Lab 4-2: Diabetic Retinopathy Detection

310551083 許嘉倫

1 Introduction

In this lab, we will need to analysis diabetic retinopathy (糖尿病所引發視網膜病變) in the following three steps. First, we need to write our own custom DataLoader through PyTorch framework. Second, we need to classify diabetic retinopathy grading via the ResNet18 and ResNet50 architecture. Finally, we have to calculate the confusion matrix to evaluate the performance.

1.1 Dataset

The dataset contains 35,124 images, we divided dataset into 28,099 training data and 7,025 testing data. The image resolution is 512x512 and has been preprocessed. The class name are as follows.

- 0 - No DR
- 1 - Mild
- 2 - Moderate
- 3 - Severe
- 4 - Proliferative DR

2 Experiment setups

2.1 The details of my model (ResNet)

ResNet solves vanishing/exploding gradient by a skip/shortcut connection. ResNet18 is composed of Basic block and ResNet50 is composed of Bottleneck block. The difference between Basic block and Bottleneck block is that there are 1x1 convolution in beginning and end of Bottleneck. This way can reduce the number of parameters. The implementation of Basic block and Bottleneck block are as follows.

```
class BasicBlock(nn.Module):
    def __init__(self, filter_nums, strides=1, expansion=False):
        super(BasicBlock, self).__init__()
        in_channels, out_channels = filter_nums
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels,out_channels, (3, 3), stride=strides,
                      padding=(1, 1), bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, (3, 3), stride=1,
                      padding=(1, 1), bias=False),
```

```
            nn.BatchNorm2d(out_channels)
        )
        self.relu = nn.ReLU(inplace=True)

        if expansion:
            self.identity = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, (1, 1),
                          stride=strides, padding=0, bias=False),
                nn.BatchNorm2d(out_channels)
            )
        else:
            self.identity = lambda x:x

    def forward(self, x):
        out = self.conv1(x)
        out = self.conv2(out)
        identity = self.identity(x)
        out = self.relu(identity + out)
        return out

class BottleneckBlock(nn.Module):
    def __init__(self, filter_nums, strides=1, expansion=False):
        super(BottleneckBlock, self).__init__()
        in_channels, mid_channels, out_channels = filter_nums
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, (1, 1), stride=strides,
                      padding=0, bias=False),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(mid_channels, mid_channels, (3, 3), stride=1,
                      padding=(1, 1), bias=False),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True)
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(mid_channels, out_channels, (1, 1), stride=1,
                      padding=0, bias=False),
            nn.BatchNorm2d(out_channels)
        )
        self.relu = nn.ReLU(inplace=True)

        if strides!=1 or expansion:
            self.identity = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, (1, 1),
                          stride=strides, bias=False),
                nn.BatchNorm2d(out_channels)
            )
```

```
        else:
            self.identity = lambda x:x

    def forward(self, x):
        out = self.conv1(x)
        out = self.conv2(out)
        out = self.conv3(out)
        identity = self.identity(x)
        out = self.relu(identity + out)
        return out
```

The implementation of ResNet18 and ResNet50 are as follows.

```
def BasicBlock_build(filter_nums, block_nums, strides=1, expansion=False):
    build_model = []
    build_model.append(BasicBlock(filter_nums, strides, expansion=expansion))
    filter_nums[0] = filter_nums[1]
    for _ in range(block_nums - 1):
        build_model.append(BasicBlock(filter_nums, strides=1))
    return nn.Sequential(*build_model)

class ResNet18(nn.Module):
    def __init__(self):
        super(ResNet18, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=(7, 7),
                      stride=(2, 2), padding=(3, 3), bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.conv2 = nn.Sequential(
            nn.MaxPool2d(kernel_size=(3, 3), stride=(2, 2), padding=1),
            BasicBlock_build(filter_nums=[64, 64], block_nums=2, strides=1)
        )
        self.conv3 = BasicBlock_build(filter_nums=[64, 128], block_nums=2,
                                      strides=2, expansion=True)
        self.conv4 = BasicBlock_build(filter_nums=[128, 256], block_nums=2,
                                      strides=2, expansion=True)
        self.conv5 = BasicBlock_build(filter_nums=[256, 512], block_nums=2,
                                      strides=2, expansion=True)
        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc = nn.Linear(512, 5)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.conv5(x)
```

```
        x = self.avgpool(x)
        x = x.view(x.shape[0], -1)
        out = self.fc(x)
        return out

def Bottleneck_build(filter_nums, block_nums, strides=1, expansion=False):
    build_model = []
    build_model.append(BottleneckBlock(filter_nums, strides, expansion=expansion))
    filter_nums[0] = filter_nums[2]
    for _ in range(block_nums - 1):
        build_model.append(BottleneckBlock(filter_nums, strides=1))
    return nn.Sequential(*build_model)


class ResNet50(nn.Module):
    def __init__(self):
        super(ResNet50, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=(7, 7),
                      stride=(2, 2), padding=(3, 3), bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.conv2 = nn.Sequential(
            nn.MaxPool2d(kernel_size=(3, 3), stride=(2, 2), padding=1),
            Bottleneck_build(filter_nums=[64, 64, 256], block_nums=3,
                             strides=1, expansion=True)
        )
        self.conv3 = Bottleneck_build(filter_nums=[256, 128, 512],
                                      block_nums=4, strides=2, expansion=True)
        self.conv4 = Bottleneck_build(filter_nums=[512, 256, 1024],
                                      block_nums=6, strides=2, expansion=True)
        self.conv5 = Bottleneck_build(filter_nums=[1024, 512, 2048],
                                      block_nums=3, strides=2, expansion=True)
        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc = nn.Linear(2048, 5)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.conv5(x)
        x = self.avgpool(x)
        x = x.view(x.shape[0], -1)
        out = self.fc(x)
        return out
```

2.2 The details of my Dataloader

My Dataloader inherits torch.utils.data.Dataset, so we need to implement __init__, __len__, __getitem__. I also use RandomRotation, RandomHorizontalFlip and normalize as my data augmentation. It is worth noting that I only use normalization as my data augmentation in testing mode. The implementation are as follows.

```
def custom_transform(mode):
    normalize = transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )
    # Transformer
    if mode == 'train':
        transformer = transforms.Compose([
            transforms.RandomRotation(degrees=20),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            normalize
        ])
    else:
        transformer = transforms.Compose([
            transforms.ToTensor(),
            normalize
        ])

    return transformer


class RetinopathyLoader(data.Dataset):
    def __init__(self, root, mode):
        """
        Args:
            root (string): Root path of the dataset.
            mode : Indicate procedure status(training or testing)
            transform: data augmentation

            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
        self.transform = custom_transform(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):
        """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
        """
        path = os.path.join(self.root, self.img_name[index] + '.jpeg')
        img = self.transform(Image.open(path).convert('RGB'))
        label = self.label[index]

        return img, label
```

2.3 Describing my evaluation through the confusion matrix

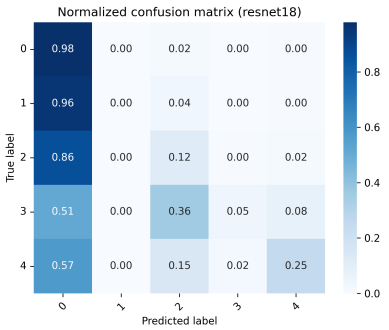
In figure 1, we can see predicted label 0 is the majority. That is because label 0 accounts for 73.5% (20,656 images) in total training dataset (28,099 images). If we predict all images as label 0, we can get at least 73.5% accuracy. This phenomenon occurs especially in the w/o pretrained model.

3 Experimental results

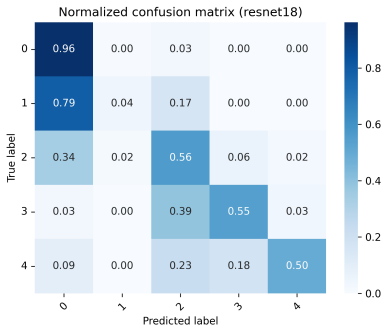
3.1 The highest testing accuracy

Both ResNet18 and ResNet50 use same hyper parameter. My hyper parameter is shown below. The following table shows the accuracy of ResNet18 and ResNet50. Figure 2 is the screenshot of highest accuracy.

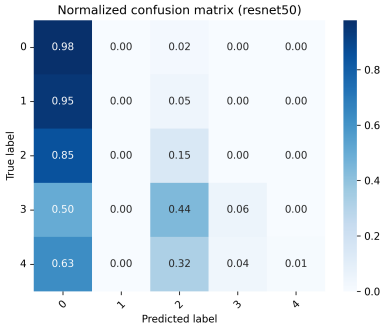
- Batch size: 16
- Learning rate: 1e-3
- Epochs: 20
- Optimizer: SGD



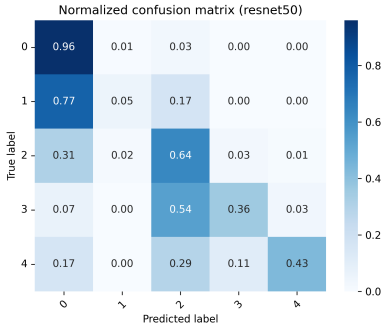
(a) ResNet18 w/o pretrained



(b) ResNet18 w/ pretrained



(c) ResNet50 w/o pretrained



(d) ResNet50 w/ pretrained

Figure 1: Confusion Matrix

- Loss function: cross entropy

	w/o pretrained	w/ pretrained
ResNet18	74.2%	81.8%
ResNet50	74.2%	**82.3%**

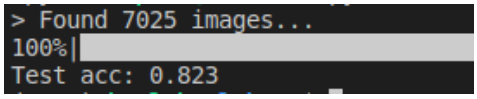


Figure 2: Screenshot of highest accuracy

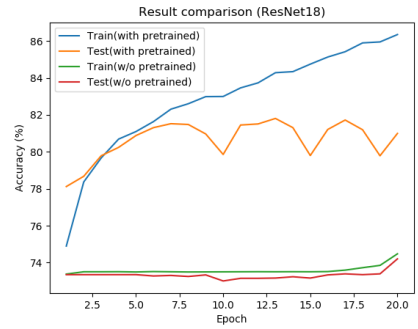
3.2 Comparison figures

Figure 3 is comparison of ResNet18 and ResNet50.

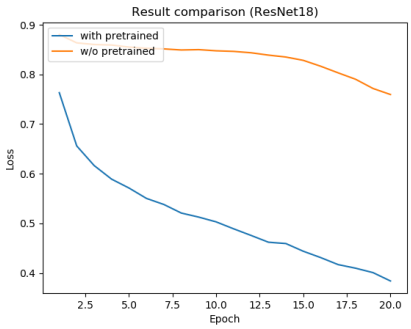
4 Discussion

4.1 The difference between original ResNet and ResNet V1.5

When I implement Bottleneck block, I find that ResNet50 provided by pytorch is different from original implementation. Bottleneck in torchvision places the stride for downsampling at 3x3 convolution, while original implementation places the stride at the first 1x1 convolution according to "Deep residual learning for image recognition". This variant is also known as ResNet V1.5 and improves accuracy according to NVIDIA implementation. This difference makes ResNet50 v1.5

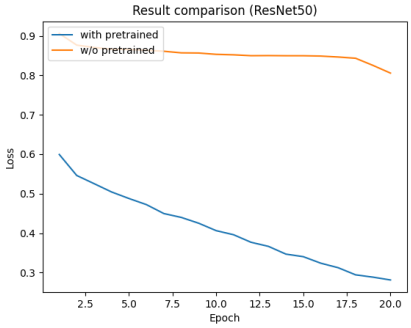


(a) ResNet18 accuracy



(b) ResNet18 loss

(c) ResNet50 accuracy



(d) ResNet50 loss

Figure 3: Comparison

slightly more accurate (0.5% top1) than v1, but comes with a small performance drawback (5% imgs/sec).

4.2 The difference between w/ pretrained and w/o pretrained

As figure3 shows, w/ pretrained model get better results than w/o pretrained model. Also we can see that imbalance dataset makes w/o pretrained models learn more difficultly. The number of class in training dataset are as follows.

- 0 - No DR: 20,656 (imgs)
- 1 - Mild: 1955 (imgs)
- 2 - Moderate: 4210 (imgs)
- 3 - Severe: 698 (imgs)
- 4 - Proliferative DR: 581 (imgs)