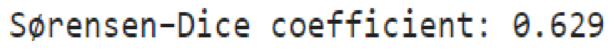
CSC420 Assignment 3
Xinqi Shen

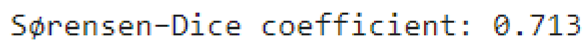
1.1 I use two loss functions in this question, **Dice loss** (Sørensen Dice) and **BCE loss** (Binary Cross Entropy). Both them are reasonable for the task of segmentation since the image segmentation can be considered as a classification problem (predict foreground and background). I will compare two loss function based on Sørensen Dice coefficient.

I set epoch size 30 and batch size 4
BCE loss in test set:



```
Sørensen-Dice coefficient: 0.629
```

Dice loss in test set:

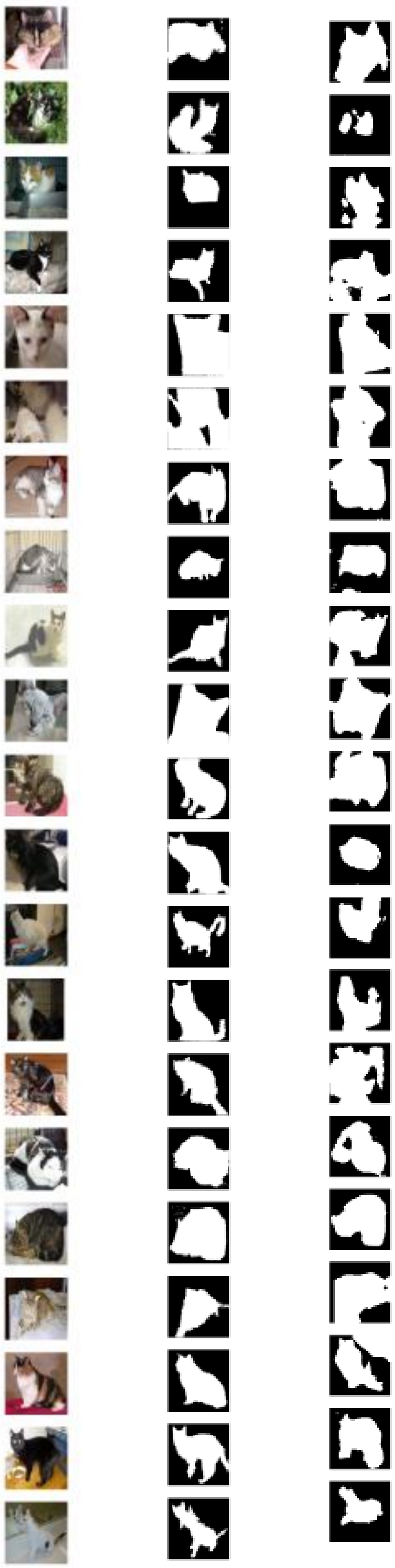


```
Sørensen-Dice coefficient: 0.713
```

The Dice loss performs better than the BCE loss in our dataset, since we measure the performance based on the dice coefficient, the Dice loss function is to minimize this loss which in turn will increase our dice coefficent even in test set. Also, the Dice loss usually performs better at class imbalanced situation. Our training dataset is small and for some training set images, the cat part occupy less pixels than background, it might be another reason that the dice loss perform better in this case. For BCE loss, it provides better gradients (easier to maximize using backpropagation), the training become more stable.

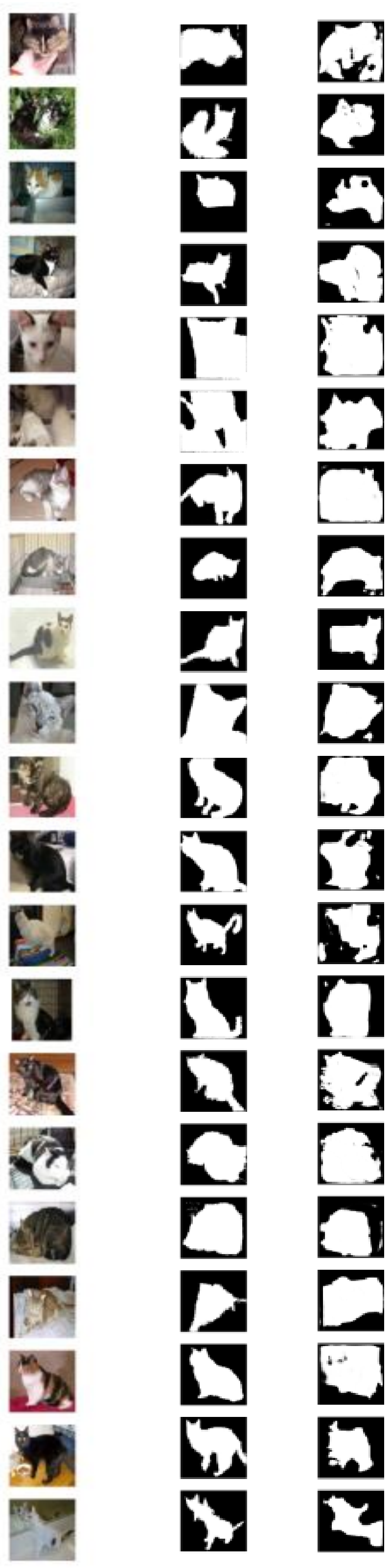
BCE performance:

The following is image, ground truth masks and predicted masks



Dice performance:

The following is image, ground truth masks and predicted masks



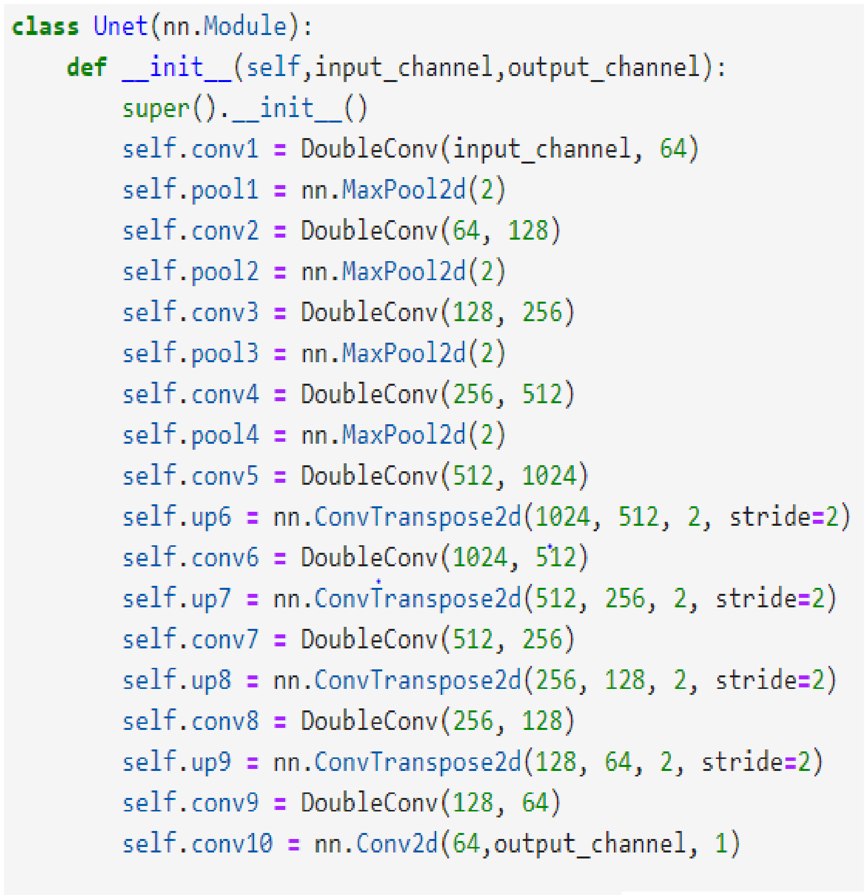
Code:

Loss function:

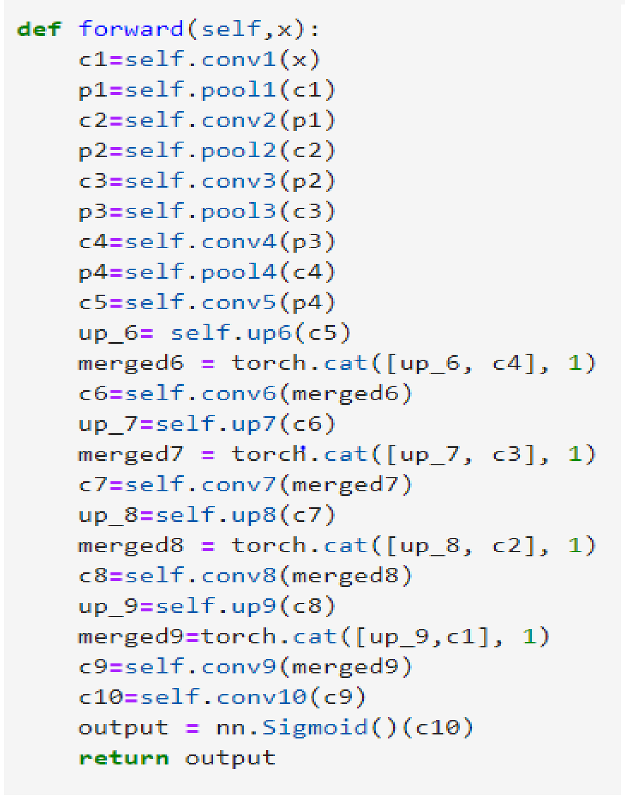
```
def bce_loss(pred, target):
    loss = -torch.mean(target*torch.log(pred) + (1-target)*torch.log(1-pred)).sum()
    return loss

def dice_loss(image,label):
    product = image.view(-1)*label.view(-1)
    intersection = torch.sum(product)
    coefficient = (2.*intersection +1.) / (torch.sum(image)+torch.sum(label) +1.)
    loss = 1. - coefficient
    return loss
```

U-net:

```
class DoubleConv(nn.Module):
    def __init__(self, input_channel, output_channel):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(input_channel, output_channel, 3, padding=1),
            nn.BatchNorm2d(output_channel),
            nn.ReLU(inplace=True),
            nn.Conv2d(output_channel, output_channel, 3, padding=1),
            nn.BatchNorm2d(output_channel),
            nn.ReLU(inplace=True)
        )

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

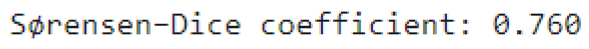


```python
class Unet(nn.Module):
    def __init__(self,input_channel,output_channel):
        super().__init__()
        self.conv1 = DoubleConv(input_channel, 64)
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = DoubleConv(64, 128)
        self.pool2 = nn.MaxPool2d(2)
        self.conv3 = DoubleConv(128, 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,output_channel, 1)
```



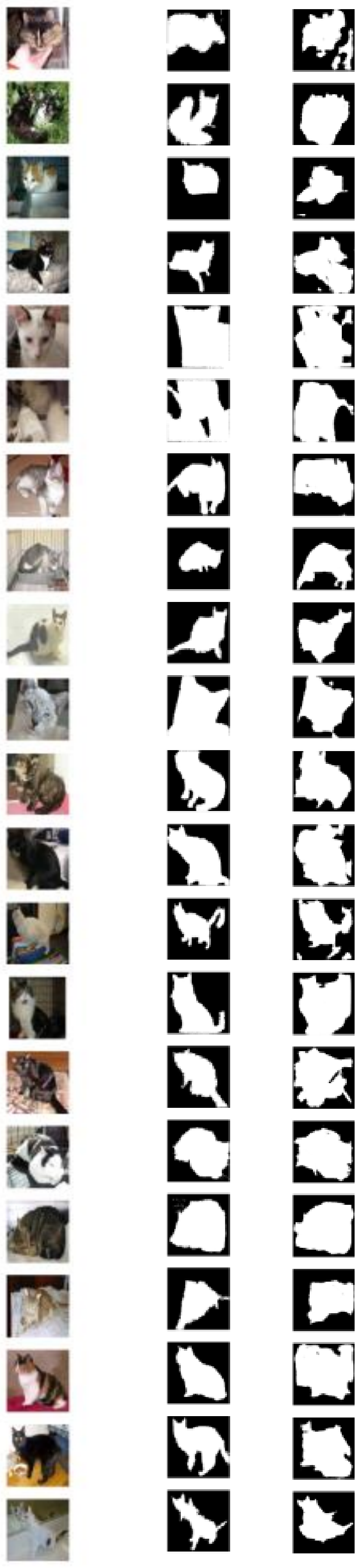
```python
def forward(self,x):
    c1=self.conv1(x)
    p1=self.pool1(c1)
    c2=self.conv2(p1)
    p2=self.pool2(c2)
    c3=self.conv3(p2)
    p3=self.pool3(c3)
    c4=self.conv4(p3)
    p4=self.pool4(c4)
    c5=self.conv5(p4)
    up_6= self.up6(c5)
    merged6 = torch.cat([up_6, c4], 1)
    c6=self.conv6(merged6)
    up_7=self.up7(c6)
    merged7 = torch.cat([up_7, c3], 1)
    c7=self.conv7(merged7)
    up_8=self.up8(c7)
    merged8 = torch.cat([up_8, c2], 1)
    c8=self.conv8(merged8)
    up_9=self.up9(c8)
    merged9=torch.cat([up_9,c1], 1)
    c9=self.conv9(merged9)
    c10=self.conv10(c9)
    output = nn.Sigmoid()(c10)
    return output
```

1.2 I used more than 4 different augmentation functions including random flip, random rotation, random affine transformation (contains translate, scale, and shear) and random resized crop. After using data augmentation, the performance is better based on Dice score.

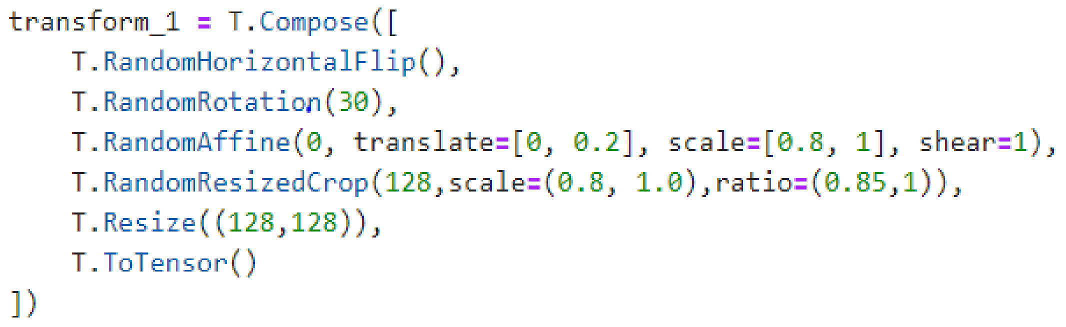


```
Sørensen-Dice coefficient: 0.760
```

The following is image, ground truth masks and predicted masks.



Code:



```
transform_1 = T.Compose([
    T.RandomHorizontalFlip(),
    T.RandomRotation(30),
    T.RandomAffine(0, translate=[0, 0.2], scale=[0.8, 1], shear=1),
    T.RandomResizedCrop(128,scale=(0.8, 1.0),ratio=(0.85,1)),
    T.Resize((128,128)),
    T.ToTensor()
])
```

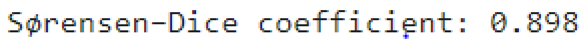
1.3 I have 3 steps for transfer learning.

Step 1: I used "The Oxford-IIIT Pet Dataset" as my new training dataset. It includes 7390 training images and masks. Since this dataset are all images of dogs and cats which are very similar to our cat training dataset, so I choose for transfer learning.

Step 2: Training u-net on this new dataset (with epoch size 30 and batch size 5) and saved the weight. The weight from this large dataset contains many generic features in earlier layers of neural networks which keep fixed and can be used later, such as edge detectors or color blob detectors. Thus, we use this weight in following steps.

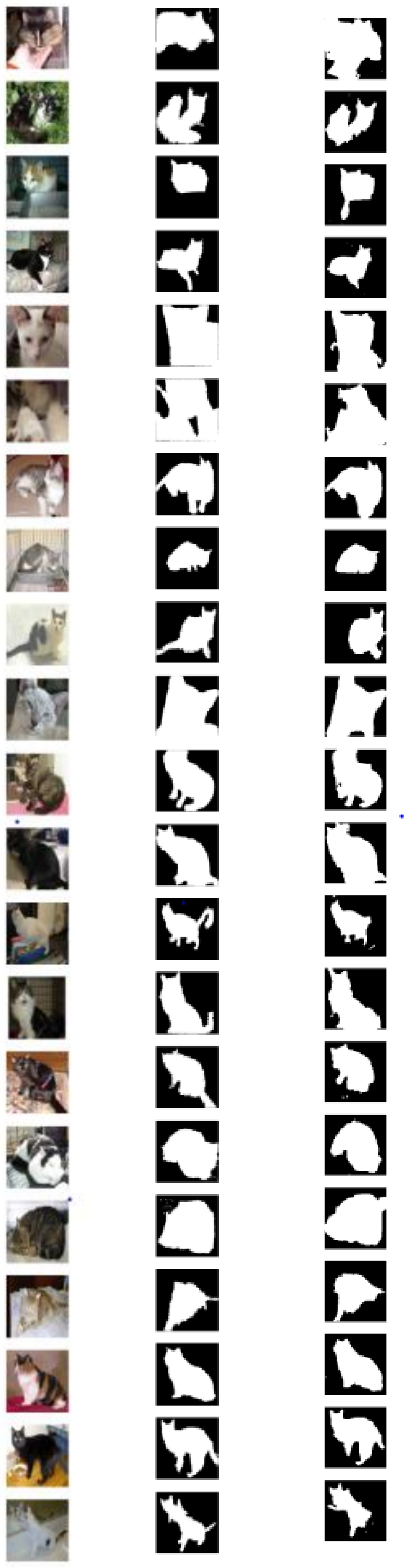
Step 3: Using the saved weight as an initialization to retrain the u-net with our cat training set (with epoch size 30 and batch size 4). This step is fine-tuning, since the weight from large dataset already contains many generic features, we want to become more specific to the details of higher-level features for our cat dataset.

Dice score:



```
Sørensen-Dice coefficient: 0.898
```

The following is image, ground truth masks and predicted masks.



Code:

```
def get_dataset(root):
    paths=[]
    count=len(os.listdir(root+'/input'))
    for file in os.listdir(root+'/input'):
        index = file.split('.')[1]
        img=os.path.join(root,"input/cat."+index+".jpg")
        mask=os.path.join(root,"mask/mask_cat."+index+".jpg")
        paths.append((img,mask))
    return paths
def get_dataset_new(root):
    paths = []
    count=len(os.listdir(root+'/input'))
    for file in os.listdir(root+'/input'):
        img = os.path.join(root,"input",file)
        name = file.split('.')[0]
        mask=os.path.join(root,"mask",name+'.png')
        paths.append((img,mask))
    return paths
```

```
class catDataset(Dataset):
    def __init__(self,root,transform = None, new_data = 0):
        if new_data == 0:
            paths = get_dataset(root)
        else:
            paths = get_dataset_new(root)
        self.new_data = new_data
        self.paths = paths
        self.transform = transform
    def __getitem__(self, index):
        img_path, mask_path = self.paths[index]
        image = Image.open(img_path).convert('RGB')
        mask = Image.open(mask_path)
        if self.new_data != 0:
            mask = np.array(mask)
            mask = (mask == 1)
            mask = mask*255
            mask = Image.fromarray(np.uint8(mask))
        if self.transform is not None:
            seed = np.random.randint(2147483647)
            random.seed(seed)
            image = self.transform(image)
            random.seed(seed)
            mask = self.transform(mask)
        return image, mask
    def __len__(self):
        return len(self.paths)
```

```python
def train_model(model,num_epoch,criterion,optimizer,dataloader,weight):
    best_sdc_score = 0
    print("start training")
    count = 0
    for epoch in range(num_epoch):
        running_loss = 0
        sdc = 0
        for imgs, labels in dataloader:
            count = count+1
            optimizer.zero_grad()
            imgs = imgs.to(device)
            labels = labels.to(device)
            outputs = model(imgs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            sdc += (1. - dice_loss(outputs, labels))
        if sdc/len(dataloader) > best_sdc_score:
            best_sdc_score = sdc/len(dataloader)
            torch.save(model.state_dict(),weight)
        print("Epoch: {}/{}".format(epoch+1, num_epoch),
              "Training Loss: {:.3f}".format(running_loss/len(dataloader)))
```

```python
def test_model(model,dataloader):
    with torch.no_grad():
        sdc = 0
        for imgs, labels in dataloader:
            imgs = imgs.to(device)
            labels = labels.to(device)
            outputs = model(imgs)
            result.append((labels,outputs,imgs))
            sdc += (1. - dice_loss(outputs, labels))
        print("Sørensen-Dice coefficient: {:.3f}".format(sdc/len(dataloader)))
```

```python
new_dataset_train = catDataset("new_data",transform,1)
new_dataloader_train = DataLoader(new_dataset_train,5)
```

```python
model.train()
train_model(model,30,dice_loss,optimizer,new_dataloader_train,'weights_new_1.pth')
```

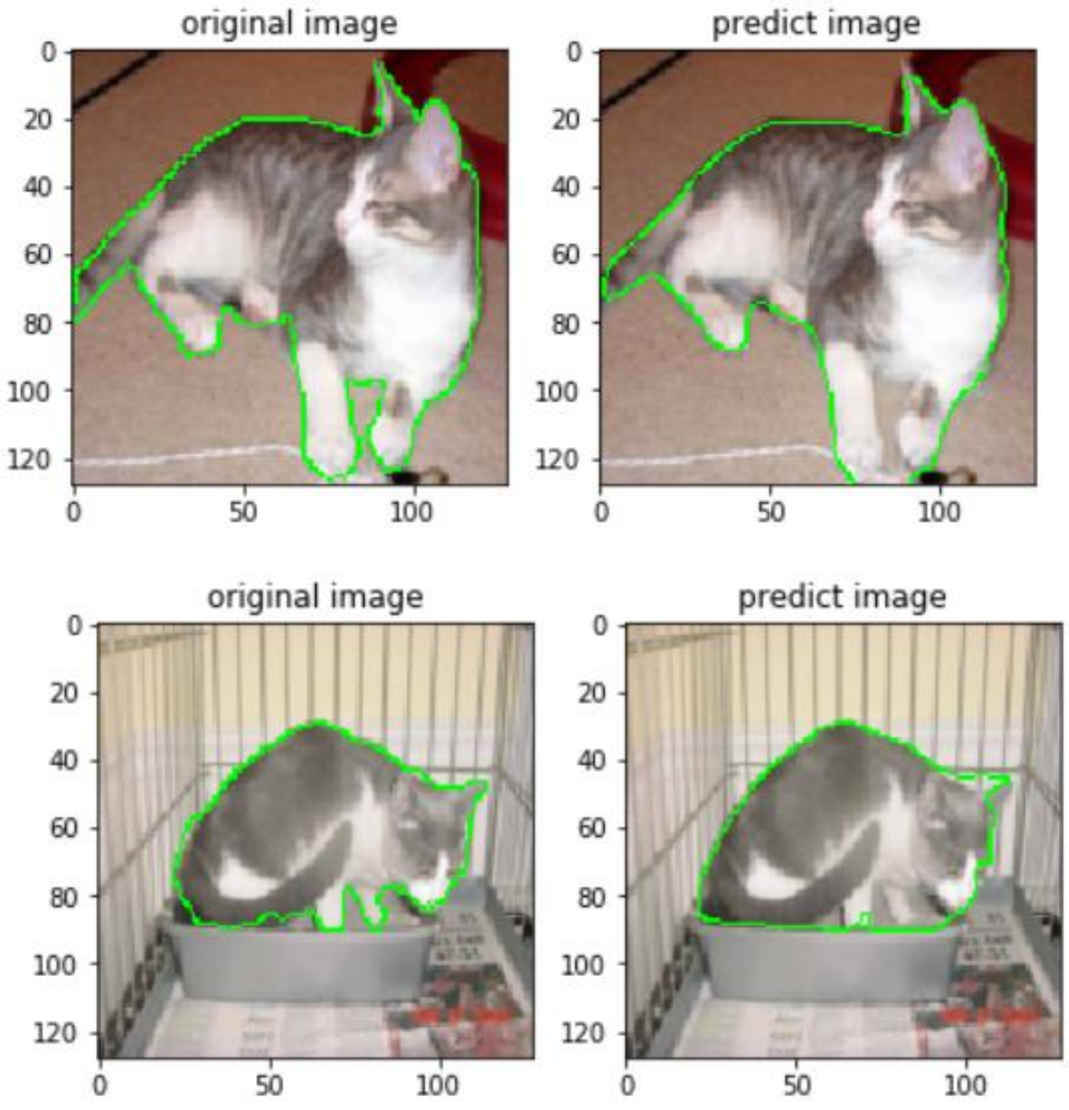
```python
cat_dataset_train = catDataset("cat_data/Train",transform_1)
dataloader_train = DataLoader(cat_dataset_train,4)
```

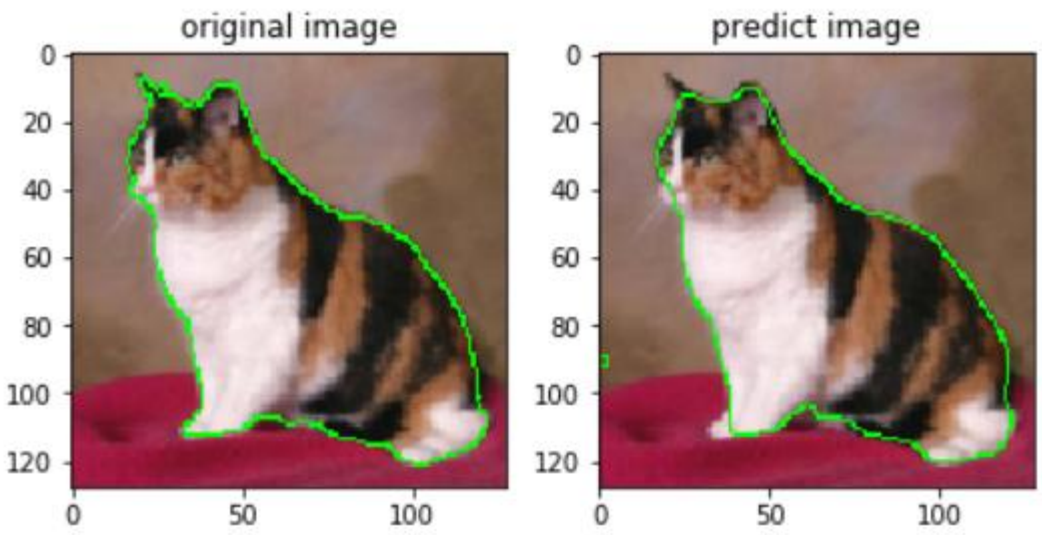
```python
model.train()
model.load_state_dict(torch.load('weights_new_1.pth'))
train_model(model,30,dice_loss,optimizer,dataloader_train,'weights_transfer_2_1.pth')
```

```
cat_dataset_test = catDataset("cat_data/Test",transform)
dataloader_test = DataLoader(cat_dataset_test)
```

```
model.eval()
model.load_state_dict(torch.load('weights_transfer_2_1.pth'))
result = []
test_model(model,dataloader_test)
```

1.4



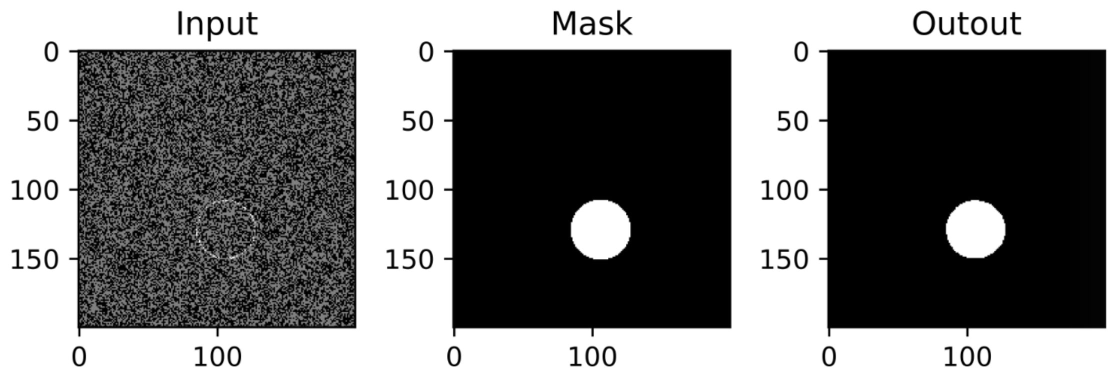


Code:

```
def draw_line(result,index):
    label,outputs,images = result[index]
    label = np.array(label.cpu().squeeze())*255
    outputs = np.array(outputs.cpu().squeeze())*255
    images = np.array(images.cpu().squeeze().permute(1, 2, 0))
    x = np.copy(images)
    y = np.copy(images)

    label_canny = cv2.Canny(np.uint8(label), 50, 120)
    x[label_canny == 255] = [0,1,0]
    plt.subplot(1,2,1)
    plt.imshow(x)
    plt.title("original image")
    outputs_canny = cv2.Canny(np.uint8(outputs), 50, 120)
    y[outputs_canny == 255] = [0,1,0]
    plt.subplot(1,2,2)
    plt.imshow(y)
    plt.title("predict image")
    plt.tight_layout()
```

2.1

Input: noisy image of a circle:
Output: filled cicle
Mask: filled circle



Loss function: Dice loss function.
Problem definition: Given noisy image of a circle, output a filled circle at the same location.

I treated this problem as image segmention first in order to get the output filled circle image without noise, then I can compute the radius and center of circle. Since the deep learning computer vision solution is end to end, given the mask, we can get corresponding output.

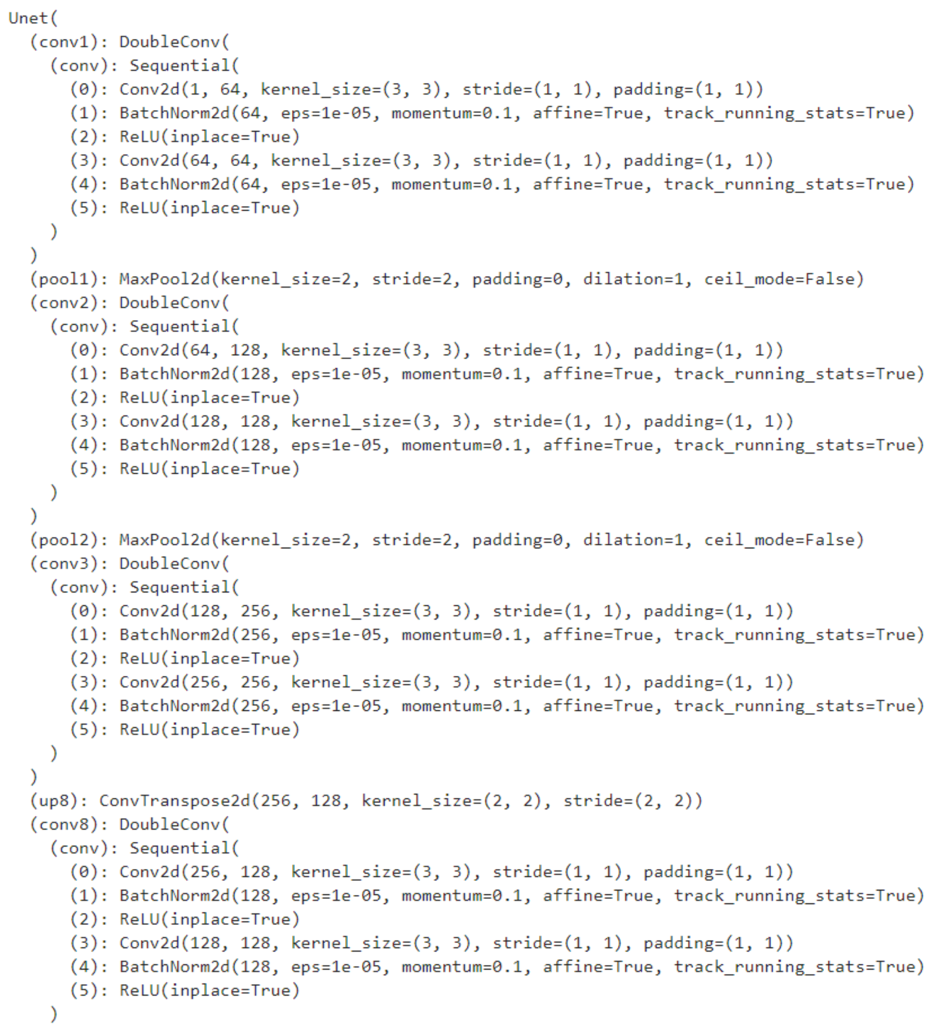
(I actually tried use circle (unfilled) as mask/label, it can visually get predicted circle in the same location, but I can not always compute correct radius since the pixels along the circle do not always show and I compute the radius according to the 4 extrema vertex (top, bottom, left, right). If too many points around vertex is missing, the radius that I computed is not accurate.)

2.2

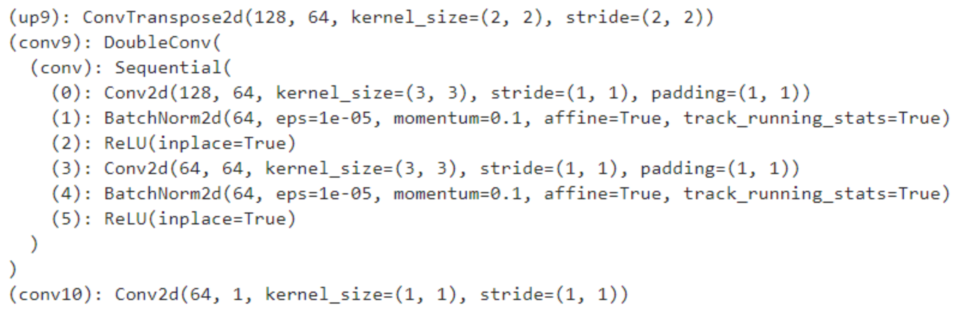
I use the simplified U-net. As we see from question 1, the U-net performs very well for image segmentation. For this simplied U-net, the contracting path will expand featue map to 256 channels, instead of original U-net 1024 channels, followed by the corresponding expansive path to output. The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3x3 convolutions. each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling. Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 convolution that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU.

Since I use the idea related to image segmentation, the U-net is an appropriate model to get the circle pattern. The architecture is based on the encoder-decoder and realize multi-scale feature recognition of image features on the network.

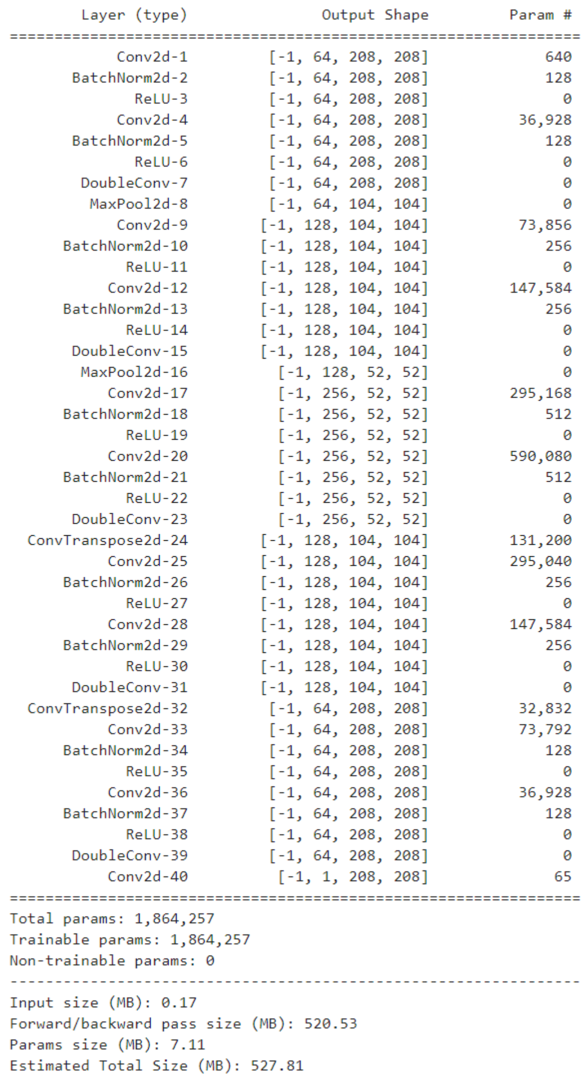
My training set is **2000** random noisy image of a circle, and with **1864257** total parameters in my model.



```
Unet(
  (conv1): DoubleConv(
    (conv): Sequential(
      (0): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): DoubleConv(
    (conv): Sequential(
      (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): DoubleConv(
    (conv): Sequential(
      (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (up8): ConvTranspose2d(256, 128, kernel_size=(2, 2), stride=(2, 2))
  (conv8): DoubleConv(
    (conv): Sequential(
      (0): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
```

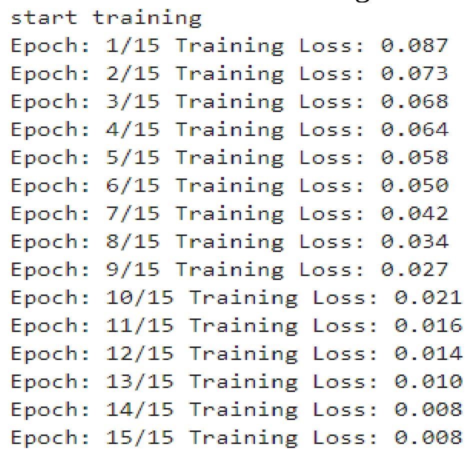


```
(up9): ConvTranspose2d(128, 64, kernel_size=(2, 2), stride=(2, 2))
(conv9): DoubleConv(
  (conv): Sequential(
    (0): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
  )
)
(conv10): Conv2d(64, 1, kernel_size=(1, 1), stride=(1, 1))
```



```
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1        [-1, 64, 208, 208]             640
       BatchNorm2d-2        [-1, 64, 208, 208]             128
              ReLU-3        [-1, 64, 208, 208]               0
            Conv2d-4        [-1, 64, 208, 208]          36,928
       BatchNorm2d-5        [-1, 64, 208, 208]             128
              ReLU-6        [-1, 64, 208, 208]               0
        DoubleConv-7        [-1, 64, 208, 208]               0
         MaxPool2d-8        [-1, 64, 104, 104]               0
            Conv2d-9       [-1, 128, 104, 104]          73,856
      BatchNorm2d-10       [-1, 128, 104, 104]             256
             ReLU-11       [-1, 128, 104, 104]               0
           Conv2d-12       [-1, 128, 104, 104]         147,584
      BatchNorm2d-13       [-1, 128, 104, 104]             256
             ReLU-14       [-1, 128, 104, 104]               0
       DoubleConv-15       [-1, 128, 104, 104]               0
        MaxPool2d-16         [-1, 128, 52, 52]               0
           Conv2d-17         [-1, 256, 52, 52]         295,168
      BatchNorm2d-18         [-1, 256, 52, 52]             512
             ReLU-19         [-1, 256, 52, 52]               0
           Conv2d-20         [-1, 256, 52, 52]         590,080
      BatchNorm2d-21         [-1, 256, 52, 52]             512
             ReLU-22         [-1, 256, 52, 52]               0
       DoubleConv-23         [-1, 256, 52, 52]               0
  ConvTranspose2d-24       [-1, 128, 104, 104]         131,200
           Conv2d-25       [-1, 128, 104, 104]         295,040
      BatchNorm2d-26       [-1, 128, 104, 104]             256
             ReLU-27       [-1, 128, 104, 104]               0
           Conv2d-28       [-1, 128, 104, 104]         147,584
      BatchNorm2d-29       [-1, 128, 104, 104]             256
             ReLU-30       [-1, 128, 104, 104]               0
       DoubleConv-31       [-1, 128, 104, 104]               0
  ConvTranspose2d-32        [-1, 64, 208, 208]          32,832
           Conv2d-33        [-1, 64, 208, 208]          73,792
      BatchNorm2d-34        [-1, 64, 208, 208]             128
             ReLU-35        [-1, 64, 208, 208]               0
           Conv2d-36        [-1, 64, 208, 208]          36,928
      BatchNorm2d-37        [-1, 64, 208, 208]             128
             ReLU-38        [-1, 64, 208, 208]               0
       DoubleConv-39        [-1, 64, 208, 208]               0
           Conv2d-40         [-1, 1, 208, 208]              65
================================================================
Total params: 1,864,257
Trainable params: 1,864,257
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.17
Forward/backward pass size (MB): 520.53
Params size (MB): 7.11
Estimated Total Size (MB): 527.81
```

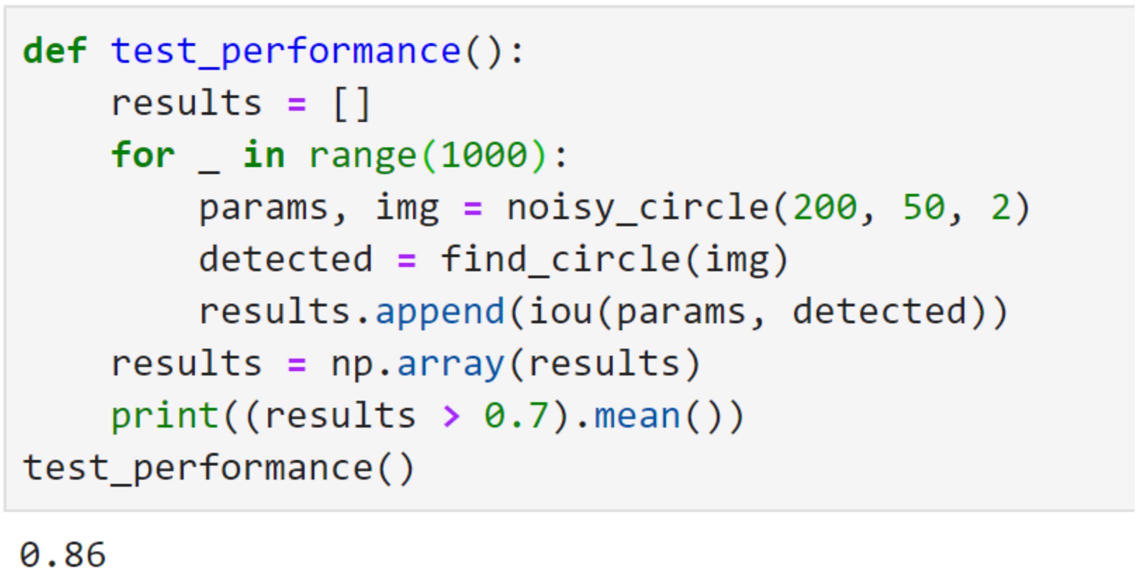
I trained the model using 2000 images (training dataset) with 15 epoch size and 4 batch size. Get 0.008 training loss in the last epoch.



```
start training
Epoch: 1/15 Training Loss: 0.087
Epoch: 2/15 Training Loss: 0.073
Epoch: 3/15 Training Loss: 0.068
Epoch: 4/15 Training Loss: 0.064
Epoch: 5/15 Training Loss: 0.058
Epoch: 6/15 Training Loss: 0.050
Epoch: 7/15 Training Loss: 0.042
Epoch: 8/15 Training Loss: 0.034
Epoch: 9/15 Training Loss: 0.027
Epoch: 10/15 Training Loss: 0.021
Epoch: 11/15 Training Loss: 0.016
Epoch: 12/15 Training Loss: 0.014
Epoch: 13/15 Training Loss: 0.010
Epoch: 14/15 Training Loss: 0.008
Epoch: 15/15 Training Loss: 0.008
```

Then I test the model performance in another randomly generated 1000 images (test dataset), given the mean IOU **0.86**

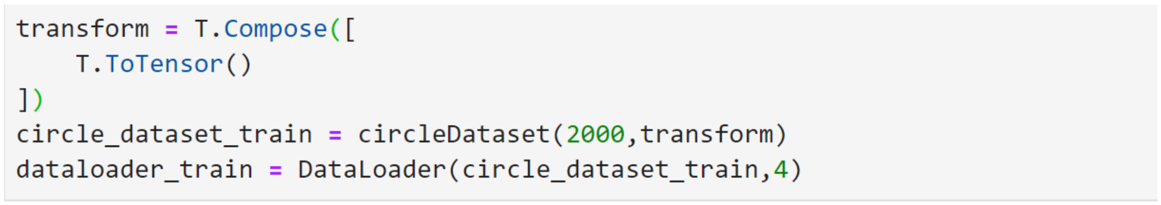
Noted: I separate the training and test dataset.



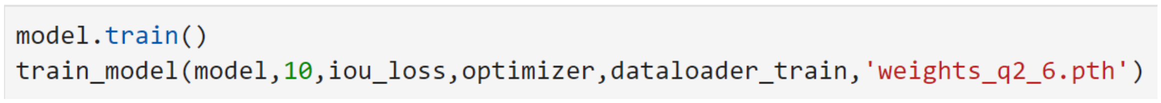
```
def test_performance():
    results = []
    for _ in range(1000):
        params, img = noisy_circle(200, 50, 2)
        detected = find_circle(img)
        results.append(iou(params, detected))
    results = np.array(results)
    print((results > 0.7).mean())
test_performance()
```

```
0.86
```

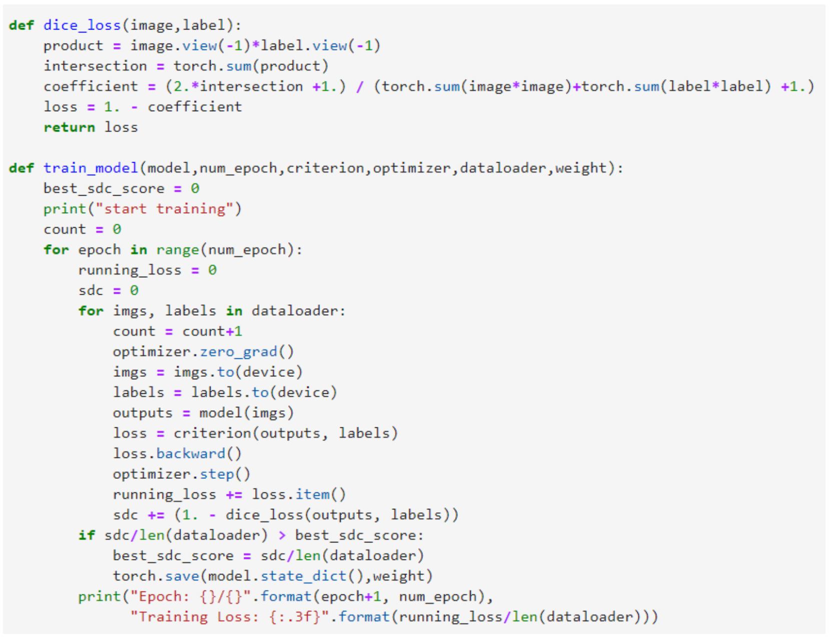
Code:



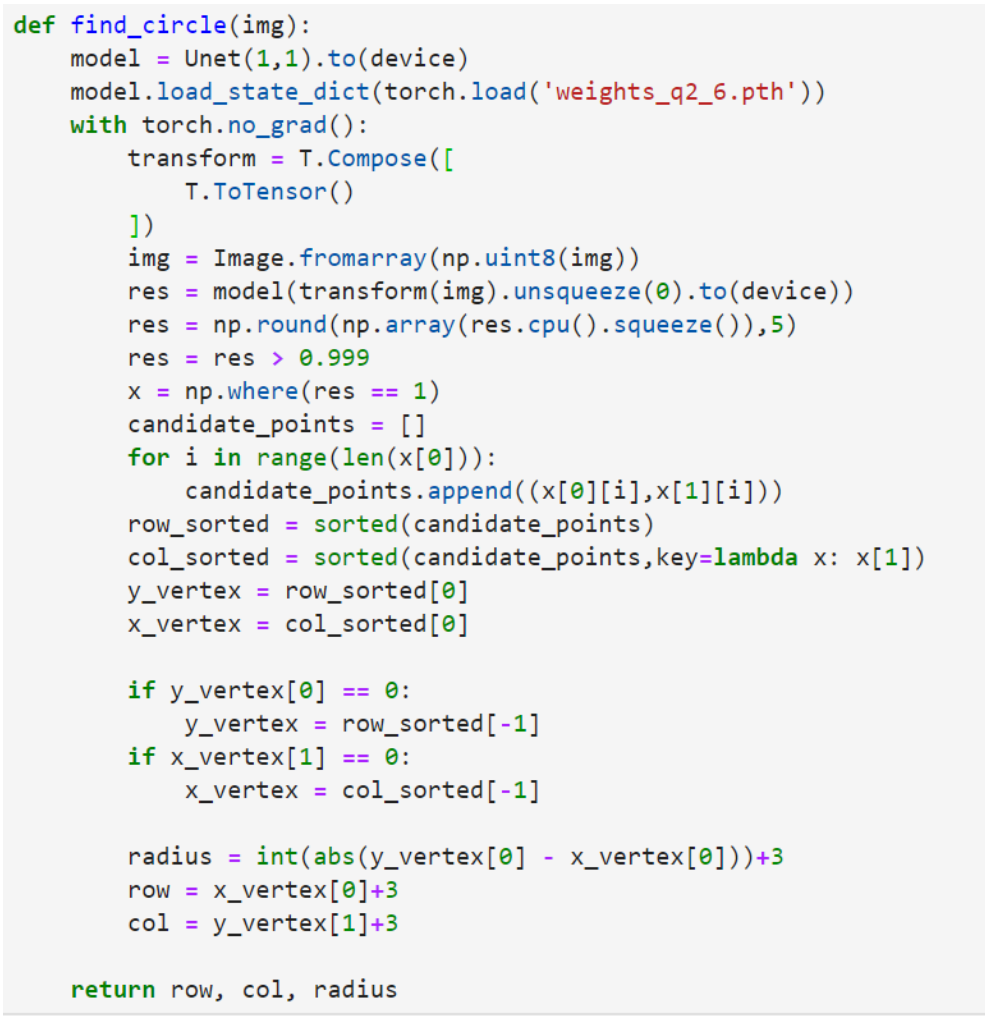
```
transform = T.Compose([
    T.ToTensor()
])
circle_dataset_train = circleDataset(2000,transform)
dataloader_train = DataLoader(circle_dataset_train,4)
```



```
model.train()
train_model(model,10,iou_loss,optimizer,dataloader_train,'weights_q2_6.pth')
```

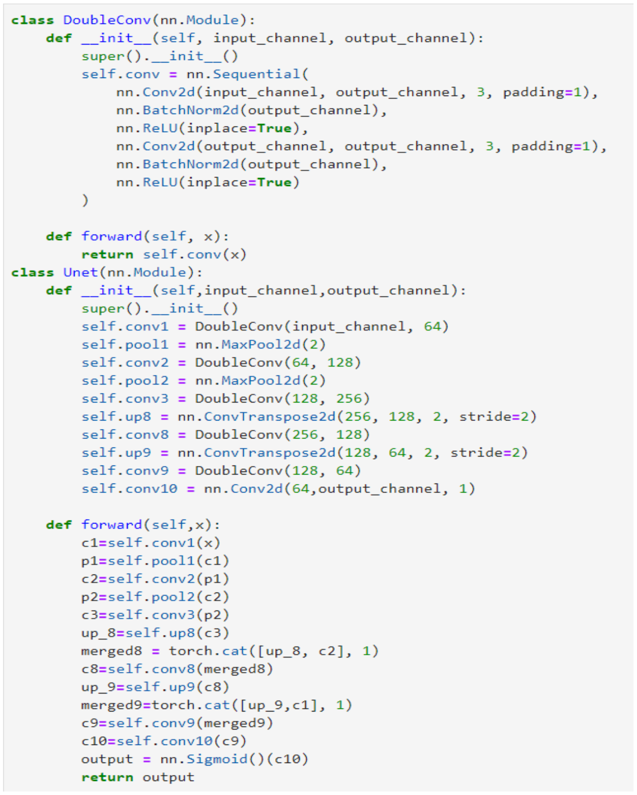


```python
def dice_loss(image,label):
    product = image.view(-1)*label.view(-1)
    intersection = torch.sum(product)
    coefficient = (2.*intersection +1.) / (torch.sum(image*image)+torch.sum(label*label) +1.)
    loss = 1. - coefficient
    return loss

def train_model(model,num_epoch,criterion,optimizer,dataloader,weight):
    best_sdc_score = 0
    print("start training")
    count = 0
    for epoch in range(num_epoch):
        running_loss = 0
        sdc = 0
        for imgs, labels in dataloader:
            count = count+1
            optimizer.zero_grad()
            imgs = imgs.to(device)
            labels = labels.to(device)
            outputs = model(imgs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            sdc += (1. - dice_loss(outputs, labels))
        if sdc/len(dataloader) > best_sdc_score:
            best_sdc_score = sdc/len(dataloader)
            torch.save(model.state_dict(),weight)
        print("Epoch: {}/{}".format(epoch+1, num_epoch),
              "Training Loss: {:.3f}".format(running_loss/len(dataloader)))
```



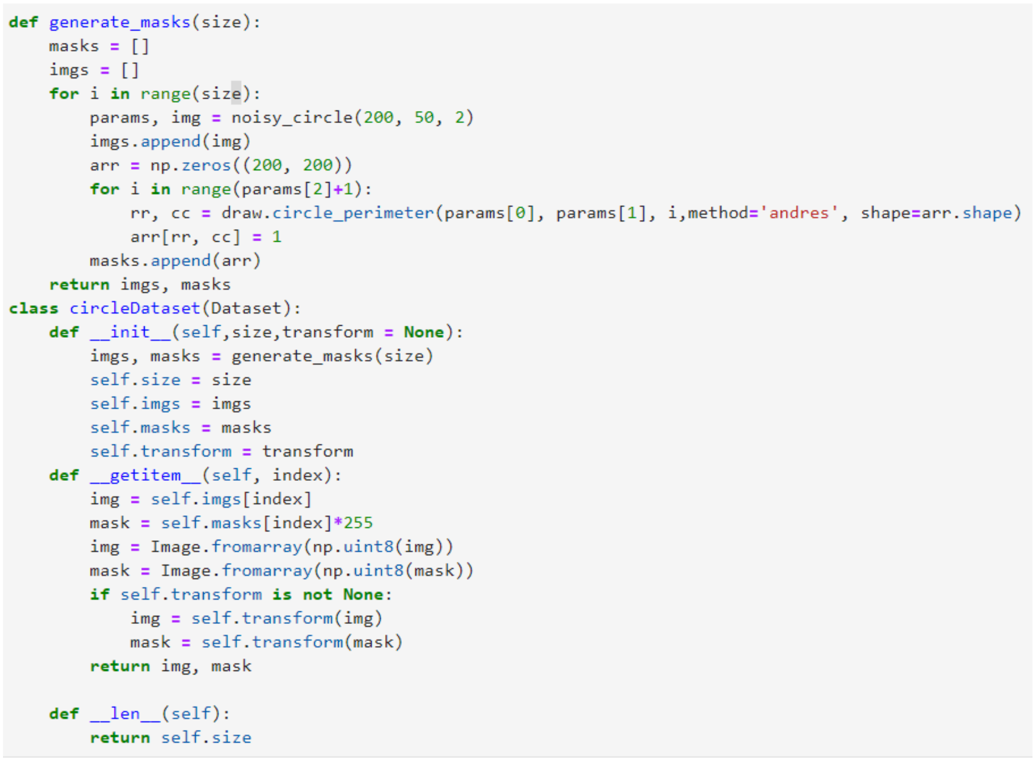
```python
def find_circle(img):
    model = Unet(1,1).to(device)
    model.load_state_dict(torch.load('weights_q2_6.pth'))
    with torch.no_grad():
        transform = T.Compose([
            T.ToTensor()
        ])
        img = Image.fromarray(np.uint8(img))
        res = model(transform(img).unsqueeze(0).to(device))
        res = np.round(np.array(res.cpu().squeeze()),5)
        res = res > 0.999
        x = np.where(res == 1)
        candidate_points = []
        for i in range(len(x[0])):
            candidate_points.append((x[0][i],x[1][i]))
        row_sorted = sorted(candidate_points)
        col_sorted = sorted(candidate_points,key=lambda x: x[1])
        y_vertex = row_sorted[0]
        x_vertex = col_sorted[0]

        if y_vertex[0] == 0:
            y_vertex = row_sorted[-1]
        if x_vertex[1] == 0:
            x_vertex = col_sorted[-1]

        radius = int(abs(y_vertex[0] - x_vertex[0]))+3
        row = x_vertex[0]+3
        col = y_vertex[1]+3

    return row, col, radius
```

Simplified U-net:



```
class DoubleConv(nn.Module):
    def __init__(self, input_channel, output_channel):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(input_channel, output_channel, 3, padding=1),
            nn.BatchNorm2d(output_channel),
            nn.ReLU(inplace=True),
            nn.Conv2d(output_channel, output_channel, 3, padding=1),
            nn.BatchNorm2d(output_channel),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.conv(x)
class Unet(nn.Module):
    def __init__(self,input_channel,output_channel):
        super().__init__()
        self.conv1 = DoubleConv(input_channel, 64)
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = DoubleConv(64, 128)
        self.pool2 = nn.MaxPool2d(2)
        self.conv3 = DoubleConv(128, 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,output_channel, 1)

    def forward(self,x):
        c1=self.conv1(x)
        p1=self.pool1(c1)
        c2=self.conv2(p1)
        p2=self.pool2(c2)
        c3=self.conv3(p2)
        up_8=self.up8(c3)
        merged8 = torch.cat([up_8, c2], 1)
        c8=self.conv8(merged8)
        up_9=self.up9(c8)
        merged9=torch.cat([up_9,c1], 1)
        c9=self.conv9(merged9)
        c10=self.conv10(c9)
        output = nn.Sigmoid()(c10)
        return output
```

Dataset:



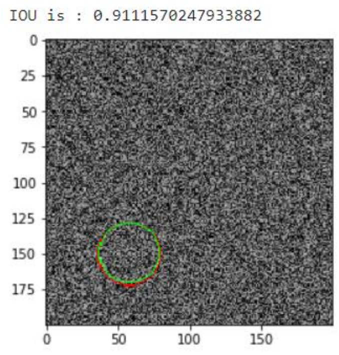
```
def generate_masks(size):
    masks = []
    imgs = []
    for i in range(size):
        params, img = noisy_circle(200, 50, 2)
        imgs.append(img)
        arr = np.zeros((200, 200))
        for i in range(params[2]+1):
            rr, cc = draw.circle_perimeter(params[0], params[1], i,method='andres', shape=arr.shape)
            arr[rr, cc] = 1
        masks.append(arr)
    return imgs, masks
class circleDataset(Dataset):
    def __init__(self,size,transform = None):
        imgs, masks = generate_masks(size)
        self.size = size
        self.imgs = imgs
        self.masks = masks
        self.transform = transform
    def __getitem__(self, index):
        img = self.imgs[index]
        mask = self.masks[index]*255
        img = Image.fromarray(np.uint8(img))
        mask = Image.fromarray(np.uint8(mask))
        if self.transform is not None:
            img = self.transform(img)
            mask = self.transform(mask)
        return img, mask

    def __len__(self):
        return self.size
```

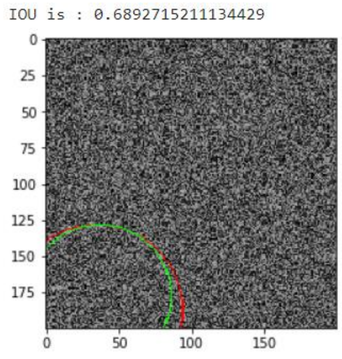
2.3

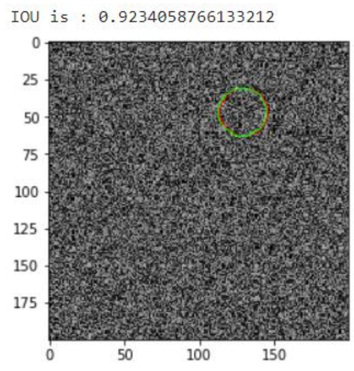
Using IOU directly, it causes the training loss unchanged or change slowly. Firstly, it does not differentiable so we are unable to get gradient. Then, the IOU is not sufficient to simply evaluate a metric while solving a relevant problem. Evaluating the metric must also help the network learn in which direction the weights must be nudged towards, so that a network can learn effectively over epochs. In total, it is not a good idea using IOU as loss function.

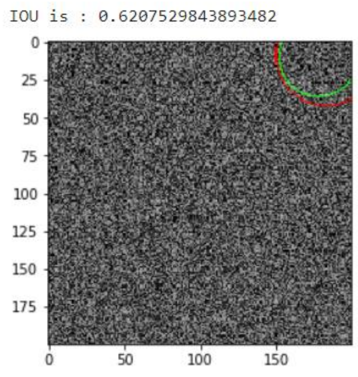
2.4

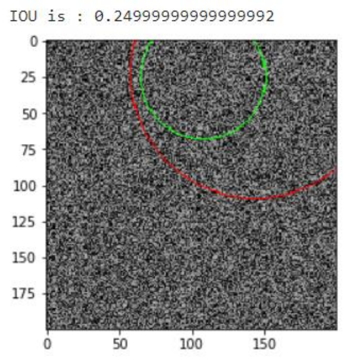
Green circle is origin circle
Red circle is predicted circle

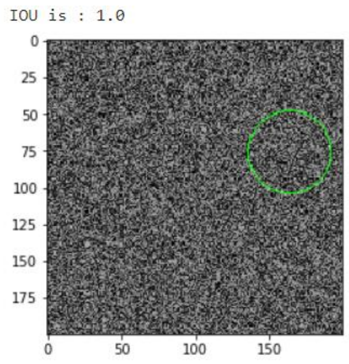






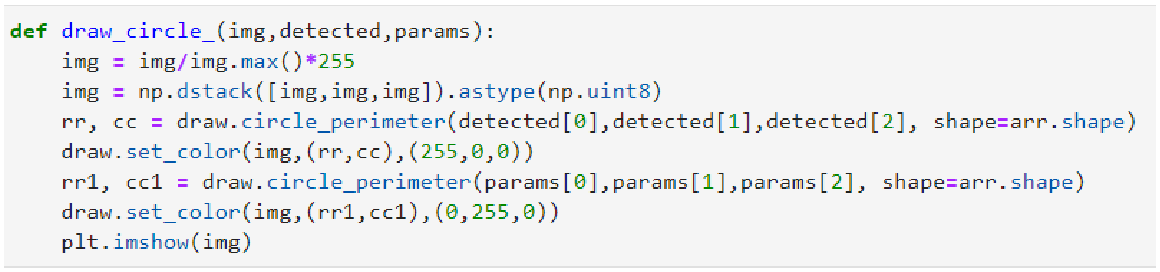






The mistakes usually happen around corner and edge, the main reason is because my input of network is not a filled circle, and I treated this problem as image segmentation, it can not perform very well in all the cases.

Code:



```
def draw_circle_(img,detected,params):
    img = img/img.max()*255
    img = np.dstack([img,img,img]).astype(np.uint8)
    rr, cc = draw.circle_perimeter(detected[0],detected[1],detected[2], shape=arr.shape)
    draw.set_color(img,(rr,cc),(255,0,0))
    rr1, cc1 = draw.circle_perimeter(params[0],params[1],params[2], shape=arr.shape)
    draw.set_color(img,(rr1,cc1),(0,255,0))
    plt.imshow(img)
```

3.

We do not know whether the picture a hot dog or not, since the output is not depending on model itself or output scalar value, it depends on what label/mask they use for training, and the threshold they use, since the deep convolutional netural network is end to end. If they label greater than threshold as hot dog, then we can check the value c whether greater than threshold to determine the result.