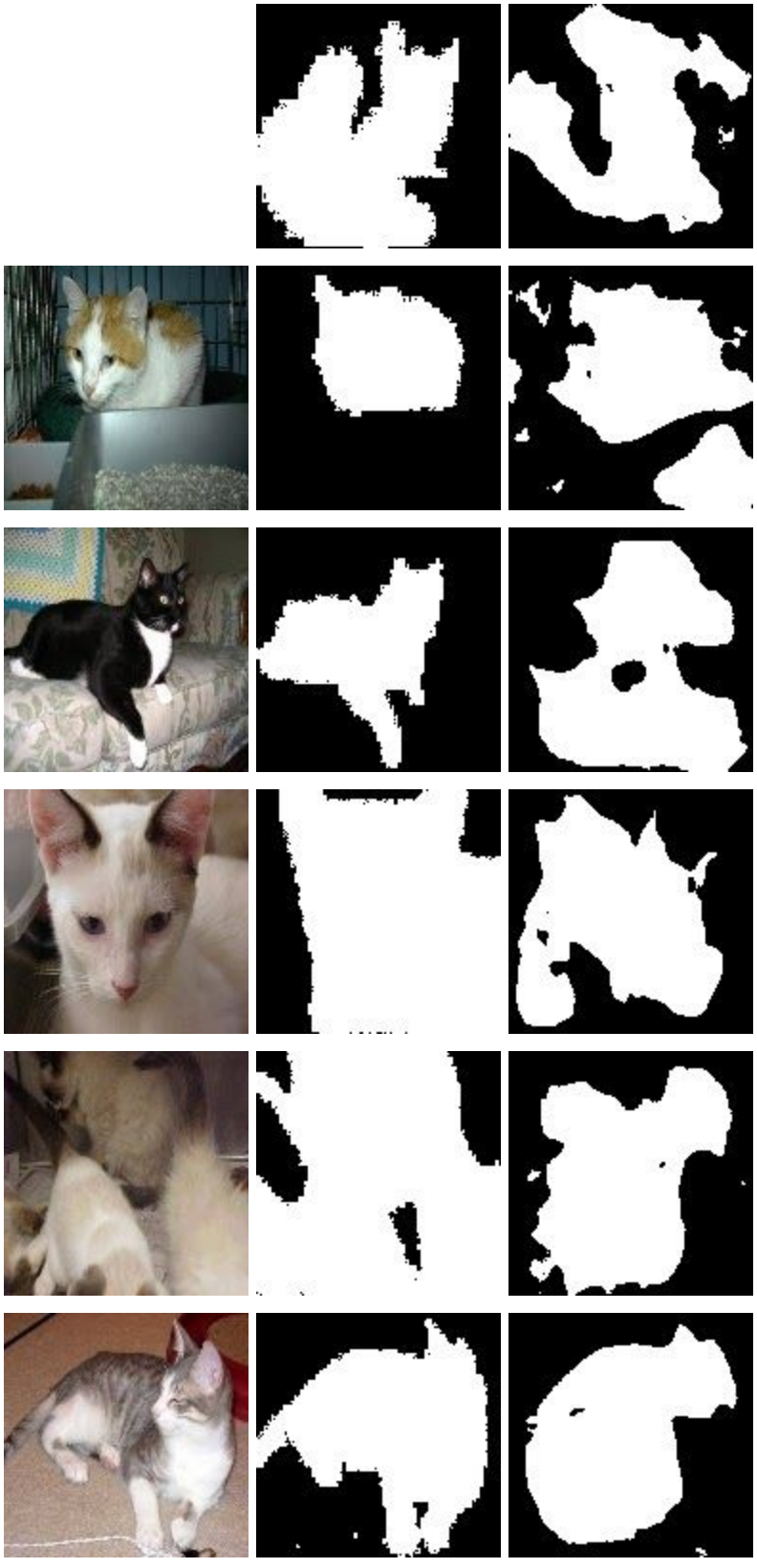


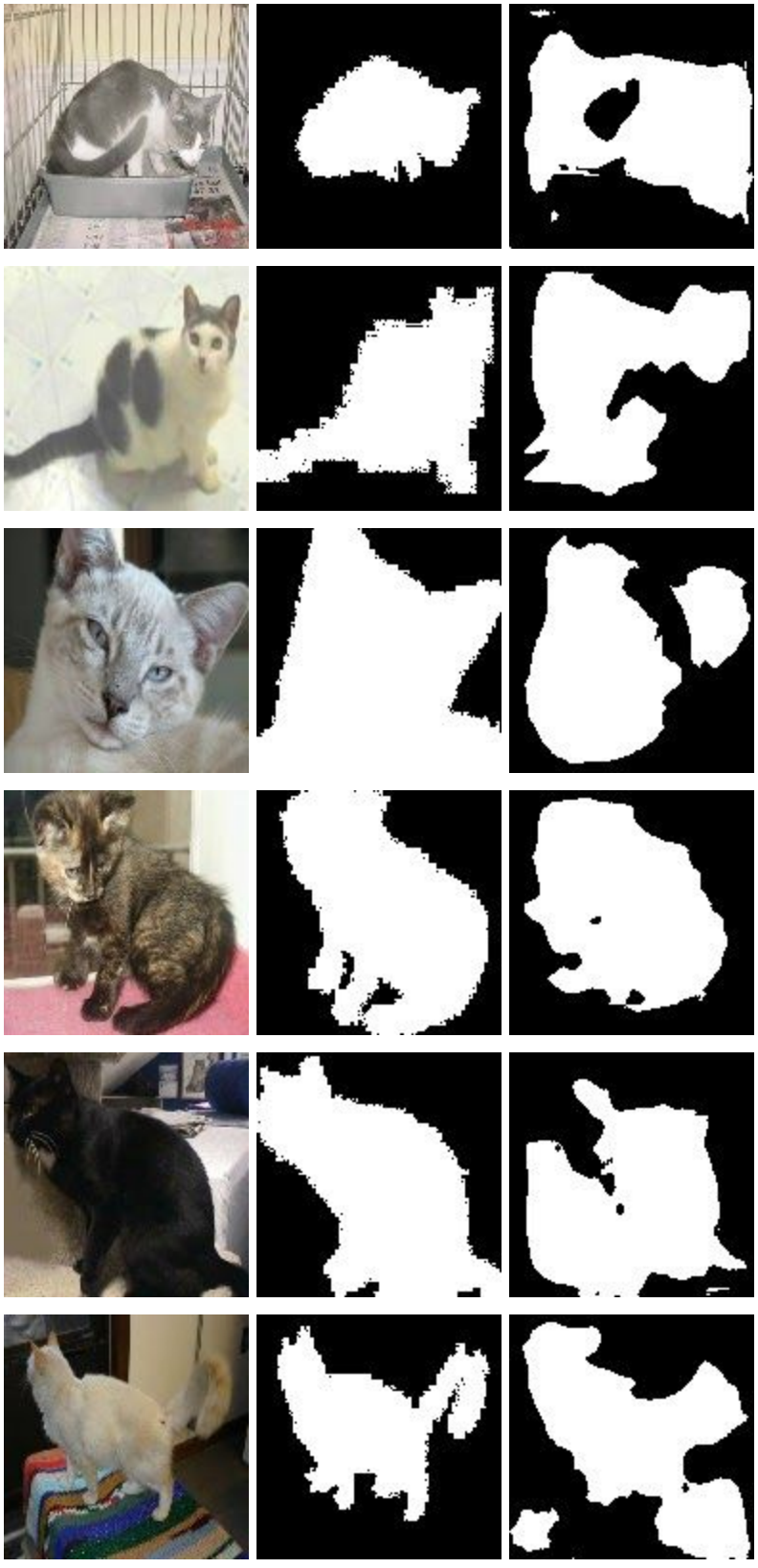


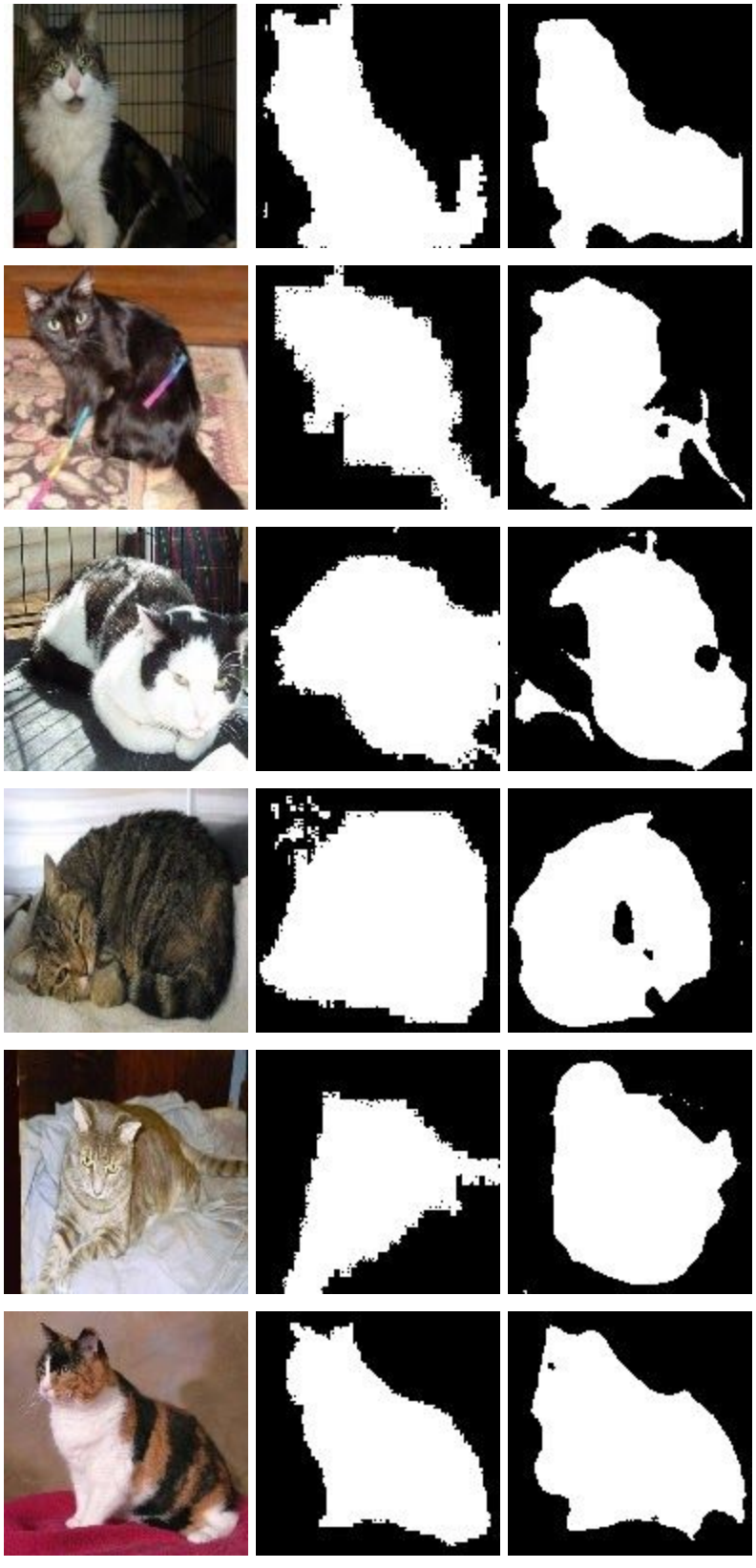


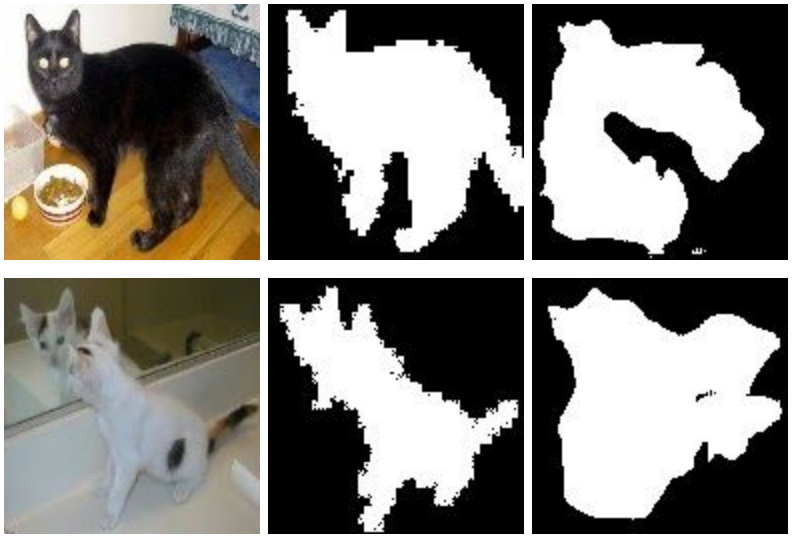
Sorensen Dice

```
Epoch 0, loss 0.3379280169804891
Epoch 1, loss 0.2620130280653636
Epoch 2, loss 0.2202739119529724
Epoch 3, loss 0.19520193338394165
Epoch 4, loss 0.18086869517962137
Epoch 5, loss 0.16739150881767273
Epoch 6, loss 0.15441476305325827
Epoch 7, loss 0.14623952905337015
Epoch 8, loss 0.14700671037038168
Epoch 9, loss 0.14037537574768066
Test Loss 0.2857369241260347
Test Accuracy 0.7142630758739653
```









By comparison, the Sorensen Dice Loss performs better than the BCE Loss with logistic, the reason is the BCE loss function mostly don't have sudden change from the function (i.e. The derivative is quite low), do the loss go down much slower than the dice loss compared to Sorensen Dice Loss. Another reason why Sorensen Dice Loss is better is because the goal for this cat segmentation prediction is to maximize the sorensen dice coefficient, which is equivalent to minimize the sorensen dice loss, and it can also use backpropagation to optimize.

2. The implementation of augmentation is shown below, which are flip, rotate, shift and change saturations
Related Files:
cat_dataset.py
sorensen_dice_coefficient_loss.py
unet.py
train_and_test_1_2.py
train_and_test_util_1.py

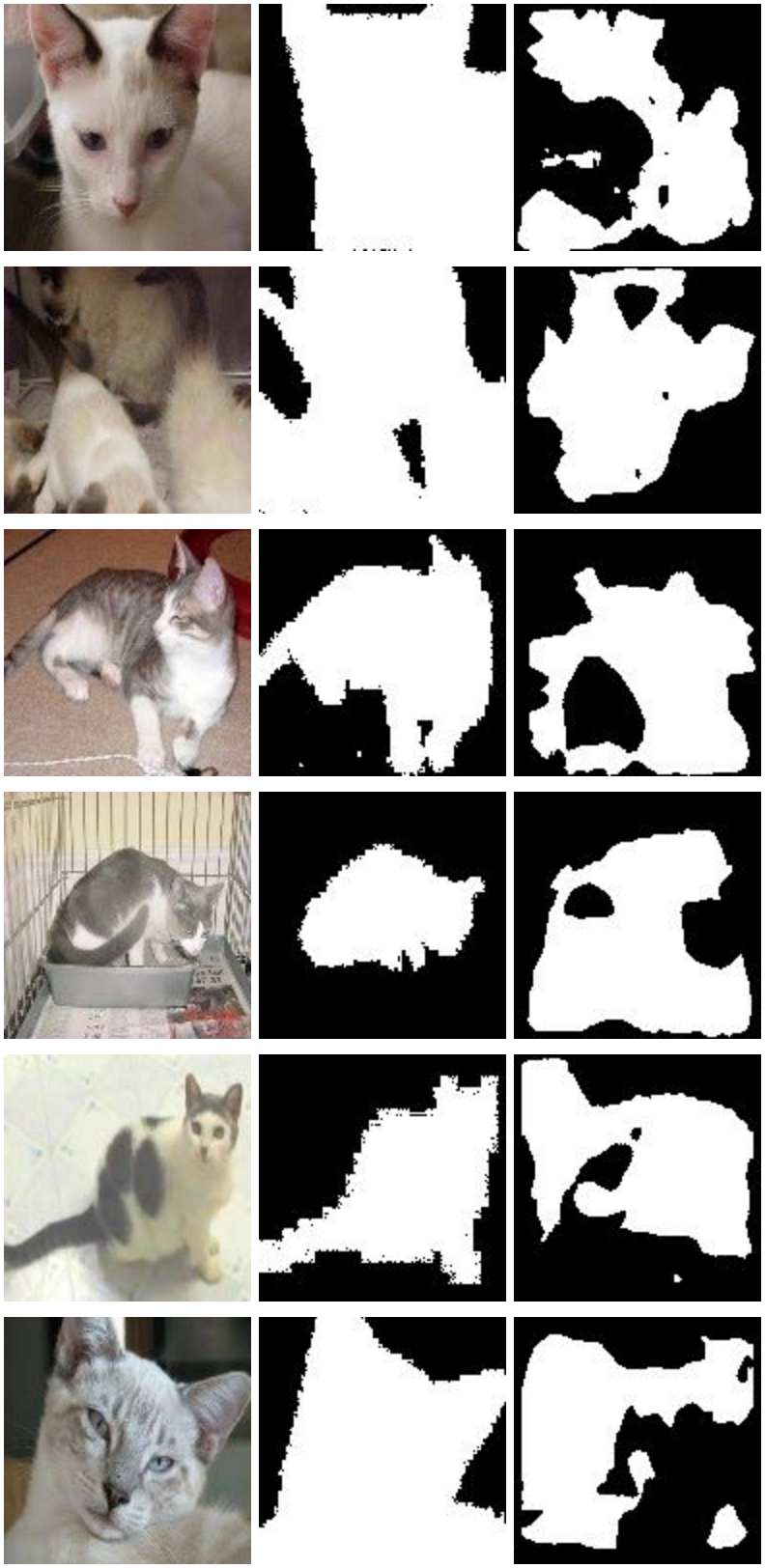
```
def flip(cat, mask):
    flip_direction = np.random.randint(3) - 1
    cat_fliped = cv2.flip(cat, flip_direction)
    mask_fliped = cv2.flip(mask, flip_direction)
    return cat_fliped, mask_fliped


def rotate(cat, mask):
```

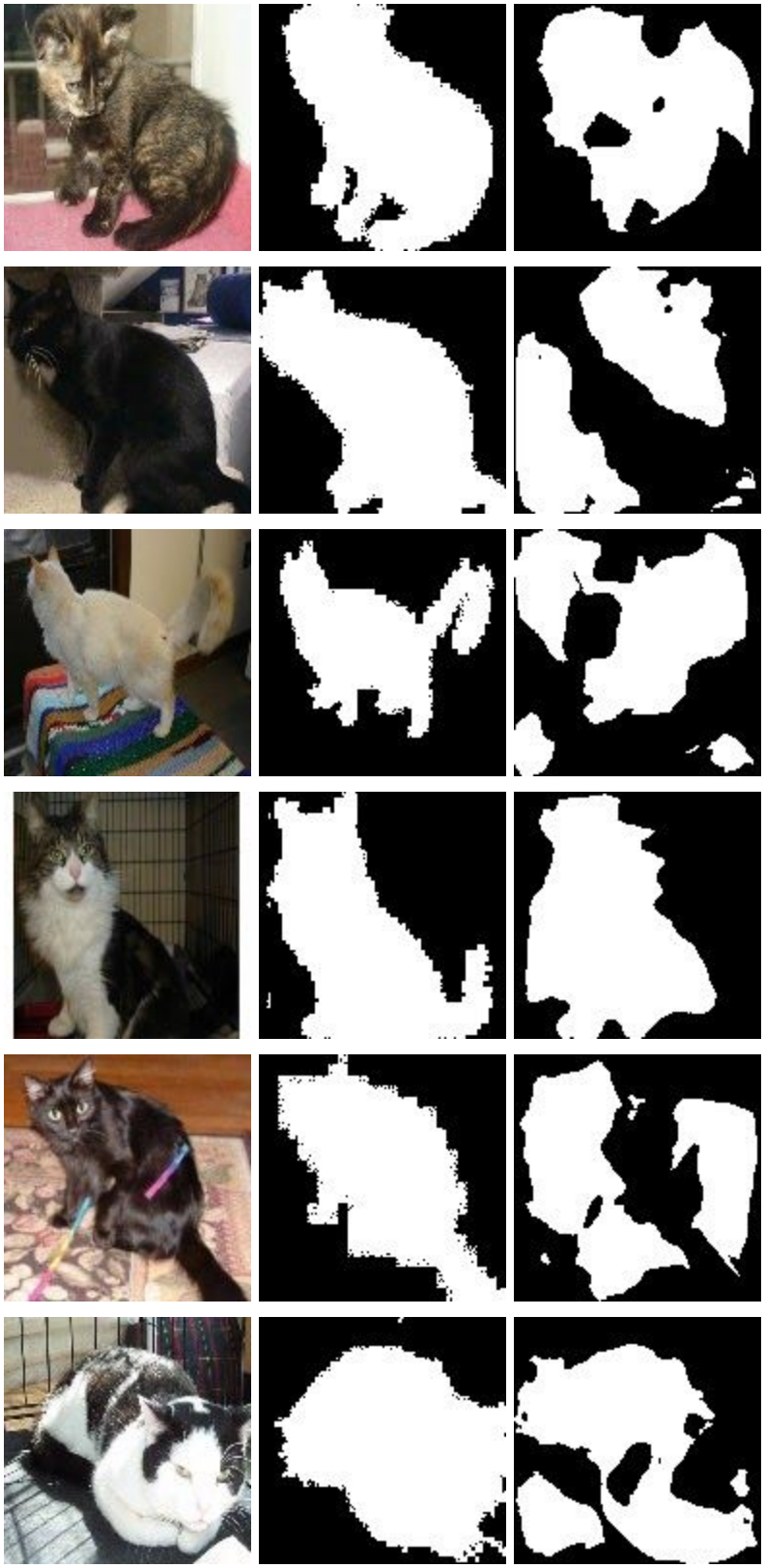
```
    rotate_degree = np.random.randint(-90, 90)

    centre = np.array([cat.shape[0], cat.shape[1]]) / 2.0
    rot = cv2.getRotationMatrix2D(tuple(centre), rotate_degree,1.0)

    cat_rotated = cv2.warpAffine(cat, rot, (cat.shape[1],
cat.shape[0]),flags=cv2.INTER_LINEAR)
    mask_rotated = cv2.warpAffine(mask, rot, (cat.shape[1], cat.shape[0])
,flags=cv2.INTER_LINEAR)

    return cat_rotated, mask_rotated

def shift(cat, mask):
    shift_unit = (np.random.randint(-cat.shape[0]//4, cat.shape[0]//4),\
                                    np.random.randint(-cat.shape[1]//4, cat.shape[1]//4) )
    M = np.float32([[1,0,shift_unit[0]], [0,1,shift_unit[1]]])
    cat_shifted = cv2.warpAffine(cat, M, (cat.shape[1], cat.shape[0]))
    mask_shifted = cv2.warpAffine(mask, M, (cat.shape[1], cat.shape[0]))
    return cat_shifted, mask_shifted


def change_saturation(cat, mask):
    hsv = cv2.cvtColor(cat, cv2.COLOR_BGR2HSV).astype(np.float64)

    hsv[:, :, 1] *= np.random.uniform(0.8, 1.2)

    hsv = np.clip(hsv, 0, 255).astype(np.uint8)

    cat_changed = cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)

    return cat_changed, mask


AUGMENTATION_FUNCTIONS = [flip, rotate, shift, change_saturation]
```

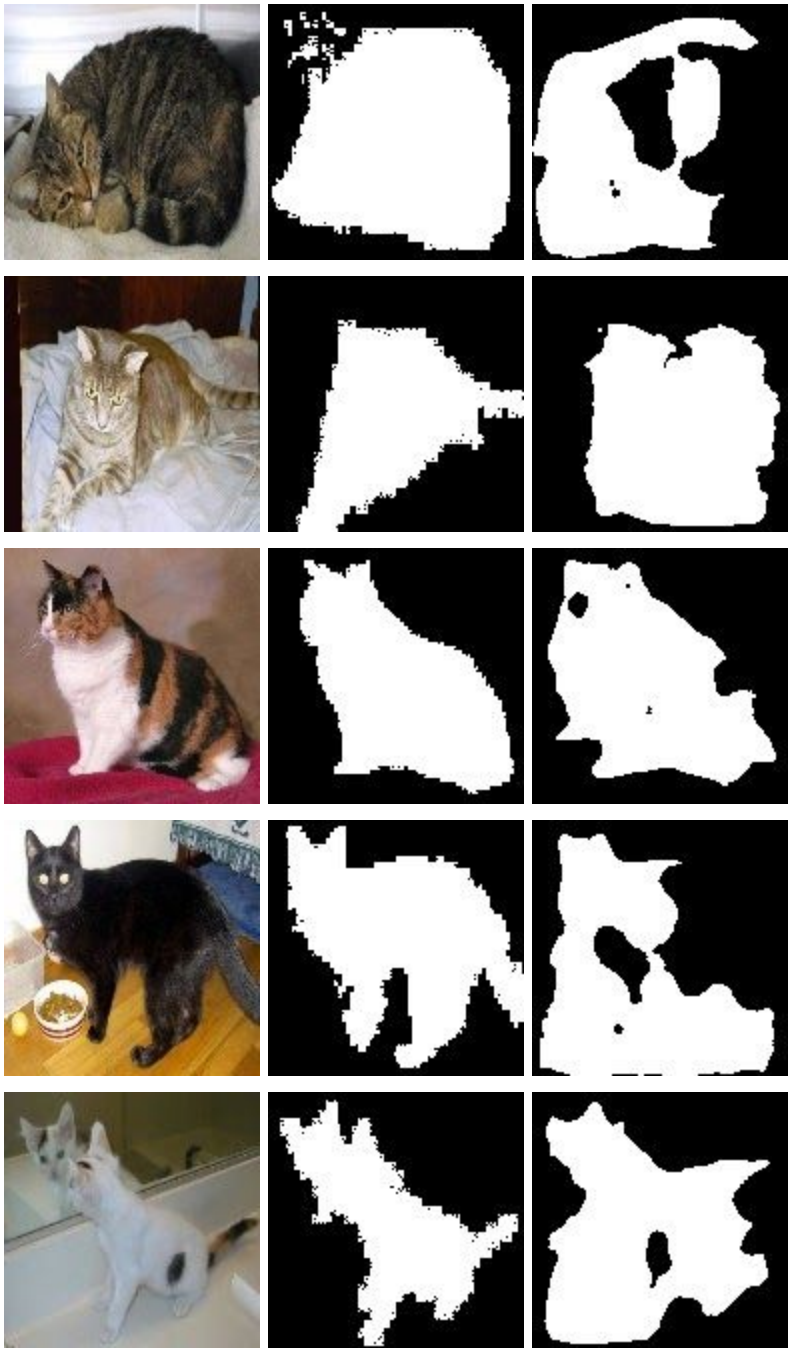
The following is the output and training losses, and test loss, and test accuracy after training with dice loss and augmented datasets

```
Epoch 0, loss 0.32938729226589203
Epoch 1, loss 0.23417545358339945
Epoch 2, loss 0.19717836876710257
Epoch 3, loss 0.17094712456067404
Epoch 4, loss 0.16215930879116058
Epoch 5, loss 0.16091999411582947
Epoch 6, loss 0.1544723610083262
Epoch 7, loss 0.14331265787283579
Epoch 8, loss 0.14356053372224173
Epoch 9, loss 0.13559950391451517
Test Loss 0.2675212025642395
Test Accuracy 0.7324787974357605
```

The following are list of original images, true masks, and predicted masks respectively







I do the data augmentation by random choose one function for each image in the training data. The total size of training set is doubled, it is a little better than the one without augmentation, the reason is the augmentation is still based on the same image, so the features are the same. The neural net will learn by these features specifically and may cause overfit, which make the test loss not as low as expected.

3. I use the dataset from the following link to do transfer learning, this dataset contain other pets.

https://www.robots.ox.ac.uk/~vgg/data/pets/

I do the transfer learning by following steps
1. Get the data from the link above
2. Convert trimap to binary mask, because trimap setting 1, 2, 3, as foreground, background, and object, which is not binary mask
3. Train the UNet(with batch size 30 and 20 epoches) by the whole transfer dataset as training set which I got online with other different kinds of pets with different kinds of variety (get unet_1_3_pretrained.pt file in the shared folder), that will learn a lot about the pattern of the pets, the reason why the other kind of pets like dogs are also a good choice is because the size and shape of the contour of dogs are similar to cat.
4. Train the UNet by own training set in given cat_dataset(same as Q1.1, with batch size 15 and 10 epoches)
5. Test the Unet by own test set in given cat_dataset (get unet_1_3.pt file in share folder)

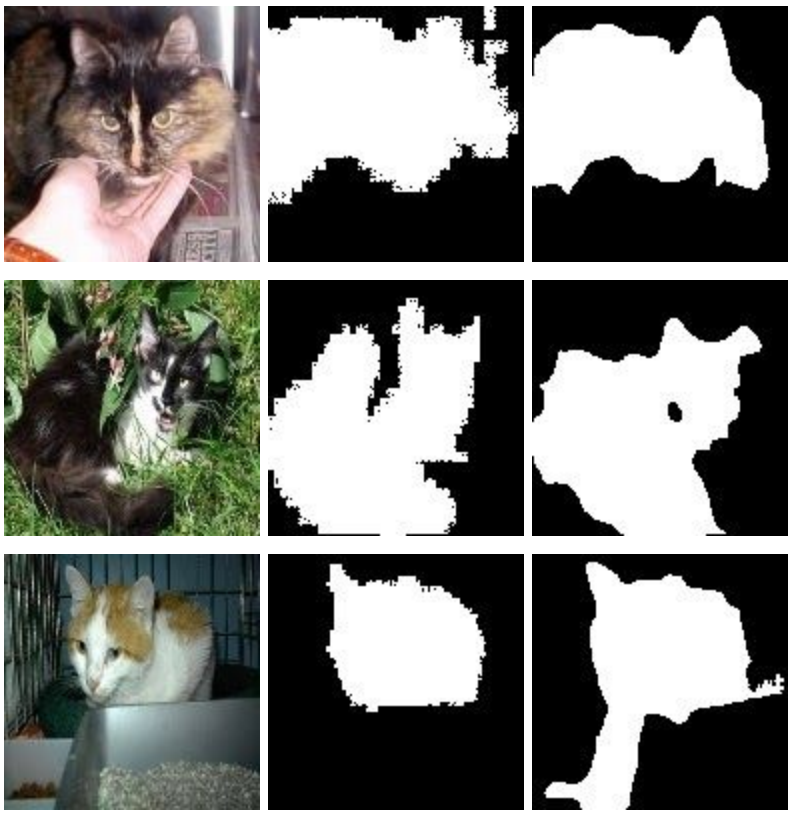
Related Files:
transfer_dataset.py
sorensen_dice_coefficient_loss.py
unet.py
train_and_test_1_3.py
train_and_test_util_1.py

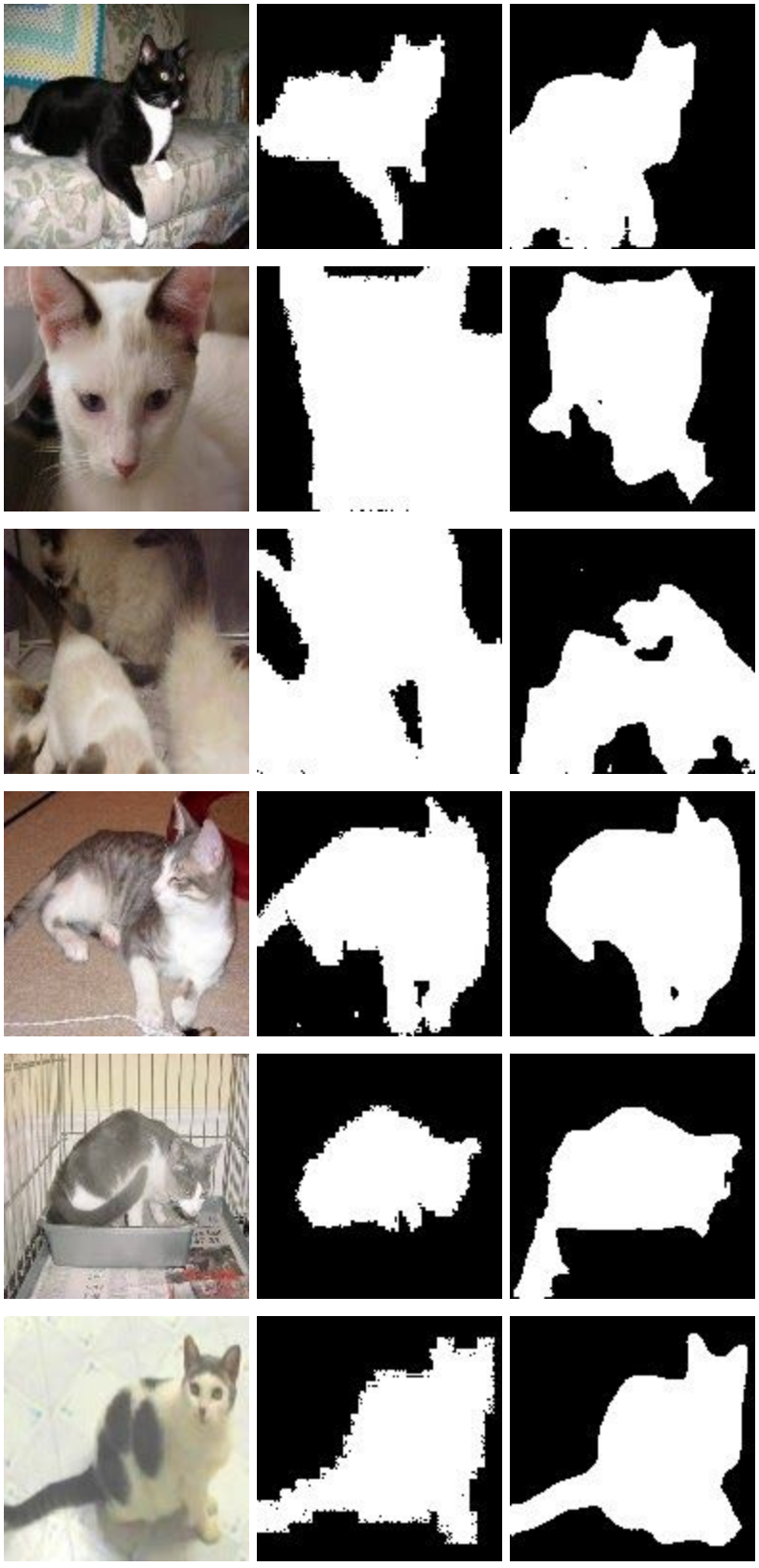
The link of the shared folder
https://drive.google.com/drive/folders/1CERB9Jk_ifLEYOSQmky_0pz0HzBXiHnF?usp=sharing

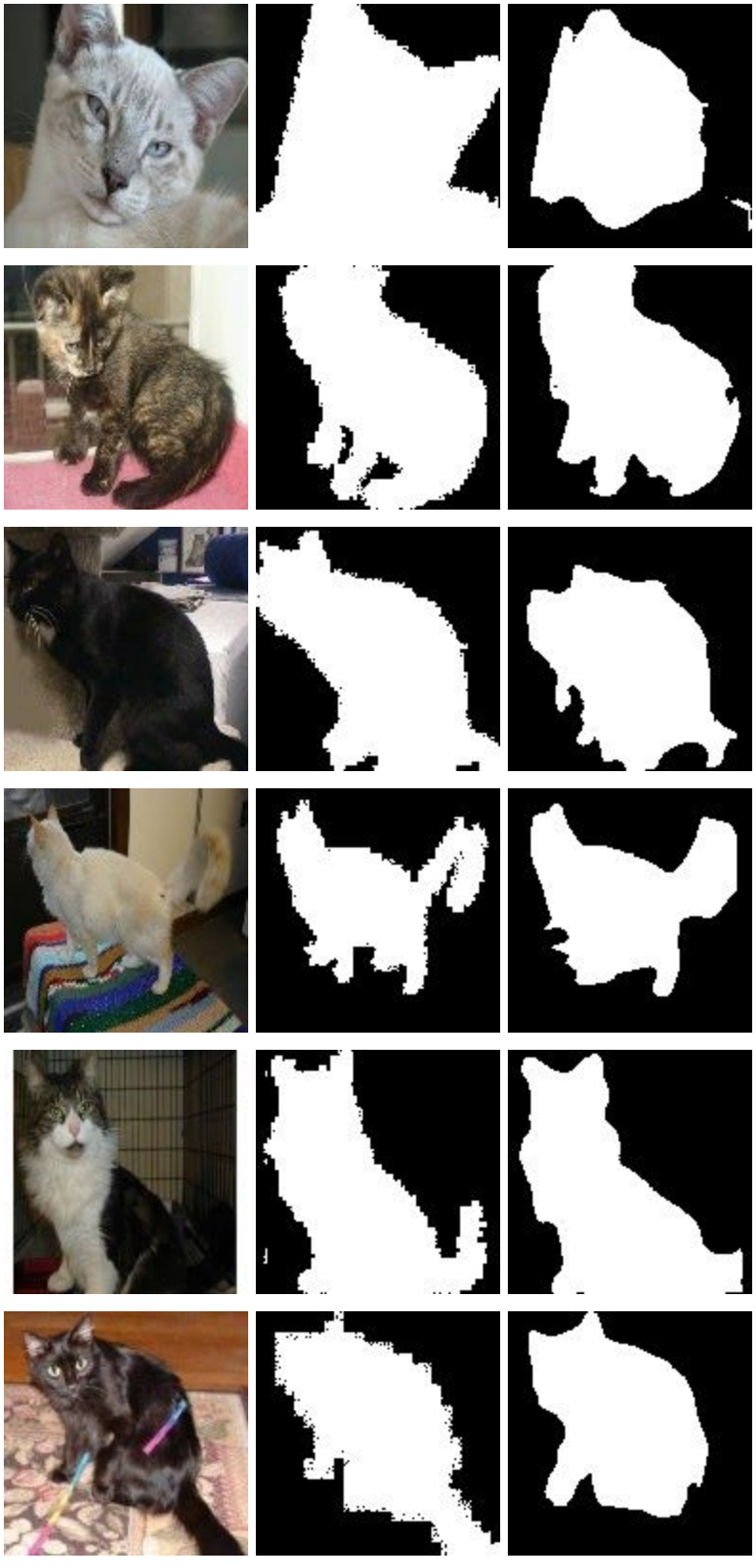
Here is the output of the training and testing process for a pretrained model

```
Epoch 0, loss 0.14913217723369598
Epoch 1, loss 0.0838160514831543
Epoch 2, loss 0.06767365336418152
Epoch 3, loss 0.05857586860656738
Epoch 4, loss 0.050029829144477844
Epoch 5, loss 0.044657379388809204
Epoch 6, loss 0.040373384952545166
Epoch 7, loss 0.03642728924751282
Epoch 8, loss 0.033582672476768494
Epoch 9, loss 0.03152108192443848
Test Loss 0.115746228467850 46
Test Accuracy 0.8842537715321496
```

The following are list of original images, true masks, and predicted masks, respectively







4.

Here is the implementation

```
def draw_contour(img, mask):

    mask_cpy = (mask * 255).astype(np.uint8)

    img_cpy = img

    mask_cpy[mask_cpy < 255/2.0] = 0

    mask_cpy[mask_cpy >= 255/2.0] = 255

    edges = cv2.Canny(mask_cpy,50,200)

    img_cpy[np.where(edges != 0)] = np.array([0,255,0])

    return img_cpy
```

Related File:
draw_contour.py

Here is the output