

## CSC420 Assignment 3 Report

Question 1:

### How to use:

- Unzip q1\_models.zip and place the folder under your current directory – where your program is
- Please make sure the folders **cat\_data** and **q1\_models** are at the **same directory** with the code.
- The pre trained models in q1\_models are model\_q1.1.pt, model\_q1.2.pt, model\_q1.3.pt, please use the corresponding model from each question.

```
# Toggle these this boolean to enable training mode and model test mode
startTrain = False
# Enable output the image
plot_cat = True
# Enable segmentation in original image
plot_cat_segmentation = True
if startTrain:
    train_q1()
    train_q2()
    train_q3()
else:
    # In model test mode, I assume you have the corresponding model in ./q1_models directory
    runQ1(plot_cat, plot_cat_segmentation)
    runQ2(plot_cat, plot_cat_segmentation)
    runQ3(plot_cat, plot_cat_segmentation)
```

**My Unet architecture:** My Unet generally is following the architecture described in paper. The input is a grayscale image (1 channel) and output is 1 channel as well (pixel intensity), so my network inputs an image and output an image as well. I use 4 down convolution layers and one rf convolution layer, and 4 up convolution layers. My final convolution layer would output 1 channel for each pixel of the image. I also use a padding of 1 in order to let 128 \* 128 image fit in the model.

**NOTE: All of my output image in grayscale because I always convert the input image to grayscale before feed into the network. I can only plot a grayscale image.**

1.1:

Main codes:

Q1 train:

```
def train_q1():
    #####
    # Question1.1
    #####
    train_set = imagesDataSet(train_img_data, train_mask_data, device)
    val_set = imagesDataSet(test_img_data, test_mask_data, device)
    trainloader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True)
    testloader = torch.utils.data.DataLoader(val_set, batch_size=batch_size, shuffle=True)
    unet = UNet(64, num_class).to(device)
    criterion = MSELoss()
    # criterion = DiceLoss()
    unet = mainLoop(unet, num_epochs, learn_rate, trainloader, testloader, device, criterion)
    saveModel(unet, "q1_models", "model_q1.1.pt")
```

## Main loop of the training

```
def mainloop(unet, num_epochs, learn_rate, trainloader, testloader, device, criterion):
    optimizer = torch.optim.Adam(unet.parameters(), lr=learn_rate)
    best_loss = 1e10
    scheduler = lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)
    best_model = None
    for epoch in range(num_epochs):
        unet, epoch_loss = train(epoch, num_epochs, optimizer, trainloader, unet, device, criterion, scheduler=scheduler)
        test(unet, epoch, num_epochs, testloader, device, criterion)
        if epoch_loss < best_loss:
            print("Saving best model")
            best_loss = epoch_loss
            best_model = unet
    return best_model
```

## Loss functions:

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

    def forward(self, predict, target):
        loss = torch.mean((predict - target) ** 2)
        return loss

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

    def forward(self, predict, target):
        predict = predict.contiguous()
        target = target.contiguous()
        intersect = (predict * target).sum(dim=2).sum(dim=2)
        loss = (1 - ((2. * intersect + 1.) / (predict.sum(dim=2).sum(dim=2) + target.sum(dim=2).sum(dim=2) + 1.)))
        return loss.mean()
```

## Unet:

```
class UNet(nn.Module):
    def __init__(self, num_filter, num_class):
        super(UNet, self).__init__()
        kernel = 3
        padding = kernel // 2
        self.downconv1 = seqLayersDown(1, num_filter, kernel, 1)
        self.downconv2 = seqLayersDown(num_filter, num_filter*2, kernel, padding)
        self.downconv3 = seqLayersDown(num_filter*2, num_filter*4, kernel, padding)
        self.downconv4 = seqLayersDown(num_filter*4, num_filter*8, kernel, padding)

        self.rfconv = nn.Sequential(
            nn.Conv2d(num_filter*8, num_filter*8, kernel_size=kernel, padding=padding),
            nn.BatchNorm2d(num_filter*8),
            nn.ReLU()
        )

        self.upconv4 = seqLayersUp(num_filter*8 + num_filter*8, num_filter*8, kernel, padding)
        self.upconv3 = seqLayersUp(num_filter*8 + num_filter*4, num_filter*4, kernel, padding)
        self.upconv2 = seqLayersUp(num_filter*4 + num_filter*2, num_filter*2, kernel, padding)
        self.upconv1 = seqLayersUp(num_filter*2 + num_filter, 3, kernel, padding)

        self.finalconv = nn.Conv2d(1+3, num_class, kernel_size=kernel, padding=padding)

    def forward(self, x):
        self.out1 = self.downconv1(x)
        self.out2 = self.downconv2(self.out1)
        self.out3 = self.downconv3(self.out2)
        self.out4 = self.downconv4(self.out3)
        self.rfOut = self.rfconv(self.out4)
        self.out5 = self.upconv4(torch.cat((self.rfOut, self.out4), dim=-1))
        self.out6 = self.upconv3(torch.cat((self.out5, self.out3), dim=-1))
        self.out7 = self.upconv2(torch.cat((self.out6, self.out2), dim=-1))
        self.out8 = self.upconv1(torch.cat((self.out7, self.out1), dim=-1))
        self.out_final = self.finalconv(torch.cat((self.out8, x), dim=-1))
        return self.out_final
```

## Train:

```
def train(epoch, num_epochs, optimizer, trainloader, unet, device, criterion, scheduler=None):
    start = time.time()
    unet.train() # Change model to 'train' mode
    running_loss = 0
    accuracy = 0

    for images, labels in trainloader:
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = unet(images)

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        if(scheduler):
            scheduler.step()
        accuracy = calculate_dice_coeff(accuracy, labels, outputs)
        running_loss += loss.item()

    train_acc = accuracy / len(trainloader)
    print('Training: Epoch [%d/%d] Loss: %.4f, Dice Coefficient: %.4f, Time (s): %d' % (
        epoch + 1, num_epochs, running_loss/len(trainloader), train_acc, time.time() - start))
    return unet, running_loss/len(trainloader)
```

## Test:

```
def test(unet, epoch, num_epochs, testloader, device, criterion):
    unet.eval()
    start = time.time()
    val_acc, val_loss, _ = runValidation(criterion, device, testloader, unet)
    time_elapsed = time.time() - start
    print('Testing: Epoch [%d/%d], Val Loss: %.4f, Dice Coefficient: %.4f, Time(s): %d' % (
        epoch + 1, num_epochs, val_loss, val_acc, time_elapsed))
    return val_acc

def runValidation(criterion, device, testloader, unet, store=False):
    accuracy = 0
    running_loss = 0
    outputs_store = []
    labels_store = []
    img_store = []

    for images, labels in testloader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = unet(images)

        if store:
            outputs_store.append(outputs)
            labels_store.append(labels)
            img_store.append(images)

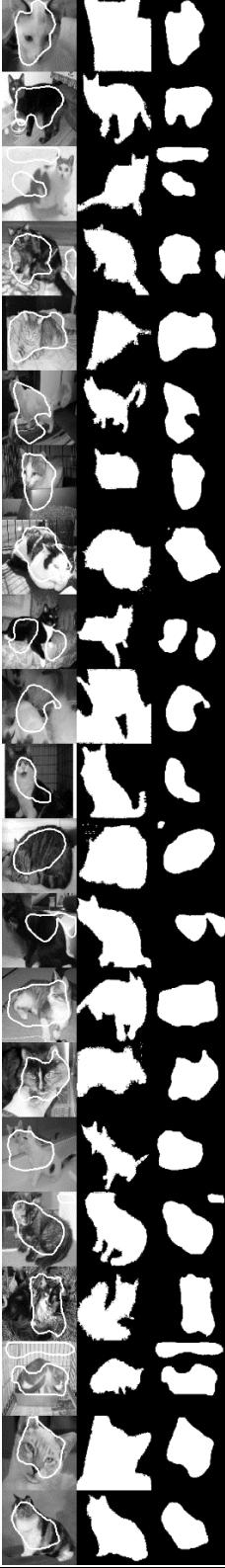
        loss = criterion(outputs, labels)
        accuracy = calculate_dice_coeff(accuracy, labels, outputs)
        running_loss += loss.item()

    val_loss = running_loss / len(testloader)
    val_acc = accuracy / len(testloader)
    stores = (outputs_store, labels_store, img_store)
    return val_acc, val_loss, stores
```

Default Loss function is MSE loss

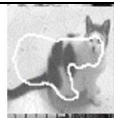

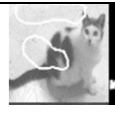

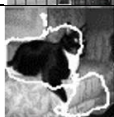





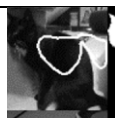

Loss function: MSE loss and Dice Loss

Result of training use MSE loss	question_1.1_plot_mse.png (cat image, mask, prediction)	
Val Loss: 0.1674 Dice Coefficient: 0.7611		

Result of Dice Loss	question1.1_plot_dice.png (cat image, mask, prediction)
<p data-bbox="203 233 560 304">Val Loss: 2.5425 Dice Coefficient: 0.7358</p>	

Observation:

The results (dice coefficient) of the two loss functions are closed, but I personally more like the MSE result.

MSE Loss result			Dice Loss result		
					
					
					

These are examples that MSE Loss wins against Dice loss. When a network is trained with dice loss, it often fails to recognize the cat and make two separate pieces of segment predictions, but except the first row, MSE loss doesn't seem to have this kind of problem, and even the first row of MSE loss predictions is minor compared to the failure cases in Dice Loss predictions.

Since my network outputs only 1 class which is just the intensity of each pixel, Euclidean distance seems to be a better choice for this kind of problem. The loss function penalizes when the same position pixels of prediction and ground truth are very different which is very intuitive. As a result, the model learns from it and improves its predictions.

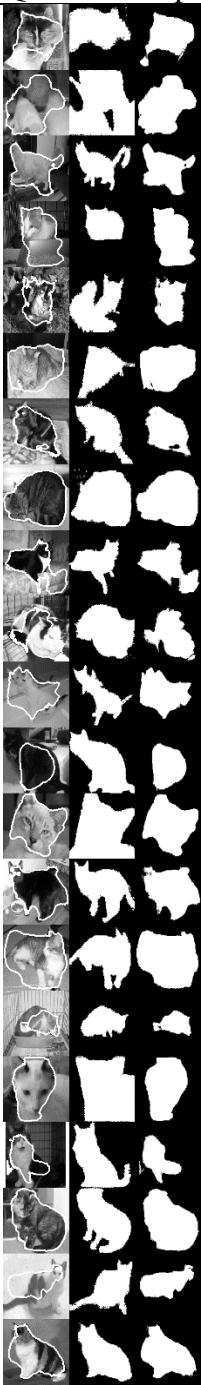
## 1.2

Main codes:

```
def train_q2():
    #####
    # Question1.2
    #####
    train_img_data_aug = makeDataSet(train_images_path, augmentation=True)
    train_mask_data_aug = makeDataSet(train_mask_path, augmentation=True)
    test_img_data_aug = makeDataSet(test_images_path)
    test_mask_data_aug = makeDataSet(test_mask_path)
    train_set = imagesDataSet(train_img_data_aug, train_mask_data_aug, device)
    val_set = imagesDataSet(test_img_data_aug, test_mask_data_aug, device)
    trainloader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True)
    testloader = torch.utils.data.DataLoader(val_set, batch_size=batch_size, shuffle=True)
    unet = UNet(64, num_class).to(device)
    criterion = MSELoss()
    unet = mainLoop(unet, num_epochs, learn_rate, trainloader, testloader, device, criterion)
    saveModel(unet, "q1_models", "model_q1.2.pt")
```

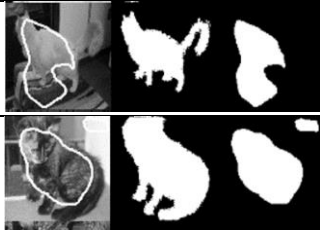
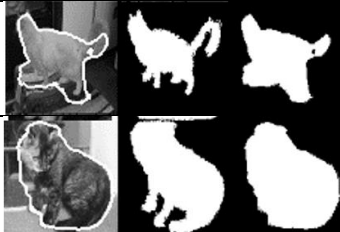
My four augmentations: horizontal flip, rotate image 90 degree clockwise, flip vertically and horizontally, and increase contrast of the image.

```
def makeDataSet(path, augmentation="_None"):
    images = []
    for file in glob.glob(path):
        img = cv.imread(file, cv.IMREAD_GRAYSCALE)
        img = resizeImg(img)
        if augmentation:
            # original image
            images.append(np.expand_dims(img, axis=0))
            # Horizontal Flip
            horizontal_flip = cv.flip(img, 1)
            horizontal_flip = np.expand_dims(horizontal_flip, axis=0)
            images.append(horizontal_flip)
            # rotated image
            rotation = cv.rotate(img, cv.ROTATE_90_CLOCKWISE)
            rotation = np.expand_dims(rotation, axis=0)
            images.append(rotation)
            # flip image horizontally and vertically
            flip = cv.flip(img, -1)
            flip = np.expand_dims(flip, axis=0)
            images.append(flip)
            # increase contrast of the image
            contrast = cv.equalizeHist(img)
            contrast = np.expand_dims(contrast, axis=0)
            images.append(contrast)
        else:
            img = np.expand_dims(img, axis=0)
            images.append(img)
    return np.array(images)/255.0
```

Loss function: MSE loss	Question1.2_plot.png
Val Loss: 0.1160 Dice Coefficient: 0.8279	



Comparison:

1.1 (MSE loss)	1.2 (MSE loss)
	

By increasing data size, these are obvious improvements between two predictions. In 1.2, the model correctly recognize the cat's shape and it is almost correct in my opinion.

1.3:

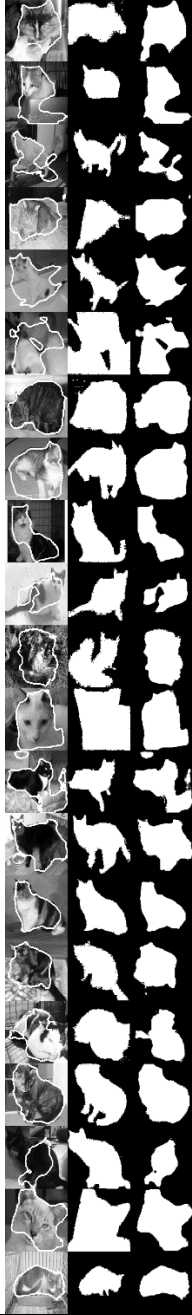
**Note: When first time run the code, the programs would create a directory ‘data’ at your current directory and download the necessary data set. When you run it the second time, it would verify your data first before training starts, it takes about 10 min for me for the whole verify and run process, so please be patient.**

Main code:

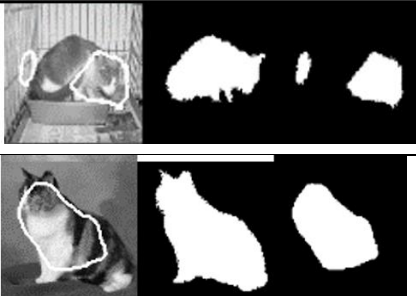
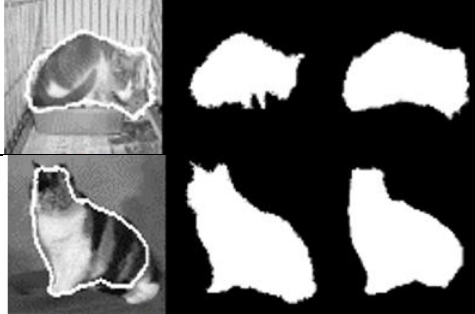
```
def train_all():
    # =====
    # Download
    # =====
    # Resize images and convert to grayscale to fit to DNN
    input_transform = transforms.Compose([
        transforms.Resize((128, 128)),
        transforms.Grayscale(),
        transforms.ToTensor()
    ])
    target_transform = transforms.Compose([
        transforms.Resize((128, 128)),
        transforms.ToTensor()
    ])
    # Fetch data from VOC2012
    train_set = torchvision.datasets.VOCSegmentation(root='./data', download=True, transform=input_transform,
                                                    target_transform=target_transform)
    test_set = torchvision.datasets.VOCSegmentation(root='./data', download=True, transform=input_transform,
                                                    target_transform=target_transform)
    trainloader = torch.utils.data.DataLoader(train_set, batch_size=16, num_workers=4, shuffle=True, pin_memory=True)
    testloader = torch.utils.data.DataLoader(test_set, batch_size=16, shuffle=True, num_workers=4)
    net = UNet(64, num_class=21)(device)
    criterion = BCELoss()
    optimizer = optim.Adam(net.parameters())
    # Train our own CNN
    train_set = ImageDataset(train_img_dir, train_mask_dir, device)
    val_set = ImageDataset(val_img_dir, val_mask_dir, device)
    trainloader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True)
    testloader = torch.utils.data.DataLoader(val_set, batch_size=batch_size, shuffle=True)
    net = nn.DataParallel(net, device_ids=[0])
    trainer = GradientDescentTrainer(net, trainloader, testloader, device, criterion, optimizer)
```

Data set used for transfer learning: VOC segmentation 2012

<http://host.robots.ox.ac.uk/pascal/VOC/voc2012/> (the website may be down for some reasons...)

Loss function: MSE loss	Question1.3_plot.png	
Val Loss: 0.1300 Dice Coefficient: 0.8363		

Comparison:

1.1(MSE Loss)	1.3 (MSE Loss)
	

The two examples show great improvement in before and after transfer learning. The data set provides images of vehicle, planes, humans, animals, etc. along with the correct labels of them. I haven't done much fine tuning for this question because without changing any of my parameters in my model, my model can run perfectly in the dataset and it achieves a dice score of 95% on every epoch. After training on VOC2012 data set, I apply the model to my data set, and it improves performance greatly as I show above. By doing find tuning, people often modify the last layer of the model and output classes in order to fit in the other data set, but VOC2012 data set has a one class label, which is same as my label, so I didn't bother to change my last layer. As a result, transfer learning works well.

1.4:

Main code:

```
def plot(outputs_store, labels_store, img_store, filename, device, x_dim=200, y_dim=200, need_visualize=False):
    if (len(outputs_store) == 0):
        return None
    else:
        output = plotImageFromData(device, outputs_store, x_dim, y_dim)
        labels = plotImageFromData(device, labels_store, x_dim, y_dim)
        if need_visualize:
            images = plotImageFromData(device, img_store, x_dim, y_dim, orig_image=True, need_visualize=need_visualize,
                                       mask_store=outputs_store)
        else:
            images = plotImageFromData(device, img_store, x_dim, y_dim, orig_image=True, need_visualize=need_visualize)
        result = np.hstack((images, labels, output))
        cv.imwrite(filename + ".png", result)
```

```
def visualize(image, mask):
    mask = (mask * 255.0).astype(np.uint8)
    mask = cv.Canny(mask, 100, 200)
    contours, hierarchy = cv.findContours(mask, cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)

    image = cv.drawContours(image, contours, -1, 1, 3)
    return image
```

The results of visualization are already showing above.

As I mention, my visualization would be in grayscale since my input is a grayscale image, I don't want to take extra memories for storing the RGB images for plotting. As long as the segmentations are clear, I don't think outputting an RGB image is necessary for this question.

Question 2:

How to use:

- Unzip q2\_models.zip and place the folder under your current directory – where your program is
- Please make sure the folder **q2\_models** are at the **same directory** with the code.
- The pre trained models in q2\_models are model\_row.pt, model\_col.pt, model\_rad.pt, please use the corresponding model from each question.

```
# Toggle training mode and model test mode using this boolean
train_model = False
# Enable circle plotting by this boolean
plot_circle = False
if train_model:
    startTraining(device)
else:
    models = loadModels(device)
    results = []
    for _ in range(1000):
        params, img = noisy_circle(200, 50, 2)
        detected = find_circle(img, models, device)
        if plot_circle:
            visualize(params, detected)
        results.append(iou(params, detected))
    results = np.array(results)
    print("100 mean: %.4f" % (results.mean()))
    print("100 > 0.7 mean: %.4f" % ((results > 0.7).mean()))
```

## 2.1:

- The input of my network is the noised circle image and output of my network is a number (row, col, or radius).

Explanation of my approach to the problem: The input is a noised circle image, I use three CNN models to predict row, column, radius of the circle separately, but using the same noised circle image. The loss function is MSE loss again: this is obviously a very suitable loss function for this problem – we want to minimize the distance between our predicted parameters and ground truth parameters of the circle. By predicting the three parameters using three models instead of predicting three parameters use one model, the model can optimize just for one parameter, make the prediction more accurate.

## 2.2

The model takes **124513** parameters, and I am using 50000 training data points (training noised circle images)

Main code:

```
class CNN(nn.Module):
    def __init__(self, num_filters, image_size):
        super(CNN, self).__init__()
        self.downconv1 = nn.Sequential(
            nn.Conv2d(1, num_filters, kernel_size=3, padding=2),
            nn.BatchNorm2d(num_filters),
            nn.ReLU(),
            nn.MaxPool2d(2), )
        self.downconv2 = nn.Sequential(
            nn.Conv2d(num_filters, num_filters * 2, kernel_size=3, padding=2),
            nn.BatchNorm2d(num_filters * 2),
            nn.ReLU(),
            nn.MaxPool2d(2), )
        self.downconv3 = nn.Sequential(
            nn.Conv2d(num_filters*2, num_filters * 4, kernel_size=3, padding=1),
            nn.BatchNorm2d(num_filters * 4),
            nn.ReLU(),
            nn.MaxPool2d(2), )
        self.downconv4 = nn.Sequential(
            nn.Conv2d(num_filters * 4, num_filters*8, kernel_size=3, padding=1),
            nn.BatchNorm2d(num_filters*8),
            nn.ReLU(),
            nn.MaxPool2d(2), )
        self.finalconv = nn.Linear(num_filters*8*(image_size//16)*(image_size//16), 1)

    def forward(self, x):
        self.out1 = self.downconv1(x)
        self.out2 = self.downconv2(self.out1)
        self.out3 = self.downconv3(self.out2)
        self.out4 = self.downconv4(self.out3)
        out = self.out4.view(self.out4.size(0), -1)
        self.out_final = self.finalconv(out)
        return self.out_final
```

```
def main(num_epochs, learn_rate, trainloader, testloader, device):
    # Initialize model per parameter
    model_row = CNN(16, 200, 16(device))
    model_col = CNN(16, 200, 16(device))
    model_rad = CNN(16, 200, 16(device))
    print("The model takes %d parameters" % (count_parameters(model_row)))
    # LOSS FUNCTION
    criterion = nn.MSELoss()
    # Initialize optimizer
    opt_row = torch.optim.Adam(model_row.parameters(), lr=learn_rate)
    opt_col = torch.optim.Adam(model_col.parameters(), lr=learn_rate)
    opt_rad = torch.optim.Adam(model_rad.parameters(), lr=learn_rate)
    # Initialize best loss
    best_loss = 1e10
    for epoch in range(num_epochs):
        model_row, epoch_loss = train(epoch, num_epochs, (opt_row, opt_col, opt_rad), trainloader,
                                     (model_row, model_col, model_rad), device, criterion, log_loss_fn=False)
        test(model_row, epoch, num_epochs, testloader, device, criterion)
        if epoch_loss < best_loss:
            print("Saving best model")
            best_loss = epoch_loss
            best_model = model_row
    return best_model
```

```

def train(epoch, num_epochs, optimizers, trainloader, device, criterion, iou_loss, False):
    n_row, n_col, n_rad = optimizers
    n_row, n_col, n_rad = models
    start = time.time()
    running_loss = 0

    for images, labels in trainloader:
        n_row.train()
        n_col.train()
        n_rad.train()
        labels = torch.stack(labels, 1).float()
        images = images.to(device)
        labels = labels.to(device)

        input = images.unsqueeze(1).float()
        output_row = n_row(input)
        output_col = n_col(input)
        output_rad = n_rad(input)

        n_row.zero_grad()
        n_col.zero_grad()
        n_rad.zero_grad()

        if not iou_loss:
            loss_row = criterion(output_row[:,0], labels[:,0])
            loss_col = criterion(output_col[:,0], labels[:,1])
            loss_rad = criterion(output_rad[:,0], labels[:,2])
            loss_row.backward()
            loss_col.backward()
            loss_rad.backward()
            running_loss += loss_row.item() + loss_col.item() + loss_rad.item()
        else:
            output = torch.stack([output_row, output_col, output_rad], dim=1)
            loss = iou_loss(output, labels.to(device))
            loss.backward()
            running_loss += loss

        n_row.step()
        n_col.step()
        n_rad.step()

    gc.collect()

```

```

def test(models, epoch, num_epochs, testloader, device, criterion):
    n_row, n_col, n_rad = models
    n_row.eval()
    n_col.eval()
    n_rad.eval()

    iou_total = 0
    running_loss = 0
    start = time.time()
    labels_store = []
    outputs_store = []

    for images, labels in testloader:
        labels = torch.stack(labels, 1).float()
        images = images.to(device)
        labels = labels.to(device)

        input = images.unsqueeze(1).float()

        output_row_tensor = n_row(input)
        output_col_tensor = n_col(input)
        output_rad_tensor = n_rad(input)

        if device == torch.device('cpu'):
            output_row = output_row_tensor.cpu().data[0]
            output_col = output_col_tensor.cpu().data[0]
            output_rad = output_rad_tensor.cpu().data[0]
            labels_param = labels.cpu().data[0]
        else:
            output_row = output_row_tensor.data[0]
            output_col = output_col_tensor.data[0]
            output_rad = output_rad_tensor.data[0]
            labels_param = labels.data[0]

        output_param = [output_row, output_col, output_rad]

        loss_row = criterion(output_row_tensor[:,0], labels[:,0])
        loss_col = criterion(output_col_tensor[:,0], labels[:,1])
        loss_rad = criterion(output_rad_tensor[:,0], labels[:,2])

        running_loss += loss_row.item() + loss_col.item() + loss_rad.item()

    iou_total += iou(output_param, labels_param)

    iou_avg = iou_total / len(testloader)
    val_loss = running_loss / len(testloader)
    time_elapsed = time.time() - start
    print('Epoch: %d, Val Avg: %.4f, Val Loss: %.4f, Time: %d, %d & %d' % (
        epoch + 1, num_epochs, iou_avg, val_loss, time_elapsed))

    return outputs_store, labels_store

```

My network visualization:

CNN(

(downconv1): Sequential(

(0): Conv2d(1, 16, kernel\_size=(5, 5), stride=(1, 1), padding=(2, 2))

(1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(2): ReLU()

(3): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

)

(downconv2): Sequential(

(0): Conv2d(16, 32, kernel\_size=(5, 5), stride=(1, 1), padding=(2, 2))

(1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)

(2): ReLU()

(3): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

)

(downconv3): Sequential(

(0): Conv2d(32, 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()

(3): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

)

(downconv4): 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()

(3): MaxPool2d(kernel\_size=2, stride=2, padding=0, dilation=1, ceil\_mode=False)

)

(finalconv): Linear(in\_features=18432, out\_features=1, bias=True)

)



Explanation: The architecture of my model is similar to the Unet without the rf conv and four up convs. The model uses four down convolution layers to reduce the image size and finally use linear transformation to transform the output to 1 class. Similarly, I use BatchNorm to reduce the outstanding individuals and ReLu as an activation function for better gradient propagation after applying convolution and I also use max pooling to reduce image size.

Result: The prediction of the model is very good. For 1000 testing noised circle image, the prediction using my model can easily achieve mean IOU of 0.85. I use 50000 training images, and 20 epochs to train the model

2.3:

```
def iou_loss(outputs, targets, device):
    if device == torch.device("cuda:0"):
        outputs = outputs.cpu().data
        targets = targets.cpu().data
    else:
        outputs = outputs.data
        targets = targets.data
    iou_total = 0
    for i in range(outputs.shape[0]):
        o = outputs[i]
        t = targets[i]

        o_param = (o[0], o[1], o[2])
        t_param = (t[0], t[1], t[2])

        iou_total += iou(o_param, t_param)

    mean = iou_total/outputs.shape[0]
    loss = torch.tensor([mean], requires_grad=True)

    return loss
```

```
def train(epoch, num_epochs, optimizers, trainer_loader, models, device, criterion, iou_loss, for_train):
    a_row, a_col, a_rad = optimizers
    m_row, m_col, m_rad = models
    start = time.time()
    running_loss = 0

    for images, labels in trainer_loader:
        a_row.train()
        a_col.train()
        a_rad.train()
        labels = torch.stack(labels, 1).float()
        images = images.to(device)
        labels = labels.to(device)

        input = images.unsqueeze(1).float()
        output_row = a_row(input)
        output_col = a_col(input)
        output_rad = a_rad(input)

        a_row.zero_grad()
        a_col.zero_grad()
        a_rad.zero_grad()

        if not iou_loss_for:
            loss_row = criterion(output_row[0], labels[:,0])
            loss_col = criterion(output_col[0], labels[:,1])
            loss_rad = criterion(output_rad[0], labels[:,2])
            loss_row.backward()
            loss_col.backward()
            loss_rad.backward()
            running_loss += loss_row.item() + loss_col.item() + loss_rad.item()
        else:
            outputs = torch.stack([output_row, output_col, output_rad], dim=1)
            loss = iou_loss(outputs, labels, device)
            loss.backward()
            running_loss += loss

    a_row.stop()
    a_col.stop()
    a_rad.stop()

    gc.collect()
```

The IOU loss function doesn't work well, and I don't turn it on by default. IOU loss doesn't work well on training simply because it is not differentiable. IOU works by area in the intersection over area in the union. If there is no intersection between the two circles, then the

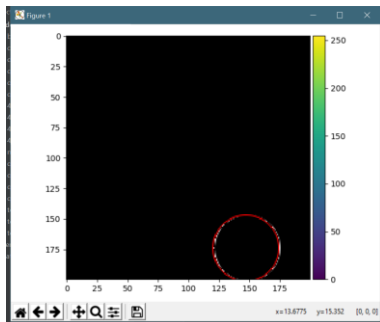
gradient will always be 0. A regular loss function can send back a non-zero gradient to the network parameters when there is an error between the output and the expected output. However, in this case, we have 0 gradient for a completely error case – no intersection between two circles, then there will be no gradient coming from the loss passes back to the weights, so no weights can be updated using gradient descent. Therefore, the model will not learn anything from the loss function.

## 2.4

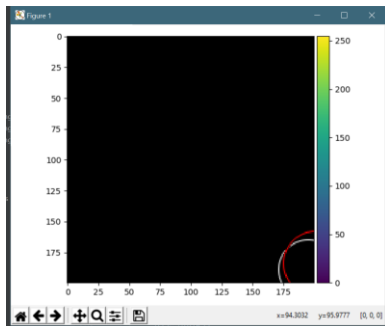
Note: For clear visualization purpose, the output images are without noise. The white circle is ground truth and red circle is prediction.

Main code:

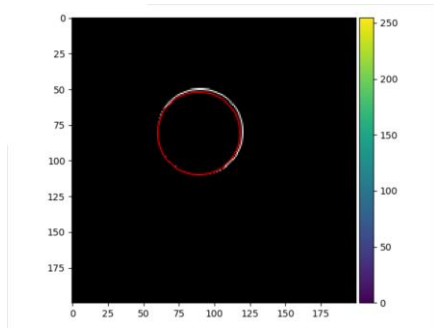
```
def visualize(params, detected):
    row0, col0, rad0 = params
    row1, col1, rad1 = detected
    img = np.zeros((200, 200, 3), dtype=np.float)
    # draw original circle
    draw_circle_color(img, int(row0), int(col0), int(rad0))
    # draw predicted circle
    draw_circle_color(img, int(row1), int(col1), int(rad1), color='red')
    img = (img*255).astype(np.uint8)
    io.imshow(img)
    plt.show()
    print('IOU for this prediction: %.4f' % iou(detected, params))
```



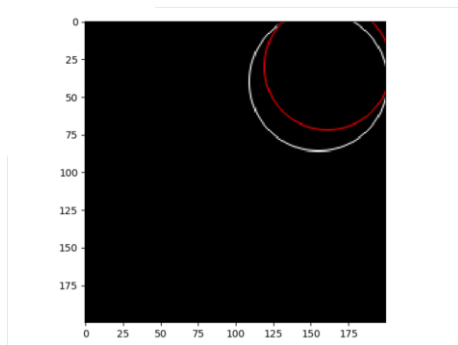
Very good prediction with an IOU of 0.967



A bad prediction with IOU of 0.6532, seems to me that the model fails to predict cleverly when the circle is at the corner



Another good prediction with IOU of 0.9554



Another less accurate prediction with IOU of 0.7079. Again, this circle is closed to the corner and not completely shows on the image, making the prediction harder.

### Question 3:

Since the output of the network is the result of a sigmoid layer, the result scalar  $c$  is between 0 and 1. However, we can't really tell the image is must be a hot dog or must be a cat from the prediction. Although we can assume that the model is well-trained, it still can't completely avoid produce false negative and false positive results. No matter how we train the model, we can't say the model can 100% predict the ground truth, we can only aim as closed to 100% as possible. Therefore, no matter what the result is, we can't conclude the picture is hot dog or not from the provided information.