Lab4: Diabetic Retinopathy Detection

Hsiang-Chun Yang
Institute of Multimedia Engineering
National Yang Ming Chiao Tung University
yanghc.cs09g@nctu.edu.tw

1 Introduction

There are three major objectives in this lab. First, we need to implement a custom data loader. Second, we have to implement ResNet18 and ResNet50, and compare the classification accuracy on diabetic retinopathy detection task between pretrained model and without-pretrained model. Third, after finishing the training step, we are asked to calculate the confusion matrix of the model on testing dataset.

1.1 Dataset

The diabetic retinopathy dataset consists of 35126 colored images, 28100 for training and 7026 for testing. The image resolution is 512x512. There are 5 classes in the dataset. Each image is classified into only one class. The details regarding each class are as follows.

- 0 - No diabetic retinopathy
- 1 - Mild diabetic retinopathy
- 2 - Moderate diabetic retinopathy
- 3 - Severe diabetic retinopathy
- 4 - Proliferative diabetic retinopathy

2 Experiment setup

2.1 Dataloader

We define a custom data loader by inheriting `torch.utils.data.Dataset`, and overwrite `__init__`, `__getitem__`, and `__len__` functions. We apply data augmentation to the training data with `RandomRotation`, `RandomHorizontalFlip`, and `ColorJitter` provided by `torchvision`. The implementation of our custom data loader is as follows.

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

```
class RetinopathyDataset(Dataset):
    def __init__(self, root, mode):
        self.root = root
        self.img_name, self.label = get_data(mode)
        self.mode = mode
        transform = [
            transforms.RandomRotation(degrees=20),
            transforms.RandomHorizontalFlip(p=0.5),
            transforms.ColorJitter(0.1, 0.1, 0.1, 0.1)
        ]
        self.transform = transforms.RandomOrder(transform)
        self.to_tensor = transforms.ToTensor()

    def __len__(self):
        return len(self.img_name)

    def __getitem__(self, index):
        img_path = os.path.join(self.root, f'{self.img_name[index]}.jpeg')
        img = Image.open(img_path)
        label = self.label[index]
        if self.mode == 'train':
            img = self.transform(img)
        img = self.to_tensor(img)
        return img, label
```

2.2 ResNet

ResNet utilizes residual block to solve the problem of vanishing/exploding gradients. There are two types of residual block, `BasicBlock` and `Bottleneck`. The former one is for ResNet18 while the latter one is for ResNet50. Both two blocks share the same basic concept, which is skip connection. The major difference between them is the number of convolution layers inside. The implementation of `BasicBlock` and `Bottleneck` are as follows.

```
def downsample(in_ch, out_ch, stride):
    return nn.Sequential(
        nn.Conv2d(
            in_ch, out_ch,
            kernel_size=(1, 1),
            stride=stride,
            bias=False),
        nn.BatchNorm2d(out_ch))

class BasicBlock(nn.Module):
    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,
                            (3, 3), (1, 1), (1, 1), bias=False)
            self.downsample = None
```

```
        else:
            self.conv1 = nn.Conv2d(
                            in_ch, out_ch,
                            (3, 3), (2, 2), (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,
                        (3, 3), (1, 1), (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

class Bottleneck(nn.Module):
    def __init__(self, in_ch, mid_ch, out_ch, downsample_stride):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(
                        in_ch, mid_ch,
                        (1, 1), (1, 1), bias=False)
        self.bn1 = nn.BatchNorm2d(mid_ch)
        if downsample_stride is None:
            self.conv2 = nn.Conv2d(
                            mid_ch, mid_ch,
                            (3, 3), (1, 1), (1, 1), bias=False)
            self.downsample = None
        else:
            self.conv2 = nn.Conv2d(
                            mid_ch, mid_ch,
                            (3, 3), downsample_stride, (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,
                        (1, 1), (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 implementation of ResNet18 and ResNet50 are as follows.

```
class ResNet18(nn.Module):
    def __init__(self):
        super(ResNet18, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, (7, 7), (2, 2), (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))
        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.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


class ResNet50(nn.Module):
    def __init__(self):
        super(ResNet50, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, (7, 7), (2, 2), (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
```

2.3 Confusion matrix

We use `confusion_matrix` provided by `scikit-learn` to compute the confusion matrix. The implementation of plotting the confusion matrix is as follows.

```
def plot_confusion_matrix(y_true, y_pred, labels, fn):
    cm = confusion_matrix(y_true, y_pred, 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.tight_layout()
    plt.title('Normalized comfusion matrix')
    plt.savefig(fn, dpi=300)
```

2.4 Experiment Setup

The hyper-parameters for experiments are as follows.

- optimizer: SGD
- loss function : cross entropy loss
- number of training epochs: 10
- batch size: 4, 8, 16
- learning rate: 1e-3
- activation function: ReLU
- weight decay: 5e-4

We will train the model in two different ways. The first is training from scratch and the second is fine-tuning with pretrained weight. Therefore, there will be 6 trials for both ResNet18 and ResNet50.

3 Experimental results

3.1 The highest testing accuracy

	w/o pretrained	w/ pretrained
ResNet18	74.85%	82.41%
ResNet50	73.35%	82.49%

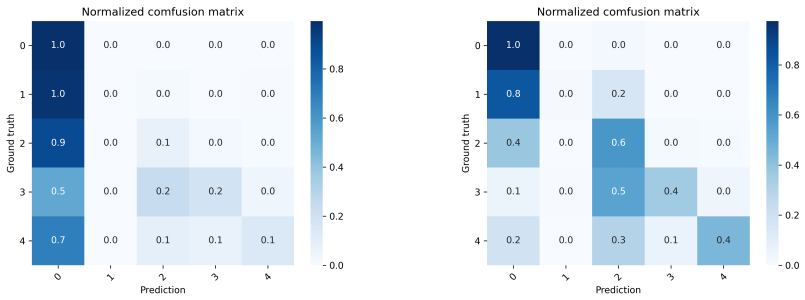
The hyper-parameters of each of the highest accuracy model are as follows.

	w/o pretrained	w/ pretrained
ResNet18	batch size: 8	batch size: 16
ResNet50	batch size: 8	batch size: 8

3.2 Comparison figures

3.2.1 ResNet18

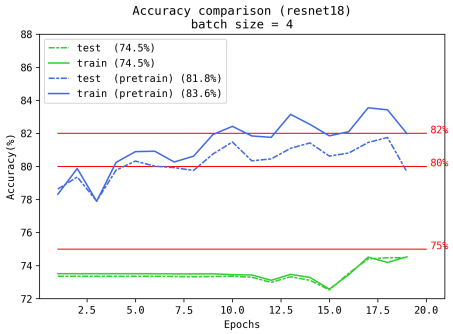
Figure 3 is the confusion matrices of ResNet18.



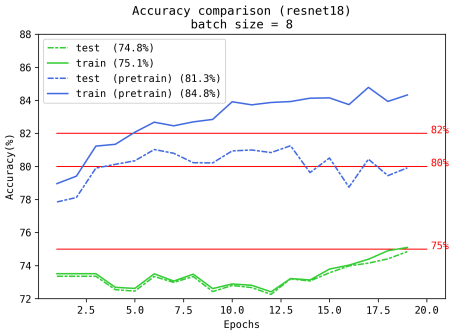
(a) Without pretrained (b) With pretrained

Figure 1: Confusion matrix of ResNet18

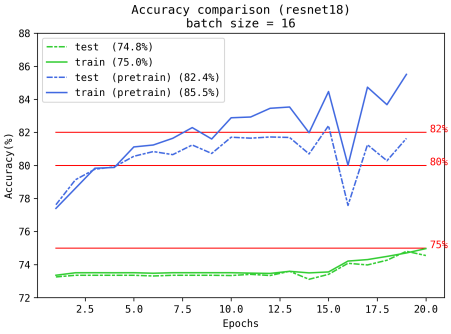
Figure 2 is the accuracy comparison between with and without pretrained with differents batch sizes.



(a) Batch size = 4



(b) Batch size = 8

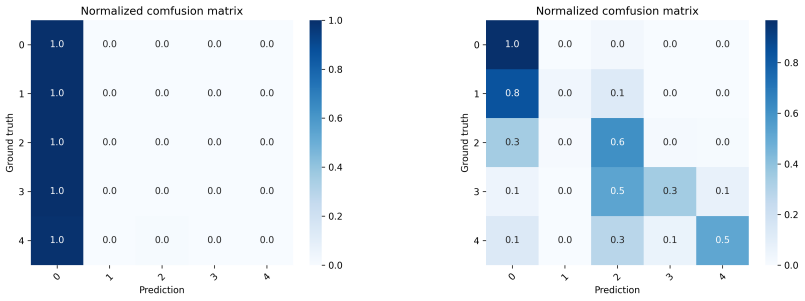


(c) Batch size = 16

Figure 2: Accuracy of ResNet18

3.2.2 ResNet50

Figure 3 is the confusion matrices of ResNet50.

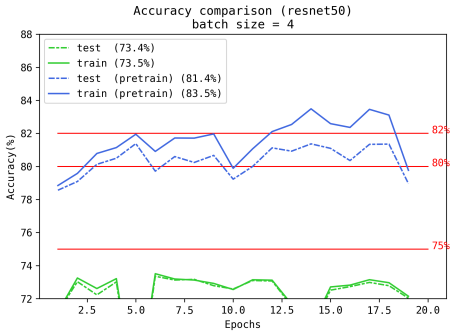


(a) Without pretrained

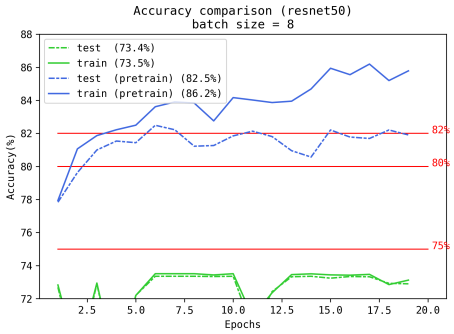
(b) With pretrained

Figure 3: Confusion matrix of ResNet50

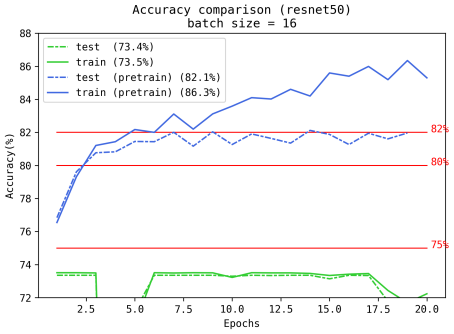
Figure 4 is the accuracy comparison between with and without pretrained with differents batch sizes.



(a) Batch size = 4



(b) Batch size = 8



(c) Batch size = 16

Figure 4: Accuracy of ResNet18

4 Discussion

4.1 ResNet18 v.s. ResNet50

There are no big differences in performance between ResNet18 and ResNet50. Both architectures tend to classify all cases into "no diabetic retinopathy" when the models are trained from scratch, while tend to misclassify class-1 into class-0 when the models are fine-tuned from pretrained.

4.2 Train from scratch v.s. fine-tune with pretrained

There is a large gap in accuracy between training from scratch and fine-tuning from pretrained. Though I think it is unreasonable since the features captured from the model pretrained on 1000-class ImageNet dataset might not be suitable for this specific dataset, the fine-tuned model outperforms the trained from scratch one. It seems that by training the model with a large number of images, the model did learn some general low-level features. Through the fine-tuning step, the model might utilize these low-level features to construct high-level features which are suitable for a specific domain.