0716325 曾正豪 DLP Lab4-2

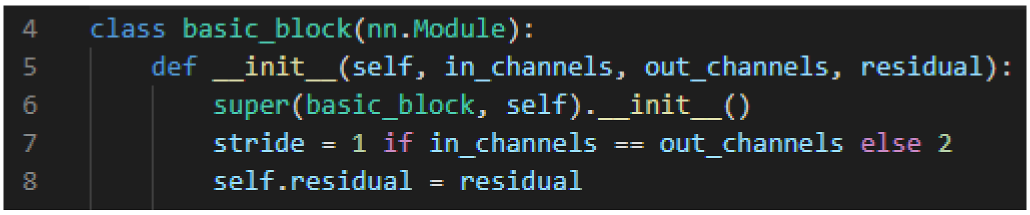
I. Introduction

In this lab, I implement the ResNet18 and ResNet50 by pytorch on python. Then apply these models on Diabetic Retinopathy Detection dataset. In the experiment, I use some data augmentation skills such as rotation and flip for the training data. After the training process is finished, I use confusion matrix to help me check how the models work on the test set. In the end, I get 80.89% accuracy on the test set.

II. Experiment setups

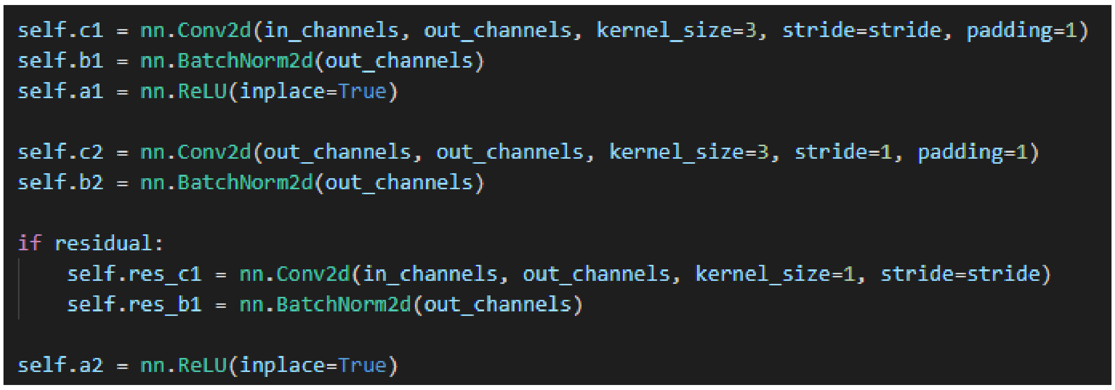
A. The details of your model (ResNet)

In my implementation, I implement the basic block and bottleneck block first. Then, use these blocks to construct the ResNet.



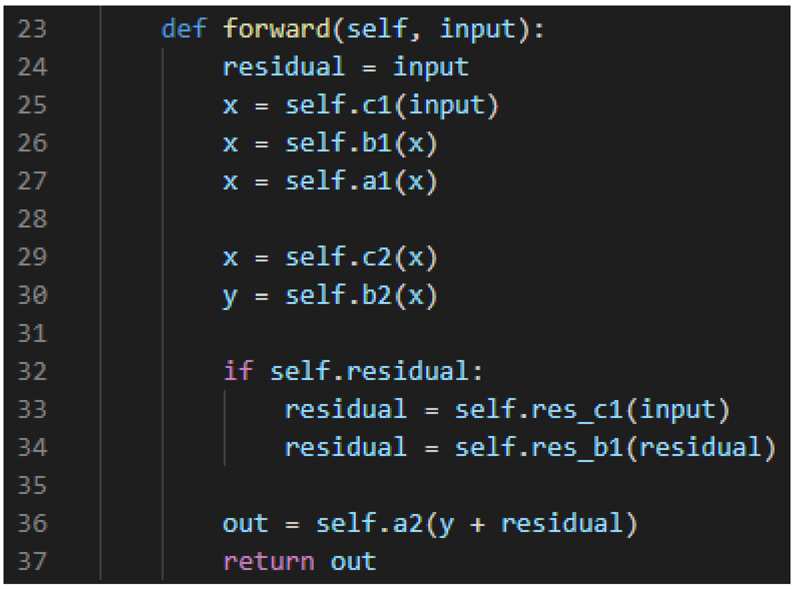
```python
class basic_block(nn.Module):
    def __init__(self, in_channels, out_channels, residual):
        super(basic_block, self).__init__()
        stride = 1 if in_channels == out_channels else 2
        self.residual = residual
```

In the constructor of basic block, I first check whether the in_channel is same to out_channel to decide whether to down sample the input data in the residual path.



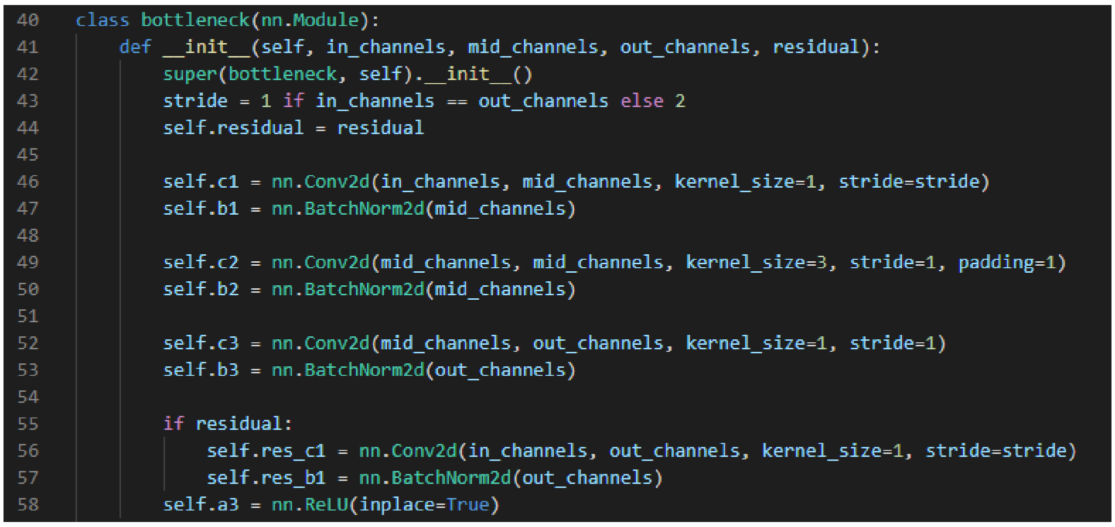
```python
self.c1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1)
self.b1 = nn.BatchNorm2d(out_channels)
self.a1 = nn.ReLU(inplace=True)

self.c2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1)
self.b2 = nn.BatchNorm2d(out_channels)

if residual:
    self.res_c1 = nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride)
    self.res_b1 = nn.BatchNorm2d(out_channels)

self.a2 = nn.ReLU(inplace=True)
```

Then following the architecture of basic block to construct each layer.



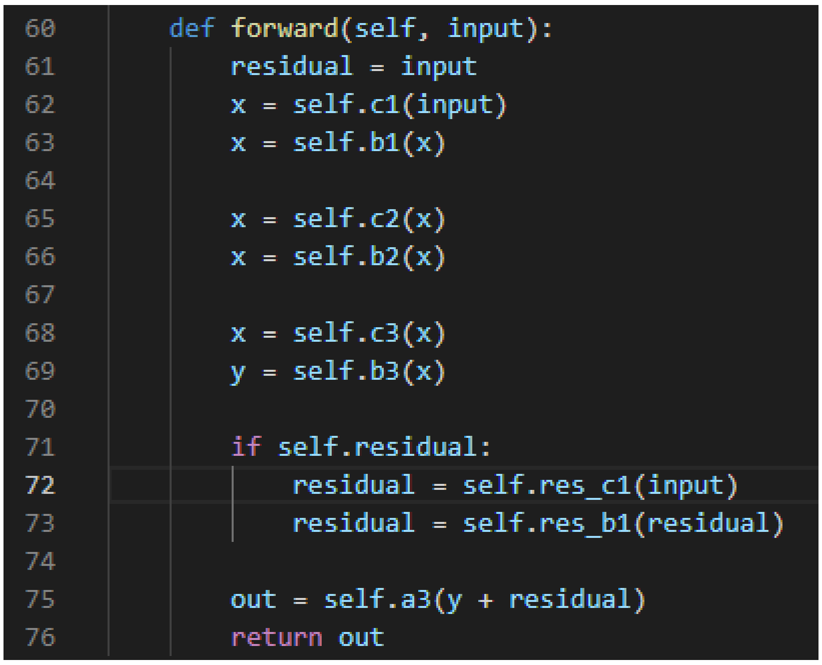
```python
    def forward(self, input):
        residual = input
        x = self.c1(input)
        x = self.b1(x)
        x = self.a1(x)

        x = self.c2(x)
        y = self.b2(x)

        if self.residual:
            residual = self.res_c1(input)
            residual = self.res_b1(residual)

        out = self.a2(y + residual)
        return out
```

In the forward function of basic block, I would copy the input data to residual. Then put the input to the layers in the basic block. If the block needs down sampling, the residual would use the convolution to down sample. After that, the residual would be added to the output and go through the *relu* activation function.



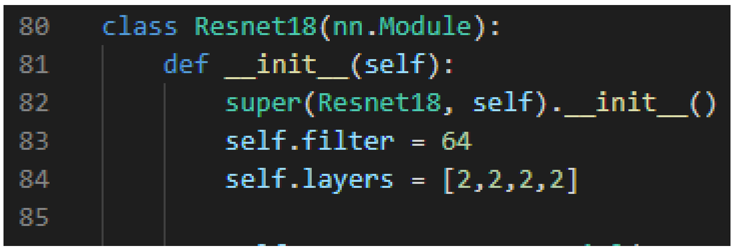
```python
class bottleneck(nn.Module):
    def __init__(self, in_channels, mid_channels, out_channels, residual):
        super(bottleneck, self).__init__()
        stride = 1 if in_channels == out_channels else 2
        self.residual = residual

        self.c1 = nn.Conv2d(in_channels, mid_channels, kernel_size=1, stride=stride)
        self.b1 = nn.BatchNorm2d(mid_channels)

        self.c2 = nn.Conv2d(mid_channels, mid_channels, kernel_size=3, stride=1, padding=1)
        self.b2 = nn.BatchNorm2d(mid_channels)

        self.c3 = nn.Conv2d(mid_channels, out_channels, kernel_size=1, stride=1)
        self.b3 = nn.BatchNorm2d(out_channels)

        if residual:
            self.res_c1 = nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride)
            self.res_b1 = nn.BatchNorm2d(out_channels)
        self.a3 = nn.ReLU(inplace=True)
```

In the constructor of bottleneck block, I first check whether the in_channel is same to out_channel to decide whether to down sample the input data in the residual path. Then following the architecture of bottleneck block to construct each layer.



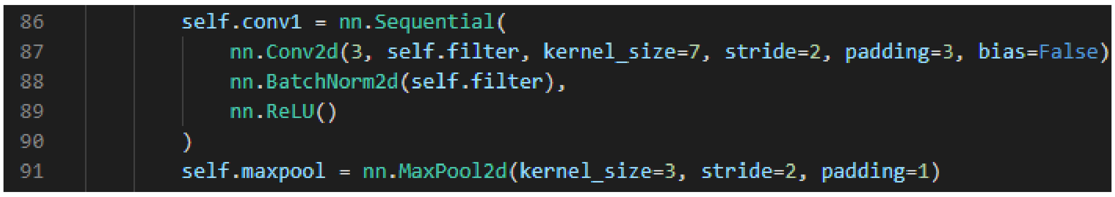
```
    def forward(self, input):
        residual = input
        x = self.c1(input)
        x = self.b1(x)

        x = self.c2(x)
        x = self.b2(x)

        x = self.c3(x)
        y = self.b3(x)

        if self.residual:
            residual = self.res_c1(input)
            residual = self.res_b1(residual)

        out = self.a3(y + residual)
        return out
```

In the forward function of bottleneck block, I would copy the input data to residual. Then feed the input to the layers in the basic block. If the block needs down sampling, the residual would use the convolution to down sample. After that, the residual would be added to the output and go through the *relu* activation function.



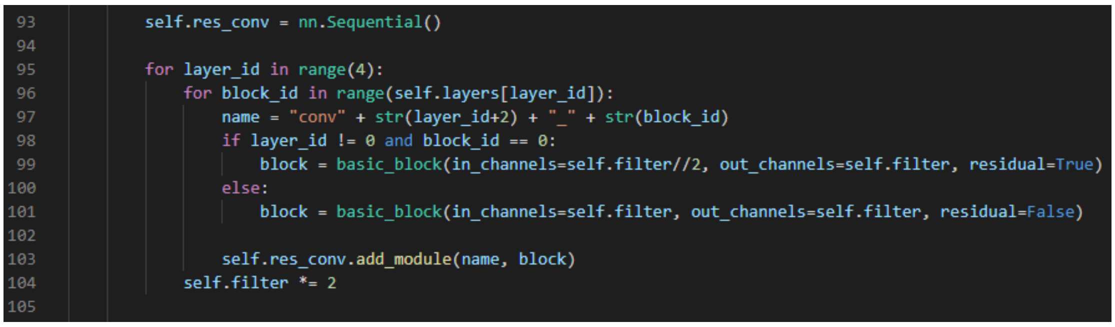
```
class Resnet18(nn.Module):
    def __init__(self):
        super(Resnet18, self).__init__()
        self.filter = 64
        self.layers = [2,2,2,2]

```

In the ResNet18, I first initialize the number of filter to 64 and the number of block in each stage to [2,2,2,2].



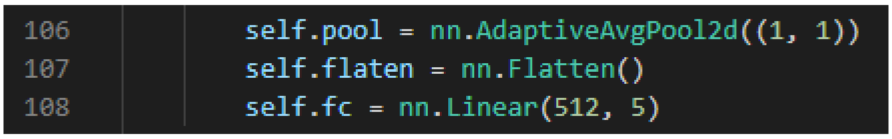
```
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, self.filter, kernel_size=7, stride=2, padding=3, bias=False)
            nn.BatchNorm2d(self.filter),
            nn.ReLU()
        )
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
```

In the very beginning of resnet18, it need a convolution layer with kernel size 7, strides 2, and padding 3. And a max pooling layer with kernel size 3 and stride 2.



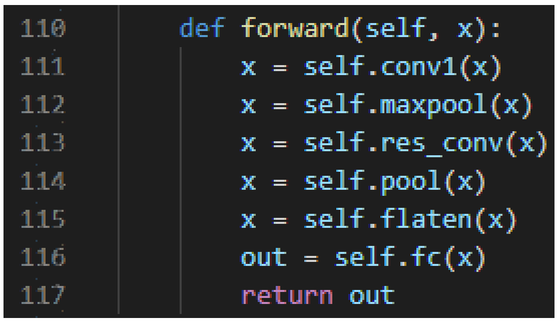
```python
        self.res_conv = nn.Sequential()

        for layer_id in range(4):
            for block_id in range(self.layers[layer_id]):
                name = "conv" + str(layer_id+2) + "_" + str(block_id)
                if layer_id != 0 and block_id == 0:
                    block = basic_block(in_channels=self.filter//2, out_channels=self.filter, residual=True)
                else:
                    block = basic_block(in_channels=self.filter, out_channels=self.filter, residual=False)

                self.res_conv.add_module(name, block)
            self.filter *= 2

```

In the following stage, I use the sequential model to construct. From the paper of ResNet, we can find that only first basic block of 2nd to 4th stage need down sampling. So, I can use this information to construct each basic block. And the number of channels would be double in each stage.



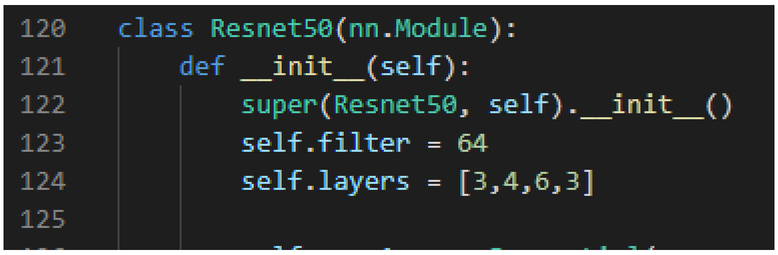
```python
        self.pool = nn.AdaptiveAvgPool2d((1, 1))
        self.flaten = nn.Flatten()
        self.fc = nn.Linear(512, 5)
```

In the end, the net needs to do the average pooling to the size of (1, 1). Currently, the size of data is (512, 1, 1), then use the flatten layer to flatten it to a vector of length 512 and send it to a fully connected layer to classify.



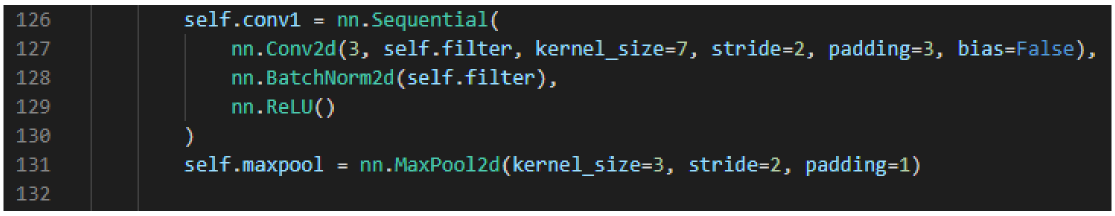
```python
    def forward(self, x):
        x = self.conv1(x)
        x = self.maxpool(x)
        x = self.res_conv(x)
        x = self.pool(x)
        x = self.flaten(x)
        out = self.fc(x)
        return out
```

In the forward function of resnet18, just call each layer in order.



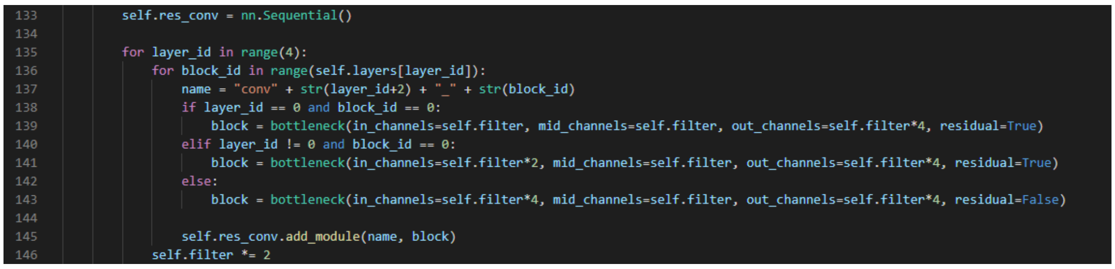
```python
class Resnet50(nn.Module):
    def __init__(self):
        super(Resnet50, self).__init__()
        self.filter = 64
        self.layers = [3,4,6,3]

```

In the ResNet50, I first initialize the number of filter to 64 and the number of block in each stage to [3,4,6,3].



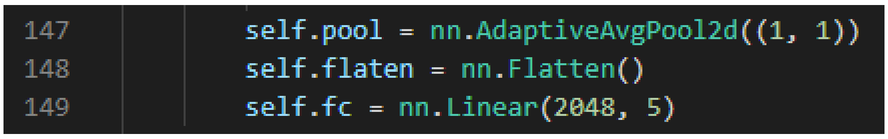
```python
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, self.filter, kernel_size=7, stride=2, padding=3, bias=False),
            nn.BatchNorm2d(self.filter),
            nn.ReLU()
        )
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

```

In the very beginning of resnet50, it need a convolution layer with kernel size 7, strides 2 , and padding 3. And a max pooling layer with kernel size 3 and stride 2.



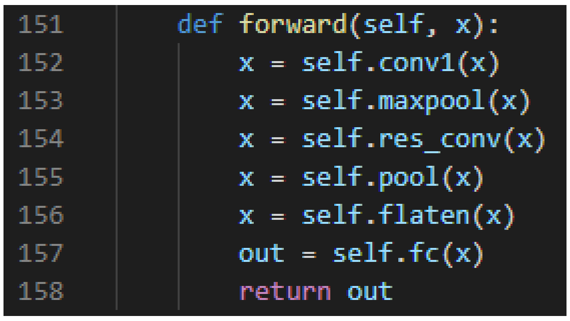
```python
        self.res_conv = nn.Sequential()

        for layer_id in range(4):
            for block_id in range(self.layers[layer_id]):
                name = "conv" + str(layer_id+2) + "_" + str(block_id)
                if layer_id == 0 and block_id == 0:
                    block = bottleneck(in_channels=self.filter, mid_channels=self.filter, out_channels=self.filter*4, residual=True)
                elif layer_id != 0 and block_id == 0:
                    block = bottleneck(in_channels=self.filter*2, mid_channels=self.filter, out_channels=self.filter*4, residual=True)
                else:
                    block = bottleneck(in_channels=self.filter*4, mid_channels=self.filter, out_channels=self.filter*4, residual=False)

                self.res_conv.add_module(name, block)
            self.filter *= 2
```

In the following stage, I use the sequential model to construct. From the paper of resnet, we can find that only first basic block of each stage need down sampling. So, I can use this information to construct each basic block. And the number of channels would be double in each stage.



```python
        self.pool = nn.AdaptiveAvgPool2d((1, 1))
        self.flaten = nn.Flatten()
        self.fc = nn.Linear(2048, 5)
```

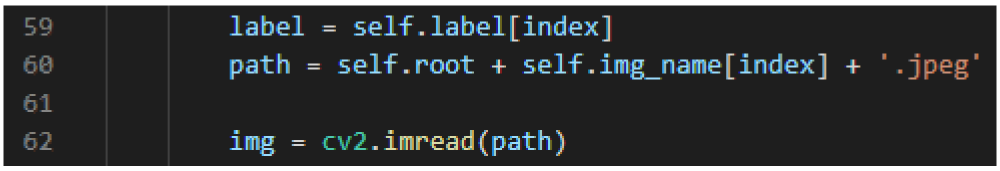
In the end, the net needs to do the average pooling to the size of (1, 1). Currently, the size of data is (512, 1, 1), then use the flatten layer to flatten it to a vector of length 512 and send it to a fully connected layer to classify.



```python
    def forward(self, x):
        x = self.conv1(x)
        x = self.maxpool(x)
        x = self.res_conv(x)
        x = self.pool(x)
        x = self.flaten(x)
        out = self.fc(x)
        return out
```

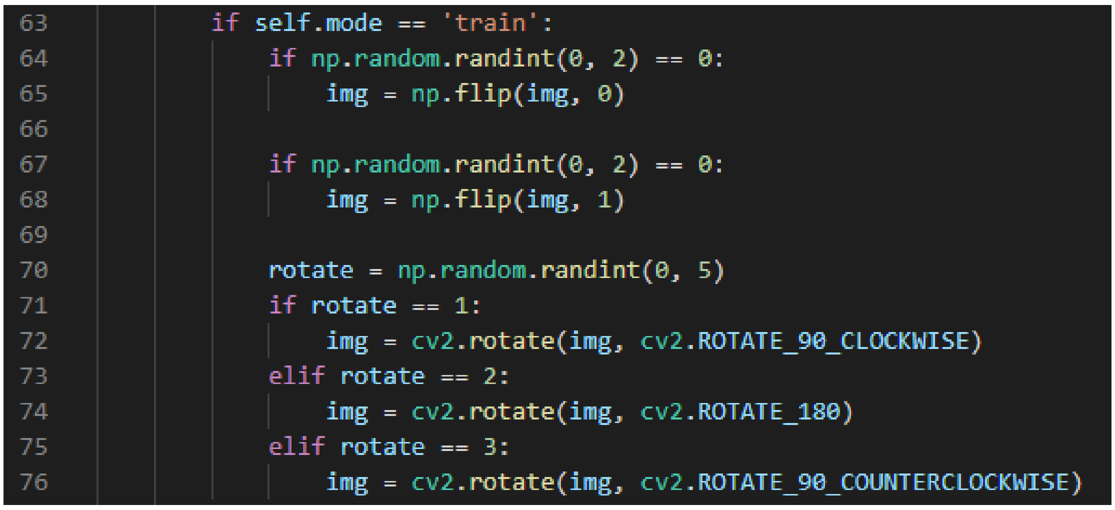
In the forward function of resnet50, just call each layer in order.

B. The details of your Dataloader



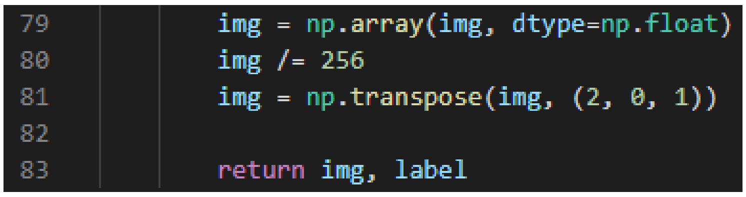
```
        label = self.label[index]
        path = self.root + self.img_name[index] + '.jpeg'

        img = cv2.imread(path)
```

In __getitem__, I first get the label from the label list. Then get the path string by the hint and use the imread function of OpenCV to read image.



```
        if self.mode == 'train':
            if np.random.randint(0, 2) == 0:
                img = np.flip(img, 0)

            if np.random.randint(0, 2) == 0:
                img = np.flip(img, 1)

            rotate = np.random.randint(0, 5)
            if rotate == 1:
                img = cv2.rotate(img, cv2.ROTATE_90_CLOCKWISE)
            elif rotate == 2:
                img = cv2.rotate(img, cv2.ROTATE_180)
            elif rotate == 3:
                img = cv2.rotate(img, cv2.ROTATE_90_COUNTERCLOCKWISE)
```

Then, if the mode is train, I would do some data augmentation. With 50% probity to flip up-down and left-right and rotate it to 4 direction in 25% probity of each direction.

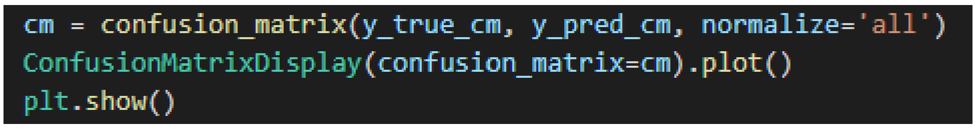


```
        img = np.array(img, dtype=np.float)
        img /= 256
        img = np.transpose(img, (2, 0, 1))

        return img, label
```

After that, I would convert it to *float* numpy array and divide by 256 to convert the pixel value to [0,1). And use numpy's transpose fumction to transpose the image shape from [H, W, C] to [C, H, W]

C. Describing your evaluation through the confusion matrix

During the test, I would save the true label and predicted label onto 2 lists.

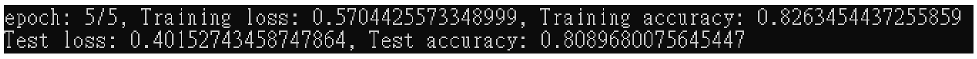


```
cm = confusion_matrix(y_true_cm, y_pred_cm, normalize='all')
ConfusionMatrixDisplay(confusion_matrix=cm).plot()
plt.show()
```

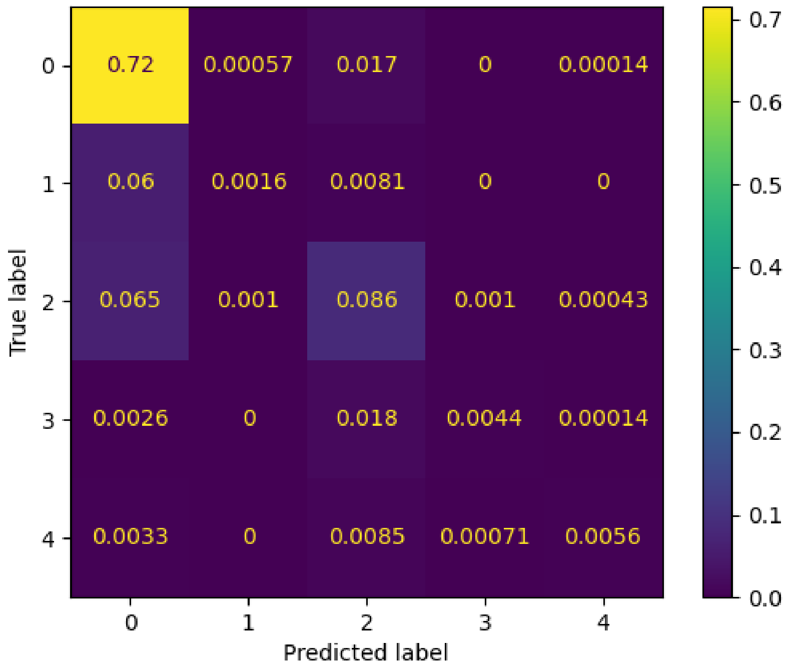
Then, use sklearn's confusion_matrix function to construct the confusion matrix and use ConfusionMatrixDisplay to draw it.

III. Experimental results

A. The highest testing accuracy

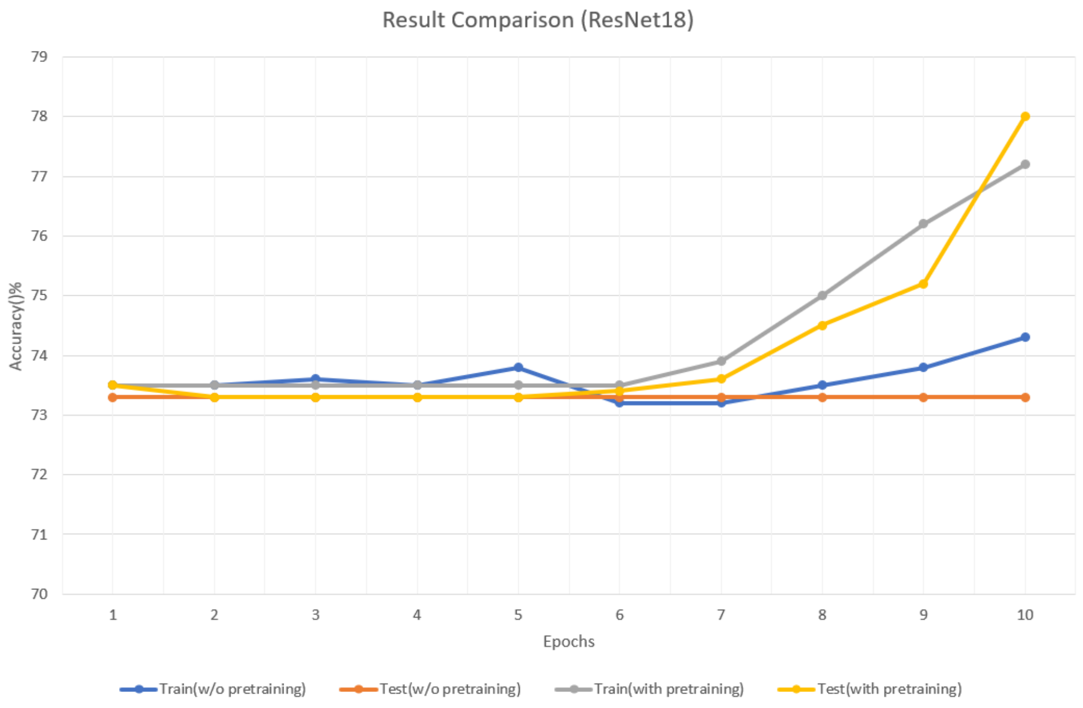


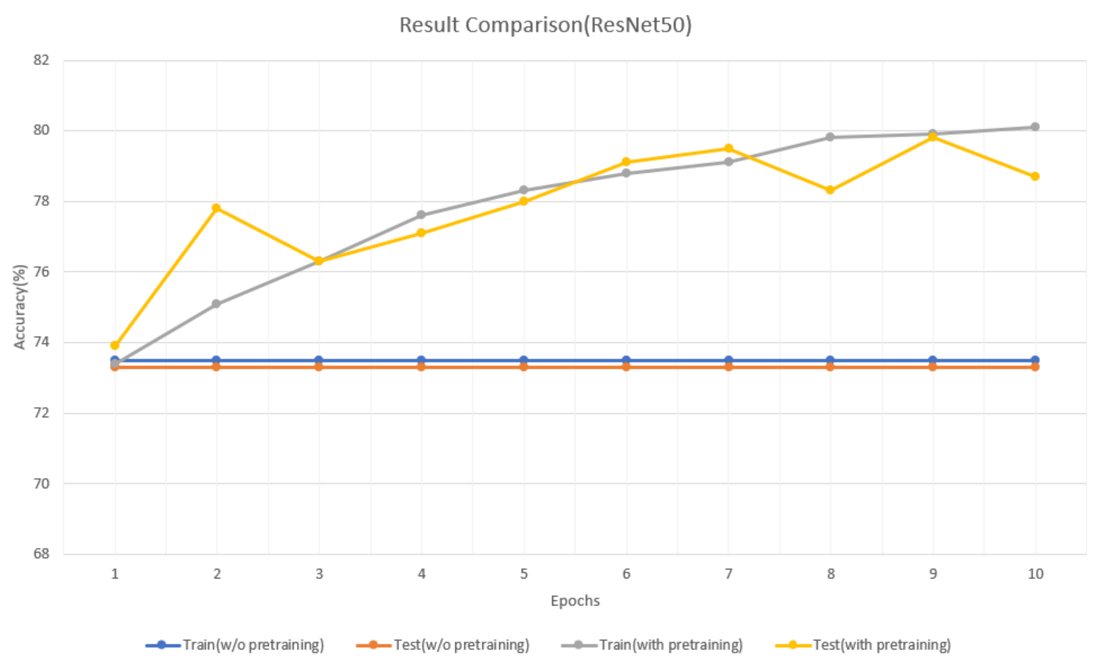
```
epoch: 5/5, Training loss: 0.5704425573348999, Training accuracy: 0.8263454437255859
Test loss: 0.40152743458747864, Test accuracy: 0.8089680075645447
```



Its accuracy is 80.89%

B. Comparison figures





IV. Discussion

From the comparison figure we can see that models with pretraining are better that models without pretraining. Models with pretraining have higher speed to converge. Although the models without pretraining's accuracy is always the same, their loss value is decreasing in fact. If I give it enough time, it may start to converge.

In the beginning of this lab, I use Adam as models' optimizer. However, I found that the models are hard to converge. They always predict class 0 as their output even after 20 epochs of training. Even I use the pretrained models or change the learning rate, it's still the same. Then, I google for it. Some article said that Adam optimizer would find a sharp minimum, but we hope to find a flat minimum. In this case, the models always predict class 0 since most of the training labels are class 0. After that, I change to use SGD with momentum. Then, this problem is solved.