Lab4: Diabetic Retinopathy Detection

Wei-Yun Hsu
Institute of Multimedia Engineering
National Yang Ming Chiao Tung University

1. Introduction

In this lab, we will need to analysis diabetic retinopathy (糖尿病視網膜病變). There are some requirements as follows:

- Implement your own custom DataLoader through PyTorch framework.
- Implement ResNet18, ResNet50 and compare them between the pretrained model and without pretraining in same architectures.
- Visualize the accuracy trend.
- Calculate the confusion matrix and plotting.

A. Dataset

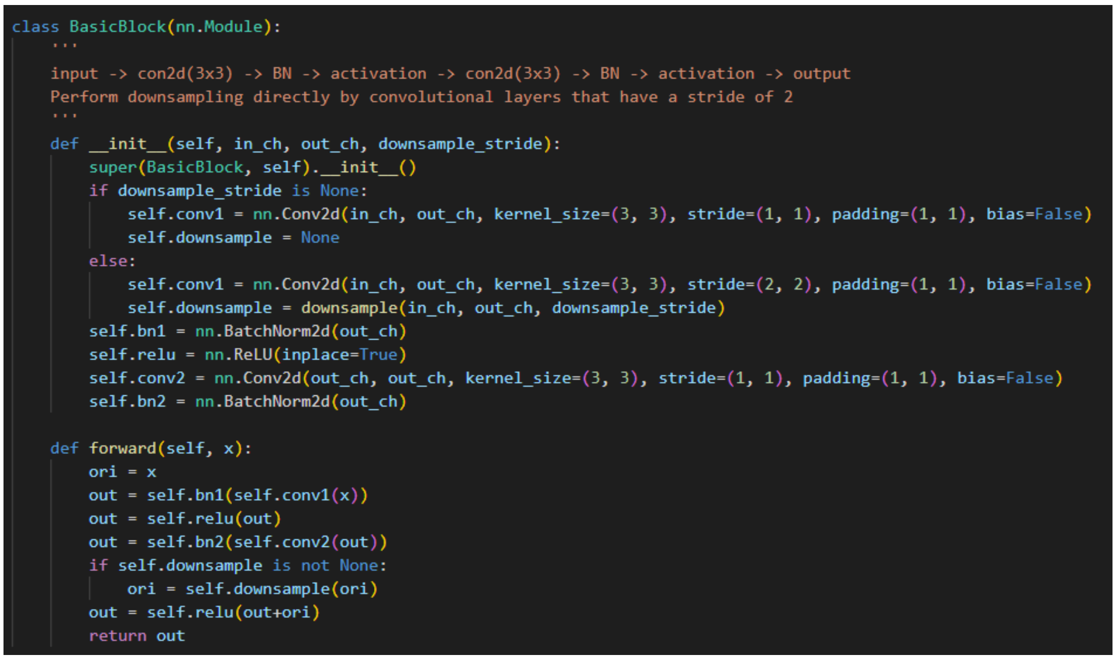
The diabetic retinopathy dataset contains 35124 images, 28100 images for training and 7026 for testing. The images' resolutions are different and are required to be preprocessed into the same resolution, which is 512 * 512.

2. Experiment setups

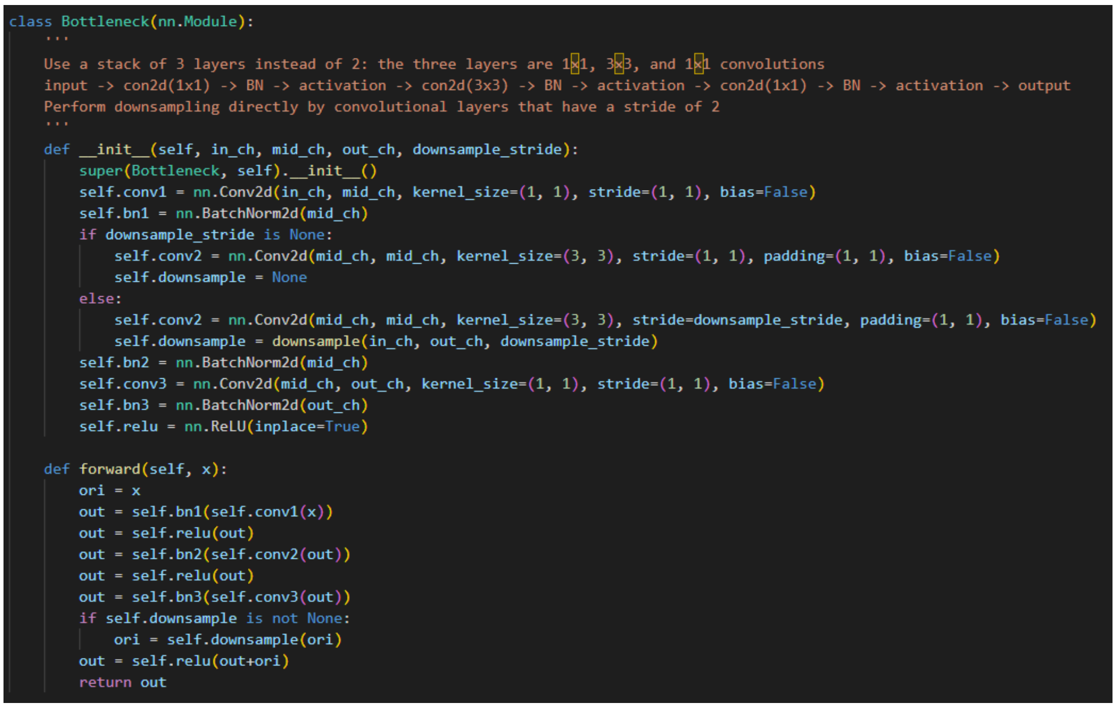
A. The details of your model (ResNet)

ResNet18 and ResNet50 are designed to address the problem of vanishing gradients in very deep neural networks. Both models use residual blocks, it consists of two or three convolutional layers with a skip connection. This skip connection allows the gradient to flow directly through the block, which can help alleviate the vanishing gradient problem. In the implementation, there are two types of residual block, BasicBlock and Bottleneck, they are used in ResNet18 and ResNet50, respectively. The key difference between BasicBlock and Bottleneck is the number of convolutional layers and the size of the filters used within each block, both designs have similar time complexity.

In ResNet18, BasicBlock is used, which consists of two convolutional layers with 3x3 filters, followed by a batch normalization layer and a ReLU activation function. This block is repeated multiple times with skip connections between them to form the full network.
In ResNet50, Bottleneck architecture is used, which includes three convolutional layers with 1x1, 3x3 and 1x1 filters respectively in each block. We can know that it is convenient to change dimensions using a 1x1 network structure, and this architecture reduces the number of parameters needed while still achieving high accuracy.
The implementation of BasicBlock and Bottleneck as follows.

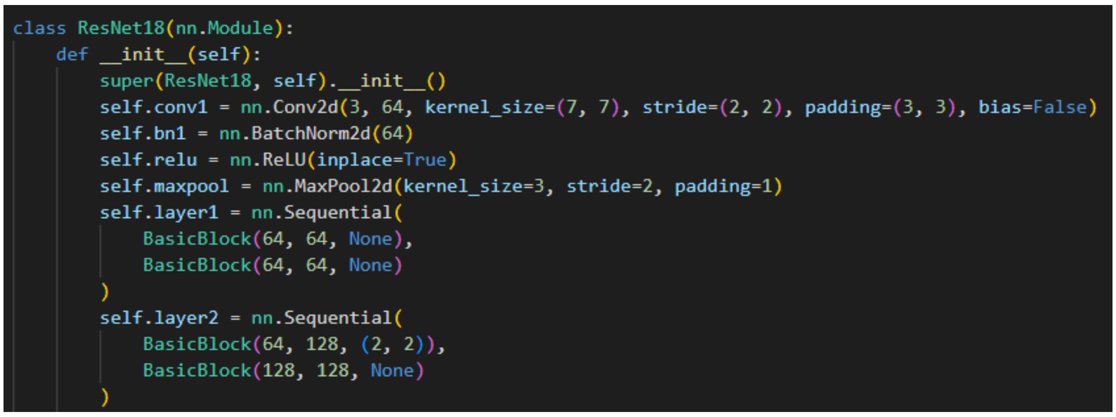


```python
class BasicBlock(nn.Module):
    '''
    input -> con2d(3x3) -> BN -> activation -> con2d(3x3) -> BN -> activation -> output
    Perform downsampling directly by convolutional layers that have a stride of 2
    '''
    def __init__(self, in_ch, out_ch, downsample_stride):
        super(BasicBlock, self).__init__()
        if downsample_stride is None:
            self.conv1 = nn.Conv2d(in_ch, out_ch, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
            self.downsample = None
        else:
            self.conv1 = nn.Conv2d(in_ch, out_ch, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
            self.downsample = downsample(in_ch, out_ch, downsample_stride)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_ch, out_ch, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.bn2 = nn.BatchNorm2d(out_ch)

    def forward(self, x):
        ori = x
        out = self.bn1(self.conv1(x))
        out = self.relu(out)
        out = self.bn2(self.conv2(out))
        if self.downsample is not None:
            ori = self.downsample(ori)
        out = self.relu(out+ori)
        return out
```

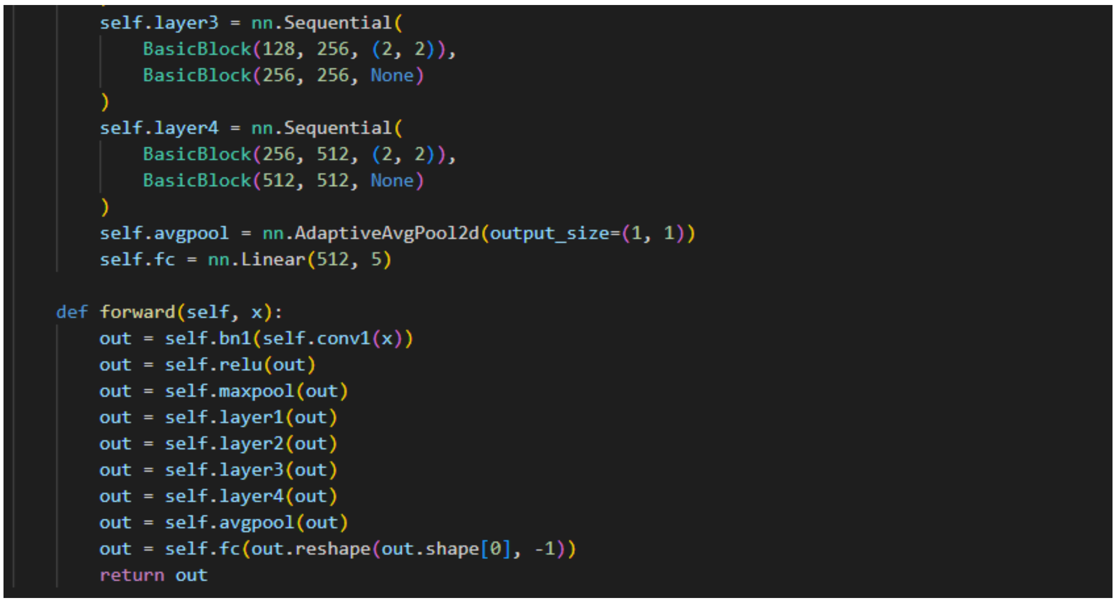


```python
class Bottleneck(nn.Module):
    '''
    Use a stack of 3 layers instead of 2: the three layers are 1×1, 3×3, and 1×1 convolutions
    input -> con2d(1x1) -> BN -> activation -> con2d(3x3) -> BN -> activation -> con2d(1x1) -> BN -> activation -> output
    Perform downsampling directly by convolutional layers that have a stride of 2
    '''
    def __init__(self, in_ch, mid_ch, out_ch, downsample_stride):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(in_ch, mid_ch, kernel_size=(1, 1), stride=(1, 1), bias=False)
        self.bn1 = nn.BatchNorm2d(mid_ch)
        if downsample_stride is None:
            self.conv2 = nn.Conv2d(mid_ch, mid_ch, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
            self.downsample = None
        else:
            self.conv2 = nn.Conv2d(mid_ch, mid_ch, kernel_size=(3, 3), stride=downsample_stride, padding=(1, 1), bias=False)
            self.downsample = downsample(in_ch, out_ch, downsample_stride)
        self.bn2 = nn.BatchNorm2d(mid_ch)
        self.conv3 = nn.Conv2d(mid_ch, out_ch, kernel_size=(1, 1), stride=(1, 1), bias=False)
        self.bn3 = nn.BatchNorm2d(out_ch)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        ori = x
        out = self.bn1(self.conv1(x))
        out = self.relu(out)
        out = self.bn2(self.conv2(out))
        out = self.relu(out)
        out = self.bn3(self.conv3(out))
        if self.downsample is not None:
            ori = self.downsample(ori)
        out = self.relu(out+ori)
        return out
```

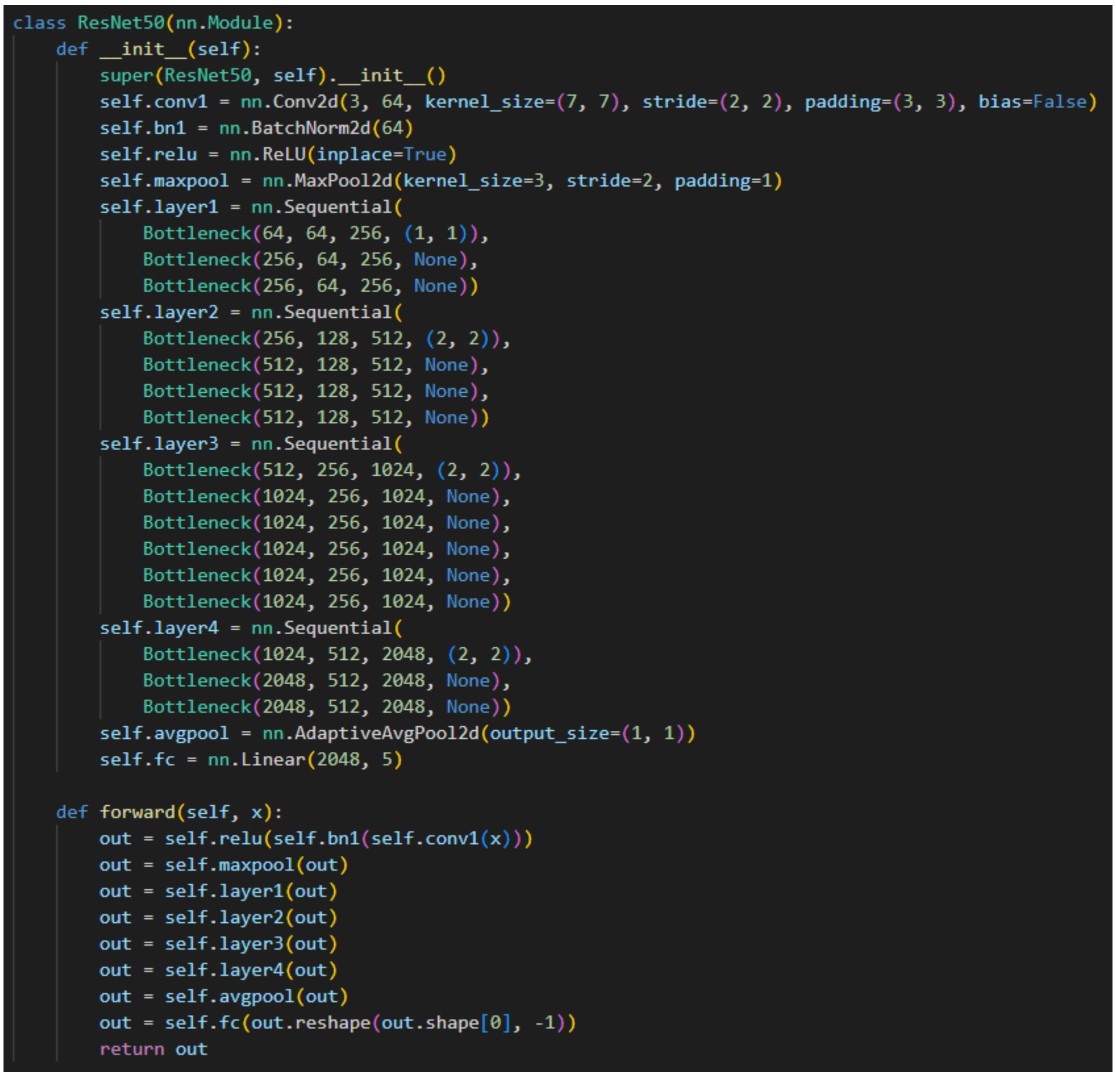
The implementations of ResNet18 and ResNet50 are shown below.



```python
class ResNet18(nn.Module):
    def __init__(self):
        super(ResNet18, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = nn.Sequential(
            BasicBlock(64, 64, None),
            BasicBlock(64, 64, None)
        )
        self.layer2 = nn.Sequential(
            BasicBlock(64, 128, (2, 2)),
            BasicBlock(128, 128, None)
        )
```



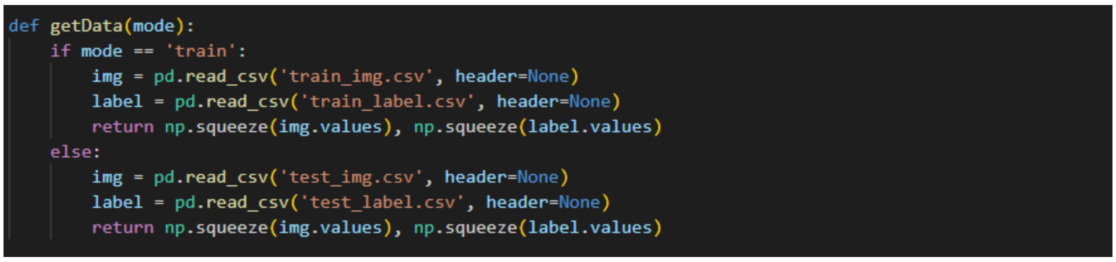
```python
        self.layer3 = nn.Sequential(
            BasicBlock(128, 256, (2, 2)),
            BasicBlock(256, 256, None)
        )
        self.layer4 = nn.Sequential(
            BasicBlock(256, 512, (2, 2)),
            BasicBlock(512, 512, None)
        )
        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc = nn.Linear(512, 5)

    def forward(self, x):
        out = self.bn1(self.conv1(x))
        out = self.relu(out)
        out = self.maxpool(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avgpool(out)
        out = self.fc(out.reshape(out.shape[0], -1))
        return out
```



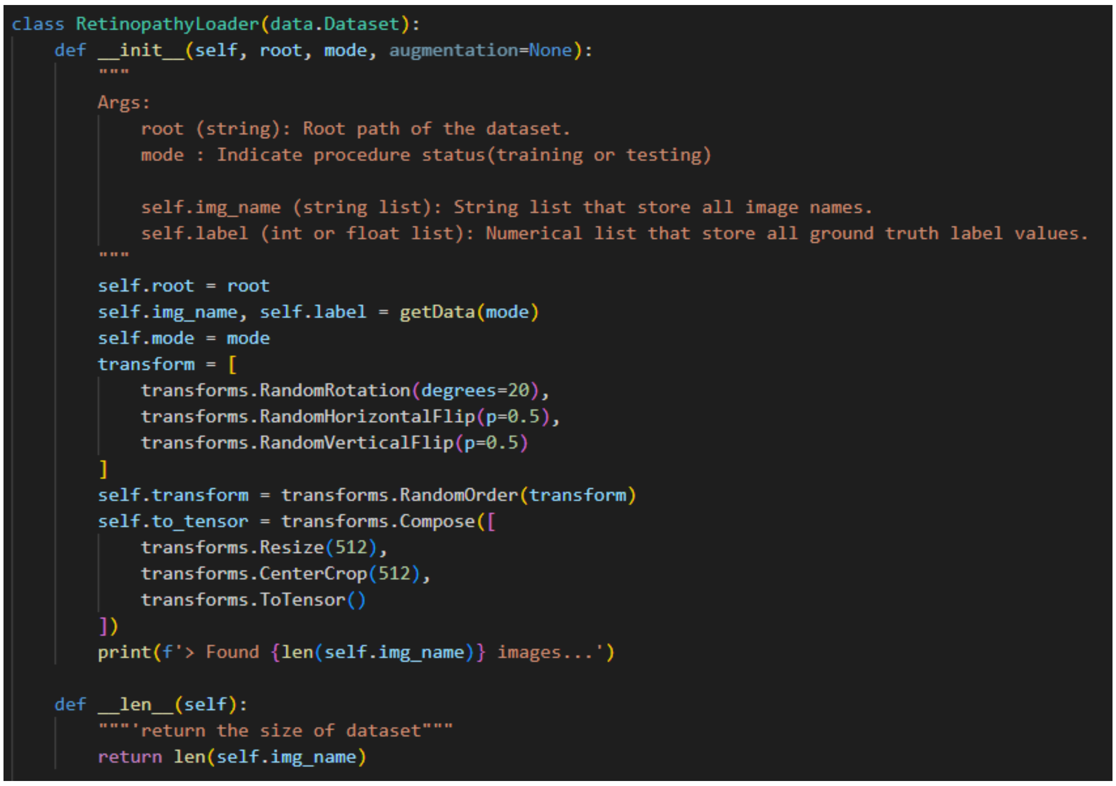
```python
class ResNet50(nn.Module):
    def __init__(self):
        super(ResNet50, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = nn.Sequential(
            Bottleneck(64, 64, 256, (1, 1)),
            Bottleneck(256, 64, 256, None),
            Bottleneck(256, 64, 256, None))
        self.layer2 = nn.Sequential(
            Bottleneck(256, 128, 512, (2, 2)),
            Bottleneck(512, 128, 512, None),
            Bottleneck(512, 128, 512, None),
            Bottleneck(512, 128, 512, None))
        self.layer3 = nn.Sequential(
            Bottleneck(512, 256, 1024, (2, 2)),
            Bottleneck(1024, 256, 1024, None),
            Bottleneck(1024, 256, 1024, None),
            Bottleneck(1024, 256, 1024, None),
            Bottleneck(1024, 256, 1024, None),
            Bottleneck(1024, 256, 1024, None))
        self.layer4 = nn.Sequential(
            Bottleneck(1024, 512, 2048, (2, 2)),
            Bottleneck(2048, 512, 2048, None),
            Bottleneck(2048, 512, 2048, None))
        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc = nn.Linear(2048, 5)

    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.maxpool(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avgpool(out)
        out = self.fc(out.reshape(out.shape[0], -1))
        return out
```

B. The details of your Dataloader

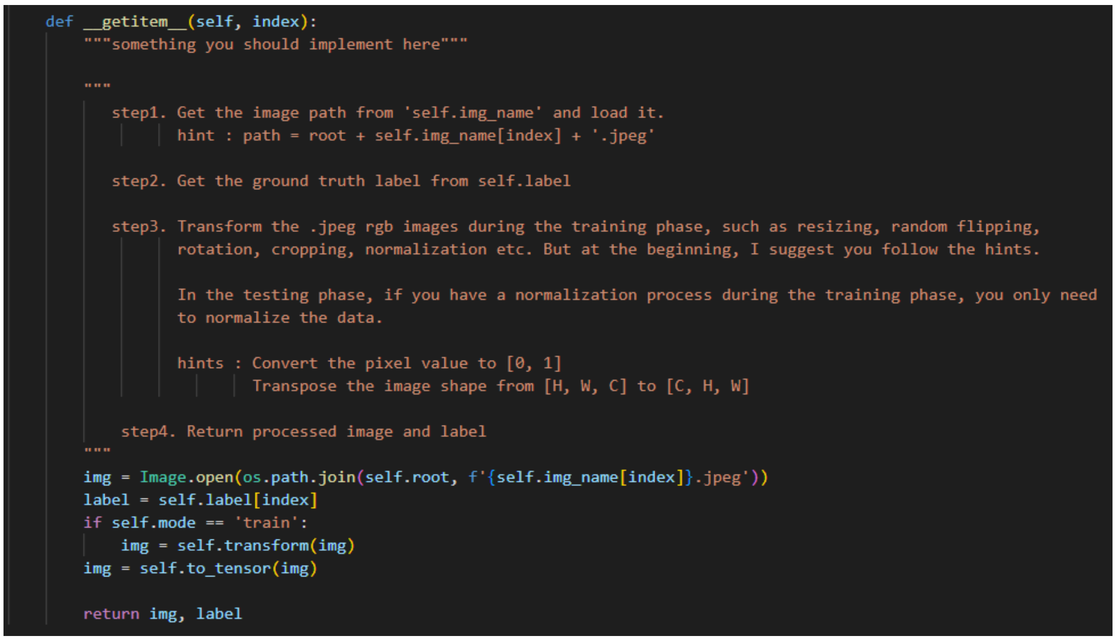
We define a custom dataloader and fill __init__(), __get_item__(), __len__() functions with the provided skeleton. We resize the all data to 512×512 because the images' resolution are different and apply the data augmentation to training data to avoid overfitting. The details of the dataloader as follows.



```python
def getData(mode):
    if mode == 'train':
        img = pd.read_csv('train_img.csv', header=None)
        label = pd.read_csv('train_label.csv', header=None)
        return np.squeeze(img.values), np.squeeze(label.values)
    else:
        img = pd.read_csv('test_img.csv', header=None)
        label = pd.read_csv('test_label.csv', header=None)
        return np.squeeze(img.values), np.squeeze(label.values)
```



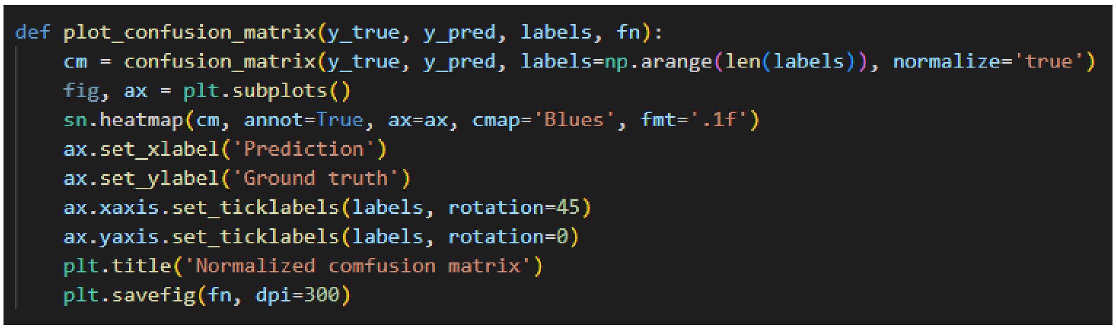
```python
class RetinopathyLoader(data.Dataset):
    def __init__(self, root, mode, augmentation=None):
        """
        Args:
            root (string): Root path of the dataset.
            mode : Indicate procedure status(training or testing)

            self.img_name (string list): String list that store all image names.
            self.label (int or float list): Numerical list that store all ground truth label values.
        """
        self.root = root
        self.img_name, self.label = getData(mode)
        self.mode = mode
        transform = [
            transforms.RandomRotation(degrees=20),
            transforms.RandomHorizontalFlip(p=0.5),
            transforms.RandomVerticalFlip(p=0.5)
        ]
        self.transform = transforms.RandomOrder(transform)
        self.to_tensor = transforms.Compose([
            transforms.Resize(512),
            transforms.CenterCrop(512),
            transforms.ToTensor()
        ])
        print(f'> Found {len(self.img_name)} images...')

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



```python
    def __getitem__(self, index):
        """something you should implement here"""

        """
           step1. Get the image path from 'self.img_name' and load it.
                  hint : path = root + self.img_name[index] + '.jpeg'

           step2. Get the ground truth label from self.label

           step3. Transform the .jpeg rgb images during the training phase, such as resizing, random flipping,
                  rotation, cropping, normalization etc. But at the beginning, I suggest you follow the hints.

                  In the testing phase, if you have a normalization process during the training phase, you only need
                  to normalize the data.

                  hints : Convert the pixel value to [0, 1]
                          Transpose the image shape from [H, W, C] to [C, H, W]

            step4. Return processed image and label
        """
        img = Image.open(os.path.join(self.root, f'{self.img_name[index]}.jpeg'))
        label = self.label[index]
        if self.mode == 'train':
            img = self.transform(img)
        img = self.to_tensor(img)

        return img, label
```

C. Describing your evaluation through the confusion matrix

We use confusion_matrix functions provided by scikit-learn to calculate the confusion matrix. The implementation of plotting confusion matrix is shown below.



```python
def plot_confusion_matrix(y_true, y_pred, labels, fn):
    cm = confusion_matrix(y_true, y_pred, labels=np.arange(len(labels)), normalize='true')
    fig, ax = plt.subplots()
    sn.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('Normalized confusion matrix')
    plt.savefig(fn, dpi=300)
```

D. Experiment setup

The hyper-parameters for experiments as shown below.

- batch size: 8, 16, 32
- learning rate: 0.001
- epochs: 10, 20
- optimizer: SGD
- loss function: cross entropy
- activation function: ReLU

3. Data preprocessing

A. How you preprocessed your data?

Generally, we usually think that images should be cropped first and then resize them when we need to preprocess image data to let them have the same resolution, since this process won't change the original image into an oval shape. But after my experiments, I found that if we change these two steps, it would get the same results. The main difference between the above preprocessing orders is the input of the `torchvision.transforms.Resize()` function, the former one is a sequence (ex. transforms.Resize((512, 512))) and the latter one is an int value (ex. transforms.Resize(512)). If the size of the function input is in int format then the size of the image will keep the aspect ratio. Therefore, I used the latter order as my preprocessing method. Second, we need to transpose the image shape from [H, W, C] to [C, H, W] because the shape of convolutional layer in PyTorch is [C, H, W]. I convert PIL image to tenser by `torchvision.transforms.ToTensor()`. During the training phase, I also apply the data augmentation to improve test accuracy and avoid overfitting. For this purpose, I use the random flipping and rotation to do data augmentation.

B. What makes your method special?

In addition to the above data preprocessing methods, I also tried to use the `torchvision.transforms.RandomOrder()` to apply a random ordering of a given list of transformations to the input data during training. When training with a fixed set of transformations in a fixed order, it is possible for the model to memorize the specific patterns and features of the transformed data rather than learning to extract more general features that can be applied to new data.
This approach can prevent the model from becoming too specialized to any particular transformation or sequence of transformations. It is also useful to increase the diversity of the data seen during training, which can help to improve the robustness of the model and prevent overfitting.

4. Experimental results

A. The highest testing accuracy

	w/o pretraining	with pretraining
ResNet18	74.52%	84.73%
ResNet50	73.20%	84.79%

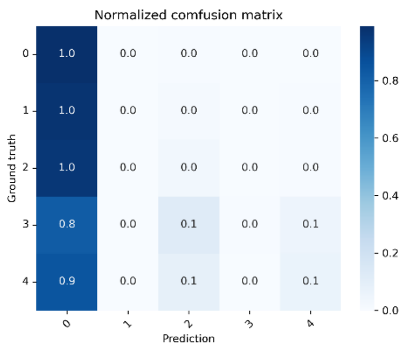
The hyper-parameters of the above results are as follows.

	w/o pretraining	with pretraining
ResNet18	batch size=32	batch size=16
ResNet50	batch size=16	batch size=8

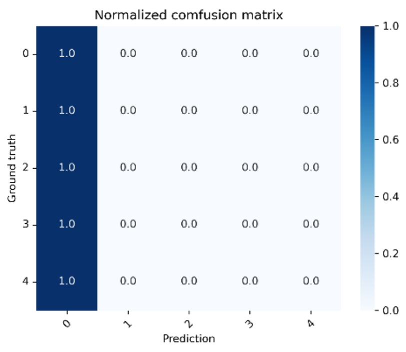
B. Comparison figures

B-1. Confusion matrix

- **ResNet18 (without pretraining)**

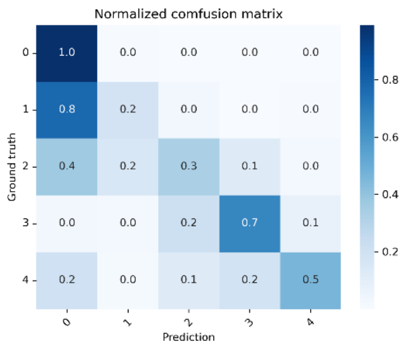


(a) batch size = 8

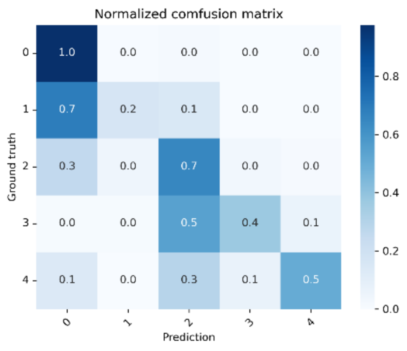


(b) batch size = 16

- **ResNet18 (with pretraining)**

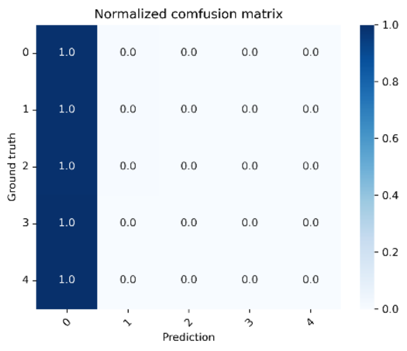


(a) batch size = 8

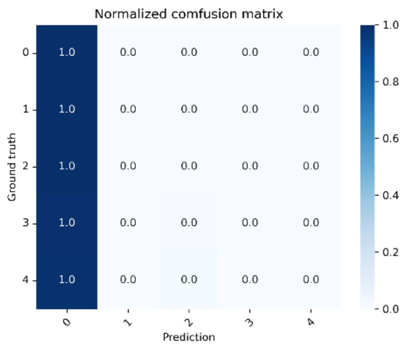


(b) batch size = 16

- **ResNet50 (without pretraining)**

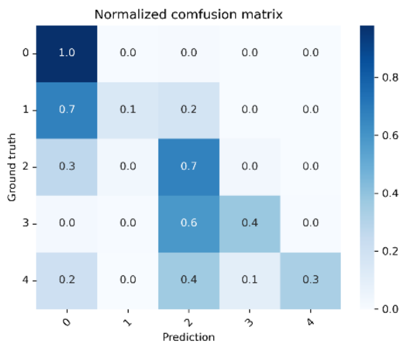


(a) batch size = 8

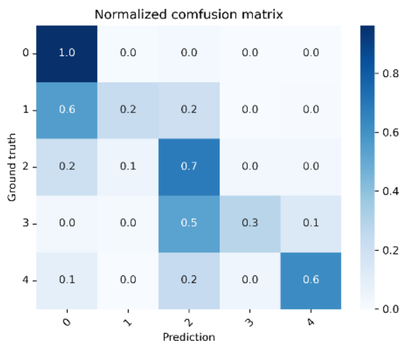


(b) batch size = 16

- **ResNet50 (with pretraining)**



(a) batch size = 8

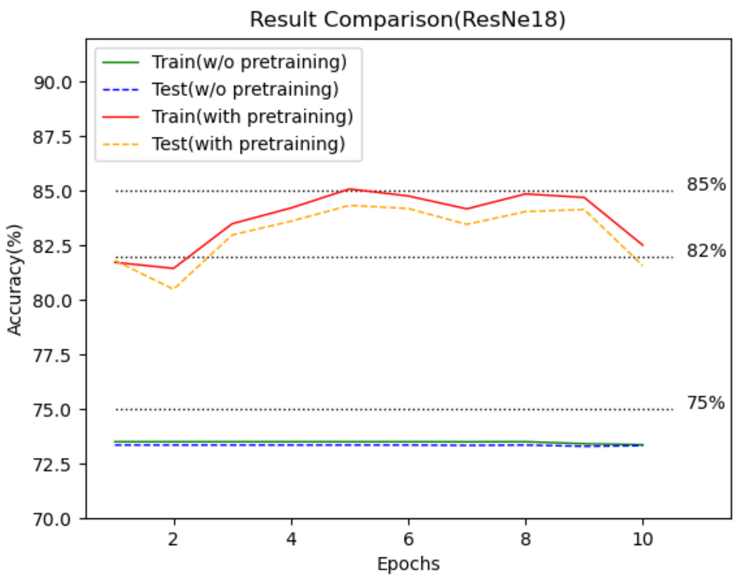


(b) batch size = 16

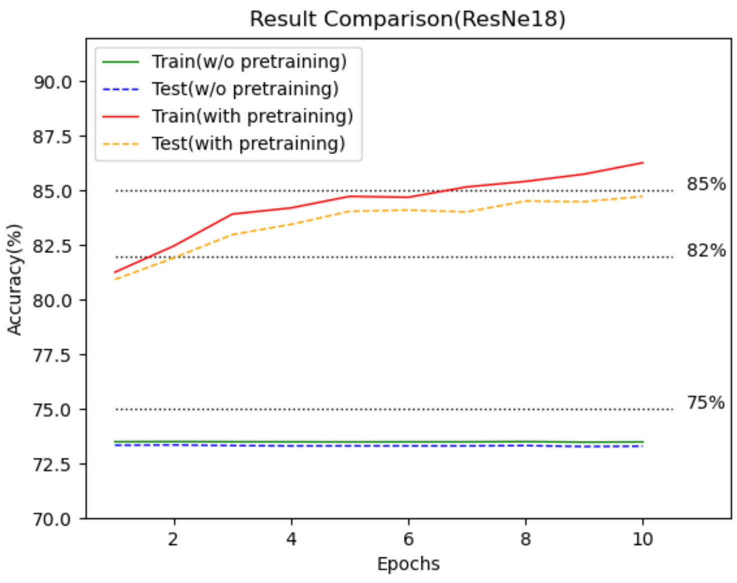
B-2. Accuracy trend

I draw the dotted line to mark the accuracy of 75%, 82% and 85%.

- **ResNet18**

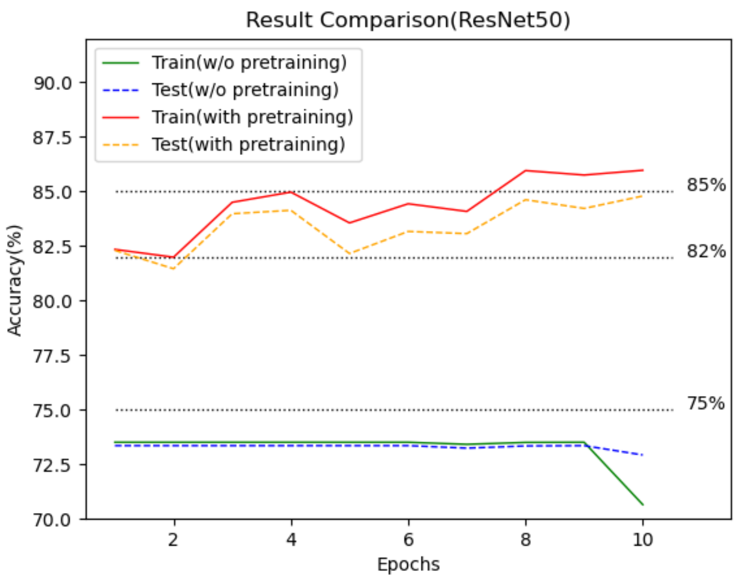


(a) batch size = 8

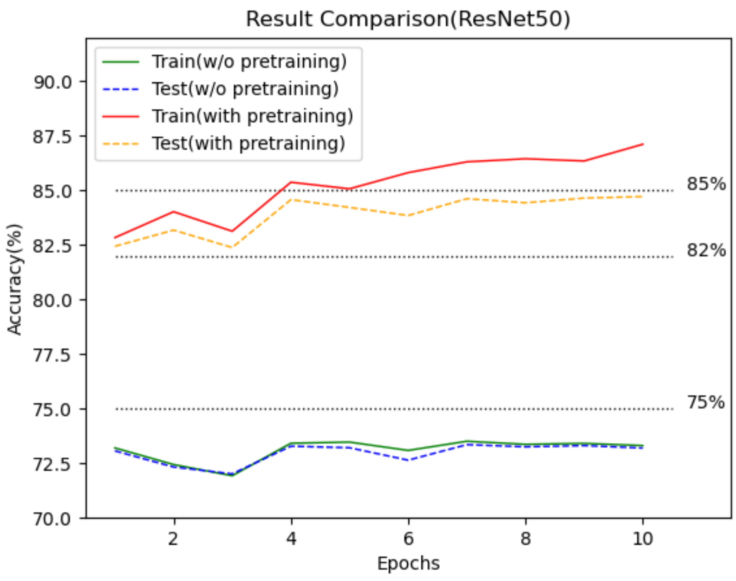


(b) batch size = 16

- **ResNet50**



(a) batch size = 8



(b) batch size = 16

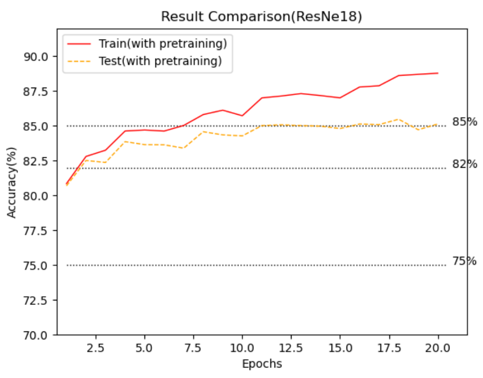
5. Discussion

A. ResNet18 v.s. ResNet50

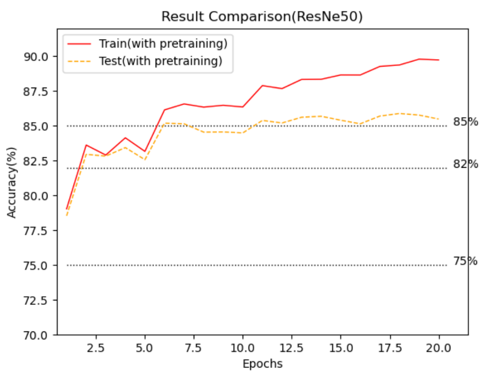
From the previous section, we can see that there are no obvious difference on the performance between the ResNet18 and ResNet50. They tend to classify all datasets as not having diabetic retinopathy when using models without pretraining. Another interesting thing is that when using pretrained models, they will classify class 1 as class 0, it means that the models will misclassify “no diabetic retinopathy” and “mild diabetic retinopathy”.

B. Other training strategy to beat and improve the test accuracy

In order to achieve higher accuracy on testing set, I tried to use the scheduler of learning rate decay in the training phase and more epochs to train the pretrained model. Because the learning rate is a hyperparameter that determines the step size of the optimization algorithm during training, and it can have a significant impact on the speed and quality of convergence. For example, the gradients used to update the model parameters can be noisy or fluctuate due to various factors such as the size of the mini-batch, the choice of optimization algorithm, or the nature of the data. By using learning rate decay, we can smooth out these fluctuations and help to ensure that the model converges to a good solution.



(a) ResNet18 with pretraining



(b) ResNet50 with pretraining

As we can see, there is a slight improvement in performance, the test accuracy can up to more than 85%. The results have shown that learning rate decay is a powerful technique that can help to improve the convergence, generalization, and robustness of neural network models during training.