Deep Learning and Practice Lab3: Diabetic Retinopathy Detection

Zhi-Yi Chin
joycenerd.cs09@nycu.edu.tw

August 2, 2021

1 Introduction

In this lab, we analyzes diabetic retinopathy using ResNet [1], specifically using ResNet18 and ResNet50. We will also compare the results of ResNet18, ResNet50, and both networks with pre-trained weights. We also calculate the confusion matrix between the true label and the predicted label.

2 Experiment Setups

2.1 Data preprocessing

First, we download the data from here. Then we build our custom dataset inherited from `torch.utila.data.Dataset`. In `__init__`, we read in the csv file, which contains image names and corresponding labels and we save these into NumPy array. In `__len__` we define the length of the whole dataset. In `__getitem__` we read in the actual image as PIL Image object and applied data augmentation using `torchvision.transforms`. Specifically we apply data augmentaions like RandomResizedCrop,RandomHorizontalFlip,RandomRotation and RandomHorizontalFlip. At last we trasform the image into tensor and normalize it. The code implementation is in Listing 1 and 2.

```
def getData(mode,csv_dir):
    if mode == 'train':
        img = pd.read_csv(Path(csv_dir).joinpath('train_img.csv'))
        label = pd.read_csv(Path(csv_dir).joinpath('train_label.csv'))
        return np.squeeze(img.values), np.squeeze(label.values)
    else:
        img = pd.read_csv(Path(csv_dir).joinpath('test_img.csv'))
        label = pd.read_csv(Path(csv_dir).joinpath('test_label.csv'))
        return np.squeeze(img.values), np.squeeze(label.values)
```

Listing 1: Get the data by reading the CSV file

```python
class RetinopathyLoader(data.Dataset):
    def __init__(self, root, mode):
        self.root = root
        self.img_name, self.label = getData(mode,
                                            Path(self.root).parent.absolute())
        self.mode = mode
        print("> Found %d images..." % (len(self.img_name)))

    def __len__(self):
        """'return the size of dataset"""
        return len(self.img_name)

    def __getitem__(self, index):
        # read in the actual image
        image_path=Path(self.root).joinpath(self.img_name[index]+".jpeg")
        img=Image.open(image_path).convert('RGB')

        if self.mode=="train":
            data_transform=transforms.Compose([
                transforms.RandomResizedCrop(224),
                transforms.RandomHorizontalFlip(p=0.5),
                transforms.RandomRotation((90,90)),
                transforms.RandomVerticalFlip(p=0.5),
                transforms.ToTensor(),
                transforms.Normalize(mean=[0.485,0.456,0.406],
                std=[0.229,0.224,0.225])
            ])
        else:
            data_transform=transforms.Compose([
                transforms.Resize((224,224)),
                transforms.ToTensor(),
                transforms.Normalize(mean=[0.485,0.456,0.406],
                std=[0.229,0.224,0.225])
            ])

        img=data_transform(img)
        label=torch.from_numpy(np.asarray(self.label[index]))

        return img, label
```

Listing 2: Custom dataset.

2.2 Network implementation

We use ResNet18 and ResNet50 as our network. For pre-trained model we use the model from `torchvision.models`. And then change the last fully connected layer of the model output feature size to 5 since we only have five classes to classify. For the non-pre-trained model, we build our

model from scratch. First, we make an auto padding version of `nn.Conv2D` in Listing 3.

```
class Conv2dAuto(nn.Conv2d):
    def __init__(self,*args,**kwargs):
        """padding version of nn.Conv2d"""
        super().__init__(*args,**kwargs)
        self.padding=(self.kernel_size[0]//2,self.kernel_size[1]//2)
```

Listing 3: Auto padding convolution.

2.2.1 Residual block

ResNet is special because it introduces residual blocks in the network. The residual block takes an input with `in_channels`, applies some blocks of convolution layers to reduce it to `out_channels` and sum it up to the original input. If their sizes mismatch, then the input goes into a `shortcut`, which is defined as a convolution followed by a batch norm layer. Listing 4 shows the residual block of ResNet.

```python
class ResidualBlock(nn.Module):
    def __init__(self,in_channels,out_channels,expansion=1,downsampling=1,
    act='relu',*args,**kwargs):
        super().__init__(*args,**kwargs)
        self.in_channels=in_channels
        self.out_channels=out_channels
        self.expansion=expansion
        self.downsampling=downsampling
        self.act=act_func(act)

        self.blocks=nn.Identity()
        if self.should_apply_shortcut:
            self.shortcut=nn.Sequential(
                nn.Conv2d(self.in_channels,self.expanded_channels,kernel_size=1,
                stride=self.downsampling,bias=False),
                nn.BatchNorm2d(self.expanded_channels)
            )
        else:
            self.shortcut=None

    def forward(self,x):
        residual=x
        if self.should_apply_shortcut:
            residual=self.shortcut(x)
        x=self.blocks(x)
        x+=residual
        return x

    @property
    def should_apply_shortcut(self):
        return self.in_channels!=self.expanded_channels

    @property
    def expanded_channels(self):
        return self.out_channels*self.expansion
```

Listing 4: Residual block.

2.2.2 Basic block

The basic block is a critical component in ResNet. A basic ResNet basic block comprises two layers of 3×3 Conv/batchnorm/relu. The figure of the basic block shows in Figure 1. We first create a handy function to stack one conv and batchnorm layer, which shows in Listing 5. The complete code of the ResNet basic block show in Listing 6.

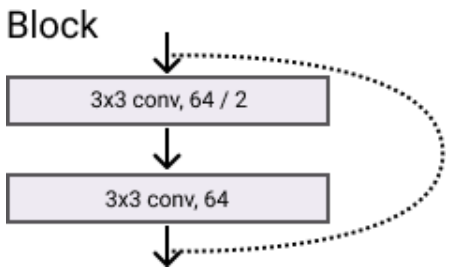

Figure 1: The lines represent the residual operation. The dotted line means that the shortcut was applied to match the input and the output dimension.

```
def conv_bn(in_channels,out_channels,*args,**kwargs):
    conv=partial(Conv2dAuto,bias=False)
    return nn.Sequential(
        conv(in_channels,out_channels,*args,**kwargs),
        nn.BatchNorm2d(out_channels)
    )
```

Listing 5: One conv and batchnorm layer.

```
class BasicBlock(ResidualBlock):
    expansion=1

    def __init__(self,in_channels,out_channels,*args,**kwargs):
        super().__init__(in_channels,out_channels,*args,**kwargs)
        self.blocks=nn.Sequential(
            conv_bn(self.in_channels,out_channels,kernel_size=3,bias=False,
            stride=self.downsampling),
            self.act,
            conv_bn(self.out_channels,self.expanded_channels,kernel_size=3,
            bias=False)
        )
```

Listing 6: ResNet basic block.

2.2.3 Bottleneck block

A bottleneck block is used for increasing the network depth while keeping the size of the parameters as low as possible. The bottleneck block contains $1 \times 1, 3 \times 3, 1 \times 1$ three layers of convolutions, where the 1×1 layers are responsible for reducing and then increasing (restoring) dimensions, leaving the 3×3 layers a bottleneck with smaller input/output dimensions. The ResNet bottleneck block shows in Listing 7.

```python
class BottleNeckBlock(ResidualBlock):
    expansion=4

    def __init__(self,in_channels,out_channels,*args,**kwargs):
        super().__init__(in_channels,out_channels,expansion=4,*args,**kwargs)
        self.blocks=nn.Sequential(
            conv_bn(self.in_channels,self.out_channels,kernel_size=1),
            self.act,
            conv_bn(self.out_channels,self.out_channels,kernel_size=3,
            stride=self.downsampling),
            self.act,
            conv_bn(self.out_channels,self.expanded_channels,kernel_size=1)
        )
```

Listing 7: ResNet bottleneck block.

2.2.4 ResNet layer

A ResNet layer is composed of the same blocks stacked one after another, as shown in Figure 2. We can easily define the ResNet layer by just stack `n` blocks one after the other; just remember that the first convolution block has a stride of two since we perform downsampling directly by convolutional layers that have a stride of 2. Listing 8 shows the code implementation for the ResNet layer.

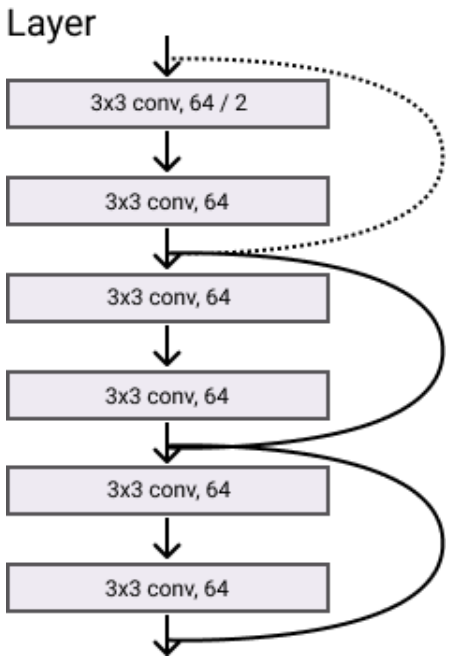

Figure 2: ResNet layer.

```python
class ResNetLayer(nn.Module):
    def __init__(self,in_channels,out_channels,block=BasicBlock,n=1,*args,
                 **kwargs):
        super().__init__()
        downsampling=2 if in_channels!=out_channels else 1
        block_list=[]
        block_list.append(block(in_channels,out_channels,*args,**kwargs,
                          downsampling=downsampling))
        for _ in range(n-1):
            block_list.append(block(out_channels*block.expansion,out_channels,
                              downsampling=1,*args,**kwargs))
        self.blocks=nn.Sequential(*block_list)

    def forward(self,x):
        x=self.blocks(x)
        return x
```

Listing 8: ResNet layer.

2.2.5 Encoder

An encoder is composed of multiple ResNet layers at increasing features size. The code is shown in Listing 9.

```python
class Encoder(nn.Module):
    def __init__(self,in_channels=3,blocks_sizes=[64,128,256,512],
                 deepths=[2,2,2,2],act='relu',block=BasicBlock,*args,**kwargs):
        super().__init__()
        self.blocks_size=blocks_sizes
        self.act=act_func(act)

        self.gate=nn.Sequential(
            nn.Conv2d(in_channels,self.blocks_size[0],kernel_size=7,stride=2,
                      padding=3,bias=False),
            nn.BatchNorm2d(self.blocks_size[0]),
            self.act,
            nn.MaxPool2d(kernel_size=3,stride=2,padding=1)
        )

        block_list=[]
        block_list.append(ResNetLayer(blocks_sizes[0],blocks_sizes[0],
                          n=deepths[0],act=act,block=block,*args,**kwargs))
        for i in range(1,len(self.blocks_size)):
            block_list.append(ResNetLayer(blocks_sizes[i-1]*block.expansion,
                              blocks_sizes[i],n=deepths[i],act=act,block=block,
                              *args,**kwargs))
        self.blocks=nn.Sequential(*block_list)

    def forward(self,x):
        x=self.gate(x)
        x=self.blocks(x)
        return x
```

Listing 9: ResNet encoder.

2.2.6 Decoder

The decoder is the last piece we need to create the full network, a linear classifier. It is a fully connected layer that maps the features learned by the network to their respective classes. The implementation is in Listing 10.

```python
class Decoder(nn.Module):
    def __init__(self,in_features,n_classes):
        super().__init__()
        self.avg_pool=nn.AdaptiveAvgPool2d((1,1))
        self.decoder=nn.Linear(in_features,n_classes)

    def forward(self,x):
        x=self.avg_pool(x)
        x=x.view(x.shape[0],-1)
        x=self.decoder(x)
        return x
```

Listing 10: ResNet decoder.

2.2.7 ResNet

We can put all the pieces together and create the final ResNet network, which is shown in Listing 11. Since we modulize every part, so the final code is clean a reader-friendly. Then we defined ResNet18 and ResNet50 as two separate functions shown in Listing 12. The two differences between the two models are: (1) ResNet18 uses the basic block, and ResNet50 uses the bottleneck block. (2) The depths of each `block_size` is different.

```python
class ResNet(nn.Module):
    def __init__(self,in_channels,n_classes,*args,**kwargs):
        super().__init__()
        self.encoder=Encoder(in_channels,*args,**kwargs)
        self.decoder=Decoder(self.encoder.blocks[-1].blocks[-1].expanded_channels,
                             n_classes)

    def forward(self,x):
        x=self.encoder(x)
        x=self.decoder(x)
        return x
```

Listing 11: ResNet.

```python
def resnet18(in_channels,n_classes,block=BasicBlock,*args,**kwargs):
    return ResNet(in_channels,n_classes,block=block,deepths=[2,2,2,2],*args,
                  **kwargs)

def resnet50(in_channels,n_classes,block=BottleNeckBlock,*args,**kwargs):
    return ResNet(in_channels,n_classes,block=block,deepths=[3,4,6,3],*args,
                  **kwargs)
```

Listing 12: ResNet18 and ResNet50.

2.3 Confusion matrix

A confusion matrix is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known. We use `sklearn.metrix.confusion_matrix` to get the values in the confusion matrix and plot it with `seaborn.heatmap`. The code is shown in Listing 13.

```
print(args)
input("Press ENTER if no problem...")

save_name="resnet50_confusion_mat"

gt,pred=test()

labels=[0,1,2,3,4]
cm=confusion_matrix(gt,pred,labels,normalize='true')

plt.rcParams["font.family"] = "serif"
fig,ax=plt.subplots()
sns.heatmap(cm, annot=True, ax=ax, cmap='Blues', fmt='.1f')
ax.set_xlabel('Prediction')
ax.set_ylabel('Ground truth')
ax.xaxis.set_ticklabels(labels, rotation=45)
ax.yaxis.set_ticklabels(labels, rotation=0)
plt.title('ResNet50 Normalized comfusion matrix')
plt.savefig(f"./results/{save_name}.jpg")
```

Listing 13: Confusion matrix.

3 Experimental Results

Table 1 shows the comparison results. These results are trained in the following setting

- Initial lr: 0.001.
- Image resize to 256×256.
- Batch size: 32.
- Use SGD as optimizer with momentum 0.9 and weight_decay 5e-4.
- Use StepLR as scheduler after 4 epochs with step_size 2 and gamma 0.5.
- 10 epochs of training.

	pretrained	train_acc	test_acc
ResNet50	✓	82.89%	82.18%
ResNet50	✗	73.51%	73.55%
ResNet18	✓	81.36%	79.24%
ResNet18	✗	73.48%	73.35%

Table 1: Accuracy comparision of ResNet50 (with pretrained), ResNet50 (w/o pretrained), ResNet18 (with pretrained), ResNet18 (w/o pretrained).

3.1 Accuracy curve

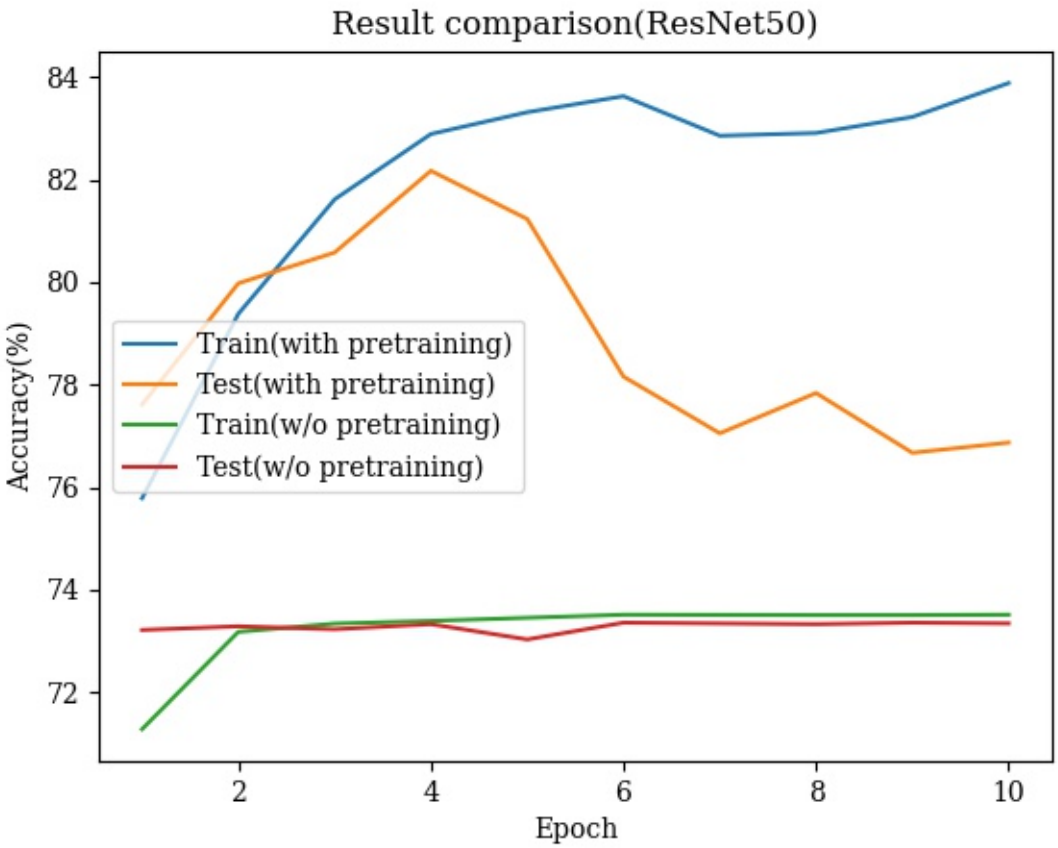

Figure 3: ResNet50 accuracy curve

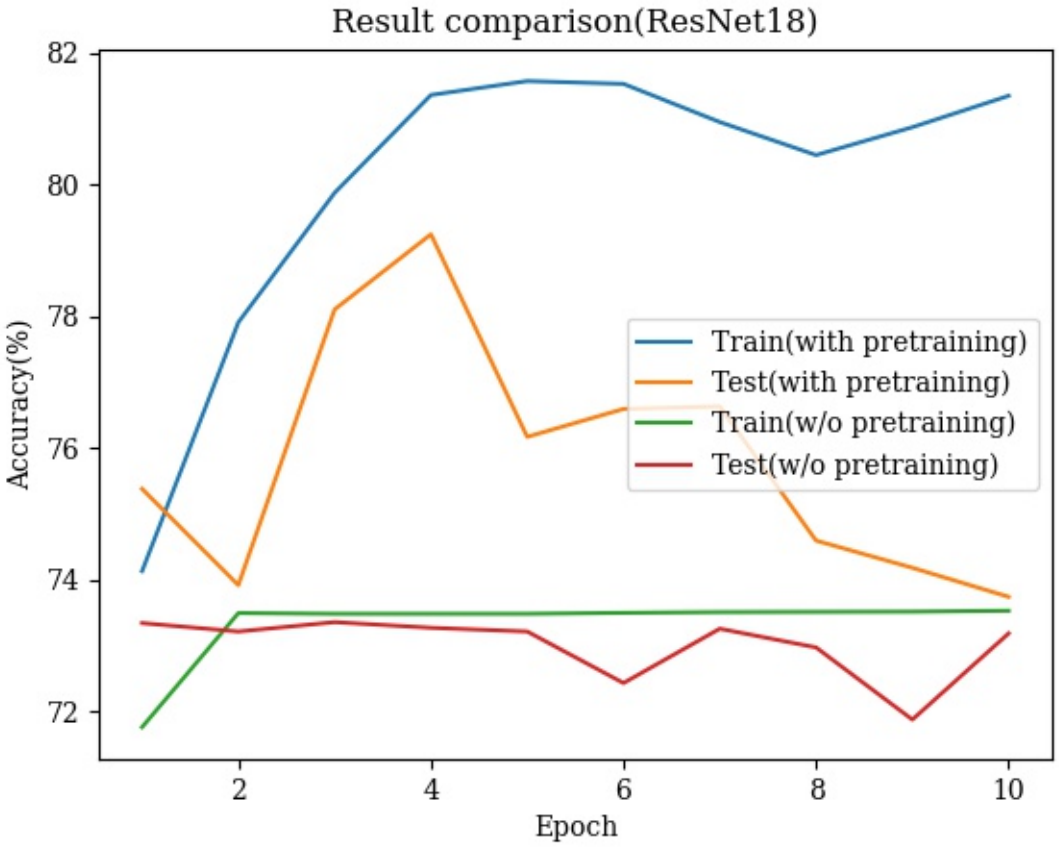

Figure 4: ResNet18 accuracy curve

3.2 Confusion matrix

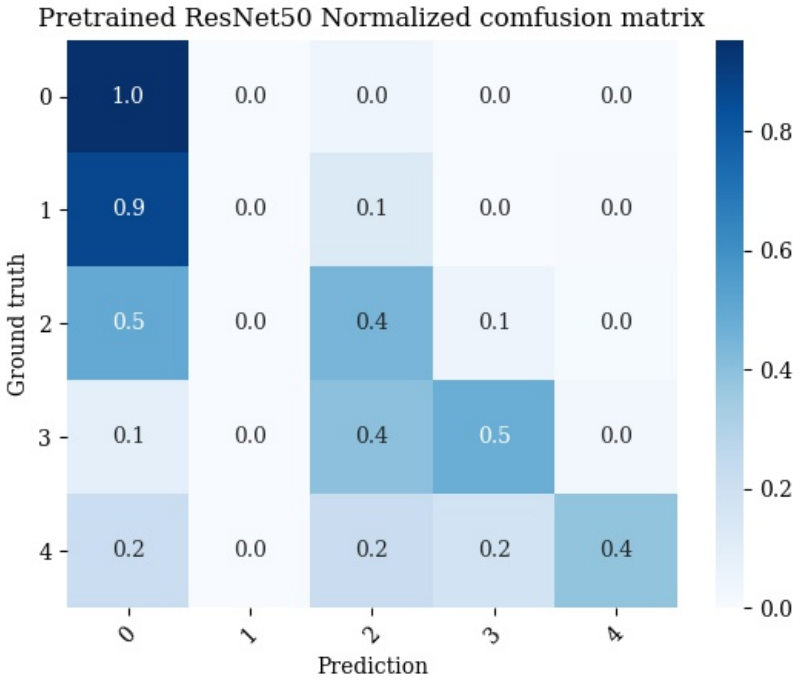

Figure 5: The confusion matrix of ResNet50 (with pretraining).

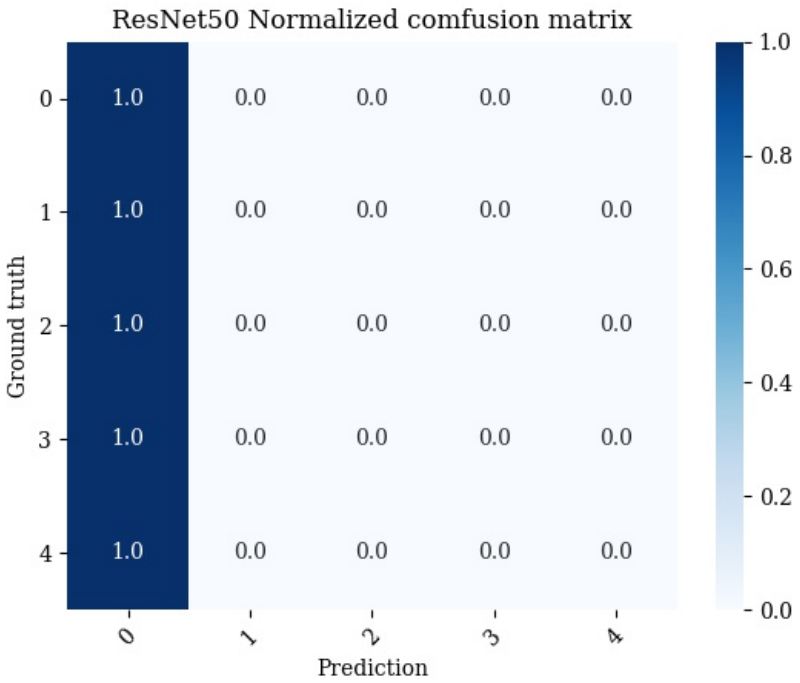

Figure 6: The confusion matrix of ResNet50 (w/o pretraining).

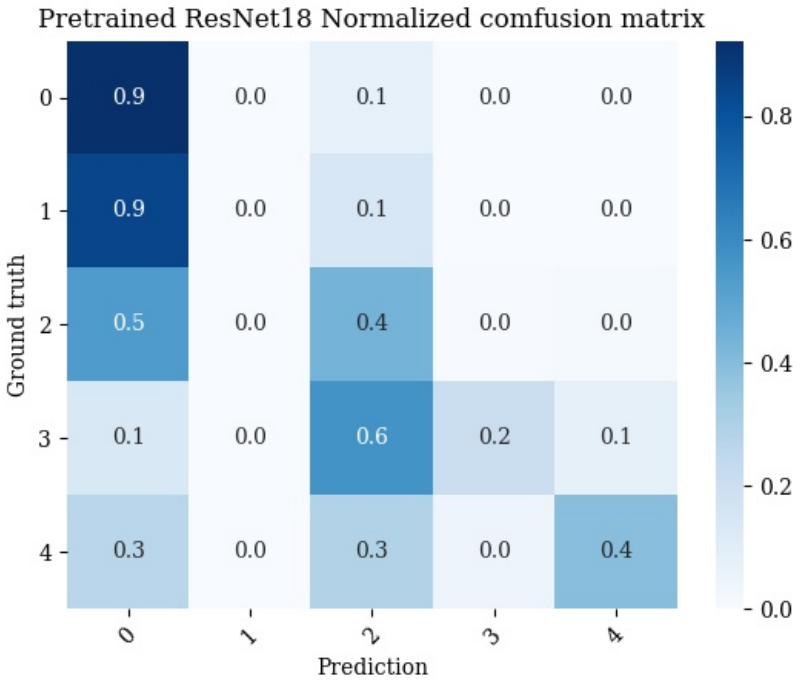

Figure 7: The confusion matrix of ResNet18 (with pretraining).

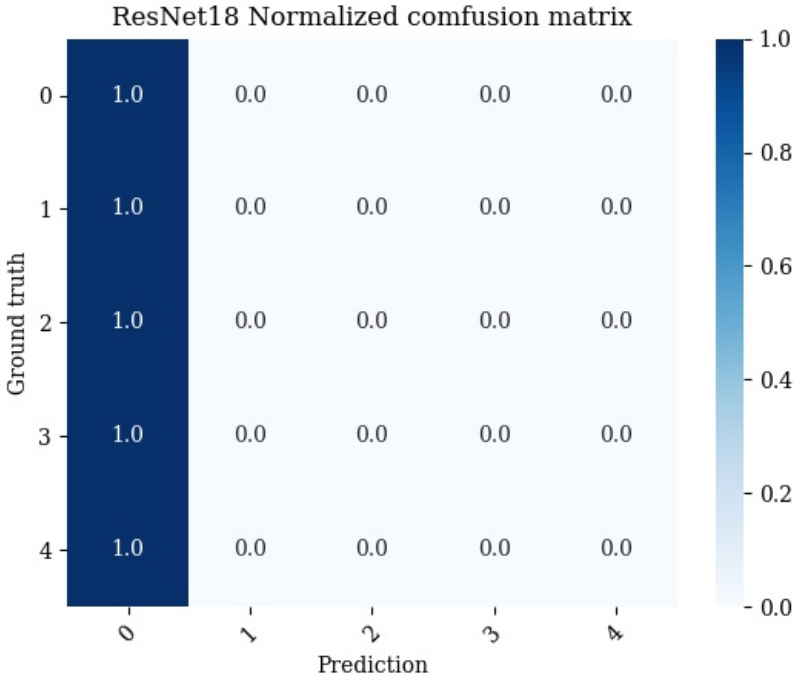

Figure 8: The confusion matrix of ResNet18 (w/o pretraining).

4 Discussion

From the experimental results, we discover three interesting points worth discussing. (1) Model with pretrained weight performs much better than the model without pretrained weight. The pretrained weight is pre-trained on ImageNet, which is nowhere similar to our data (medical image), but still using the pre-trained model performs better. We think this is because the features captured from the model pre-trained on the 1000-class ImageNet dataset might not be suitable for this specific dataset. However, the model did learn some general low-level features. (2) Both ResNet50 and ResNet18 without pretrained performs poorly in this data. From the confusion matrix, we can see that no matter what image comes in, the prediction of these two models is class 0. Although we know there is a big problem with the data, this prediction is unacceptable because this means the model does not learn anything. (3) We find that our data suffer from severe imbalance. Most of the data we have is in class 0, and other classes have much less than class 0. This affects the results of all the models. Only class 0 performs well, but other classes suffer.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.