Deep Learning

Lab4: Diabetic Retinopathy Detection

Name: 許子駿
Student ID: 311551166

Institute of Computer Science and Engineering

2023 年 4 月 21 日

Table of Contents

Chapter 1

Introduction

This task is to solve diabetic retinopathy detection problem by ResNet-18 and ResNet-50 models. Additionally, compare two models, one with pretrained weight and the other without pretrained.
The dataset includes 5 classes,

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

Chapter 2

Experiment setups

Experiment setups are listed in this chapter including 3 parts, ResNet, Data preprocess and DataLoader, and confusion matrix.

2.1 ResNet

ResNet is composed of `BasicBlock`, `BottleNeck` (See Listings 2.1 and 2.2). `BasicBlock` is used by ResNet-18 and `BottleNeck` is used by ResNet-50, since the structure of each block of two models are different, but the overall architecture is the same.
`__make_layers` function in `ResNet` is used to create the blocks, and `__init_weights` is used to initialize weights of layers (See Listing 2.3). The last layer of `ResNet`, i.e. `classifier`, is different from original, since the input image size is 512×512 in this task.
In this experiment, the pretrained model is loaded from `torchvision` library, and the train-from-scratch model uses the hand-crafted model without loading weights. If we want to use ResNet-18 or ResNet-50, call functions `ResNet_18` or `ResNet_50`, respectively.

```
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self,
            in_channels: int, out_channels: int, stride: int) -> None:
        super().__init__()

        self.conv1 = ConvBlock(
            in_channels, out_channels, kernel_size=3,
            stride=stride, padding=1, bias=False)
        self.conv2 = ConvBlock(
```

```python
            out_channels, out_channels, kernel_size=3,
            stride=1, padding=1, bias=False, activation=None)
        self.relu1 = nn.ReLU(inplace=True)

        if stride >= 2 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(
                    in_channels, out_channels, kernel_size=1,
                    stride=stride, padding=0, bias=False),
                nn.BatchNorm2d(out_channels)
            )
        else:
            self.shortcut = nn.Sequential()
```

Listing 2.1: Python code of **BasicBlock** (some code is omitted).

```python
class BottleNeck(nn.Module):
    expansion = 4

    def __init__(self,
            in_channels: int, out_channels: int, stride: int) -> None:
        super().__init__()

        self.conv1 = ConvBlock(
            in_channels, out_channels, kernel_size=1,
            stride=1, padding=0, bias=False)
        self.conv2 = ConvBlock(
            out_channels, out_channels, kernel_size=3,
            stride=stride, padding=1, bias=False)
        self.conv3 = ConvBlock(
            out_channels, out_channels * 4, kernel_size=1,
            stride=1, padding=0, bias=False, activation=None)
        self.relu1 = nn.ReLU(inplace=True)

        if stride >= 2 or in_channels != out_channels * 4:
            self.shortcut = nn.Sequential(
                nn.Conv2d(
                    in_channels, out_channels * 4, kernel_size=1,
                    stride=stride, padding=0, bias=False),
                nn.BatchNorm2d(out_channels * 4)
            )
        else:
            self.shortcut = nn.Sequential()
```

Listing 2.2: Python code of **BottleNeck** (some code is omitted).

```python
class ResNet(nn.Module):
    def __init__(self,
                 block: Union[BasicBlock, BottleNeck], groups: list, num_classes: int,
                 dim_hidden=128, init_weights=True) -> None:
        super().__init__()

        self.channels = 64
        self.block = block

        self.conv1_x = nn.Sequential(
            nn.Conv2d(
                in_channels=3, out_channels=self.channels,
                kernel_size=7, stride=2, padding=3, bias=False),
            nn.BatchNorm2d(self.channels),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )
        self.conv2_x = self.__make_layers(64, groups[0], 1)
        self.conv3_x = self.__make_layers(128, groups[1], 2)
        self.conv4_x = self.__make_layers(256, groups[2], 2)
        self.conv5_x = self.__make_layers(512, groups[3], 2)

        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten(),
            nn.Linear(
                in_features=512 * self.block.expansion,
                out_features=dim_hidden),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.25),
            nn.Linear(in_features=dim_hidden, out_features=num_classes),
        )

        if init_weights:
            self.__init_weights()

def ResNet_18(num_classes=5):
    return ResNet(block=BasicBlock, groups=[2, 2, 2, 2], num_classes=num_classes)
def ResNet_50(num_classes=5):
    return ResNet(block=BottleNeck, groups=[3, 4, 6, 3], num_classes=num_classes)
```

Listing 2.3: Python code of **ResNet** (some code is omitted).

2.2 Data preprocess and DataLoader

2.2.1 Data preprocess

For the training set, I first call function `center_crop` to **centerly crop** each image to get rid off the useless part, and make the cropped image square. Then, `RandomHorizontalFlip` and `RandomVerticalFlip` with 50% probability, and `Resize` it to be 512×512. But for testing set, each image is **centerly cropped** and `Resize` only (See Listing 2.4).
This preprocess method makes the transformed images contain only the retina, and horizontal and vertical flips are data augmentation to let the model learn from different kinds of input to improve model generalization and capacity.
Normalization is not necessary in this task empirically, since the accuracy does not improve. Without normalization the computation cost is reduced a little bit.

2.2.2 DataLoader

`DataLoader` is implemented by a class with two classes as members, training dataset and testing dataset. And `get_data_loader` function returns Dataloaders of training and testing sets (See Listing 2.4).

```python
class RetinopathyDataLoader(object):
    def __init__(self, dir_dataset: str, transform: Optional[list]=[
            transforms.RandomHorizontalFlip(p=0.5),
            transforms.RandomVerticalFlip(p=0.5),
            transforms.Resize((512, 512)),
            transforms.ToTensor()
        ]) -> None:

        self.transform = transform if transform else []
        self.transform = transforms.Compose(self.transform)

        self.training_set = self.RetinopathyTrainingDataset(
            dir_dataset, self.transform)
        self.testing_set = self.RetinopathyTestingDataset(
            dir_dataset, transforms.Compose([
              transforms.Resize((512, 512)), transforms.ToTensor()]))

    def get_data_loader(self,
```

```python
            batch_size: int, num_workers: int, shuffle=True) -> list[DataLoader, DataLoader]:
        return [
            DataLoader(self.training_set, batch_size=batch_size,
                shuffle=shuffle, num_workers=num_workers),
            DataLoader(self.testing_set, batch_size=batch_size,
                shuffle=shuffle, num_workers=num_workers)]

def center_crop(self, img: PIL.Image):
    width, height = img.size
    cropped = img.crop(((width - height) / 2, 0, (width + height) / 2, height))

    return cropped
```

Listing 2.4: Python code of **DataLoader** (some code is omitted).

2.3 Confusion matrix

In this experiment, confusion matrix is implemented by storing true labels and predicted labels of **epoch with best testing-accuracy**. Then, call the function `confusion_matrix` to create the confusion matrix. Finally, display the matrix by function `ConfusionMatrixDisplay`. Both functions are from `sklearn.metrics` (See Listing 2.5).

```python
def confusion_matrix(self,
        true_label: np.ndarray, pred_label: np.ndarray, path: str, model_name: str) -> None:
    plt.figure(figsize=(15, 15))

    plt.title(f'Confusion matrix of {model_name}')
    plt.xlabel('Predicted label', fontsize='18')
    plt.ylabel('True label', fontsize='18')

    cm = confusion_matrix(y_true=true_label, y_pred=pred_label, normalize='true')
    ConfusionMatrixDisplay(
        confusion_matrix=cm, display_labels=[0, 1, 2, 3, 4]).plot(cmap=plt.cm.Blues)

    plt.savefig(path, dpi=400, bbox_inches='tight', pad_inches=0.1)
    plt.close()

    print(f'[INFO]: Finish saving confusion matrix to {path}...')
```

Listing 2.5: Python code of **Confusion matrix** (some code is omitted).

Chapter 3

Experimental results

Experiments of this chapter are done with setup below:

- Seed: 42.
- Batch size: 64 (ResNet-18)/16 (ResNet-50).
- Number of epochs: 20 (ResNet-18)/10 (ResNet-50).
- Learning rate: 10^{-3}.
- Optimizer: `SGD` with momentum 0.9 and weight decay 5×10^{-4}.

3.1 Accuracy and loss

From Figures 3.1 and 3.2, we can see that the models without pretrained does not work, since the accuracy does not improve through training. Loss is the same (See Figures 3.3 and 3.4).

The best accuracy of ResNet-18 is about **87%** for training and **84%** for testing, and ResNet-50 is about **86%** for training and **85%** for testing.

ResNet-50 is a little bit better for **testing-accuracy** of this task.

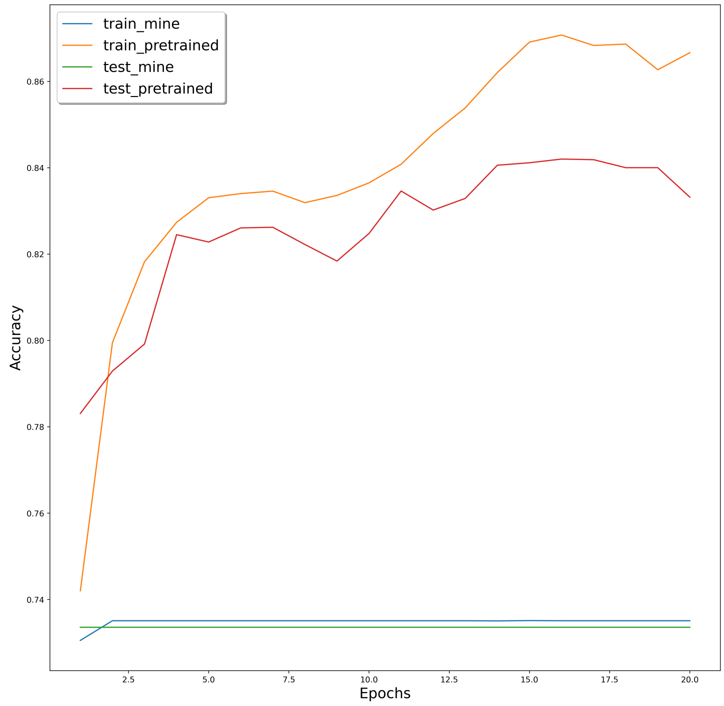

Figure 3.1: Accuracy of ResNet-18 (with and without pretrained).

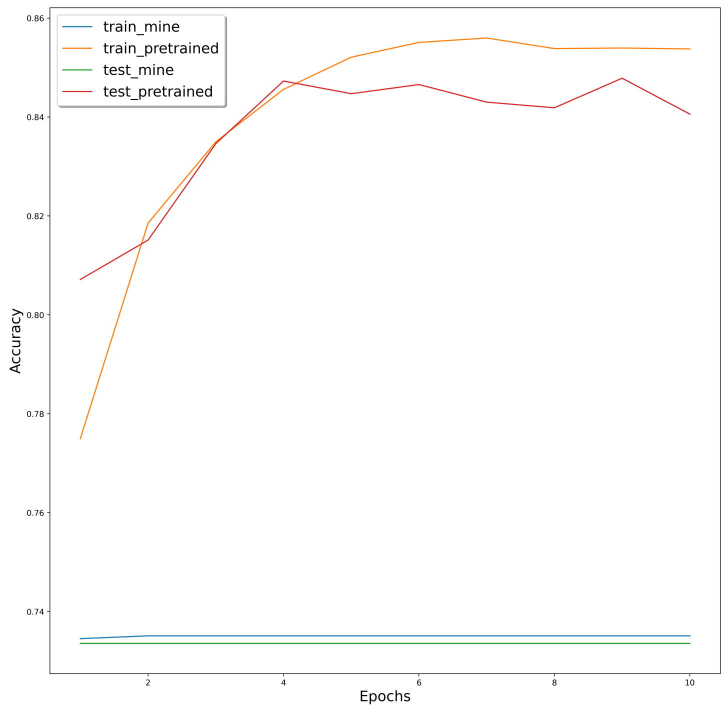

Figure 3.2: Accuracy of ResNet-50 (with and without pretrained).

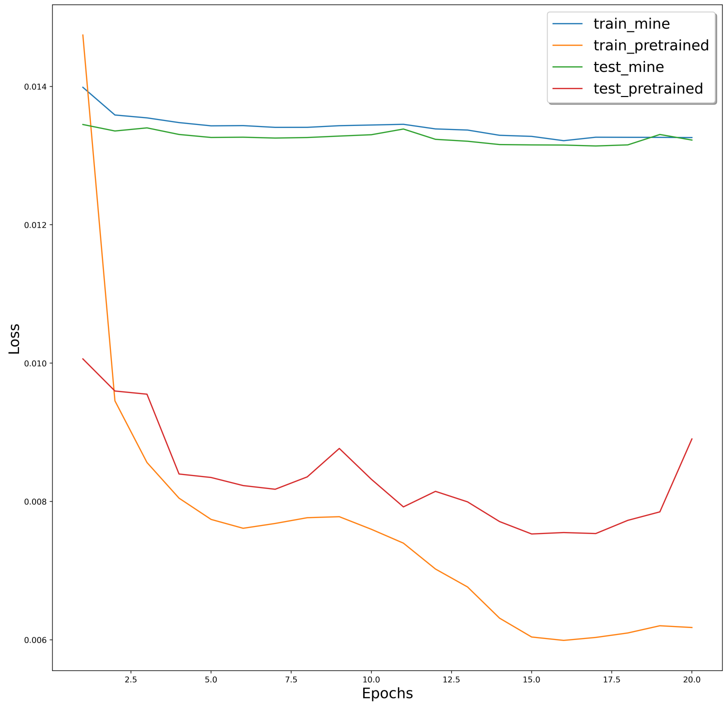

Figure 3.3: Loss of ResNet-18 (with and without pretrained).

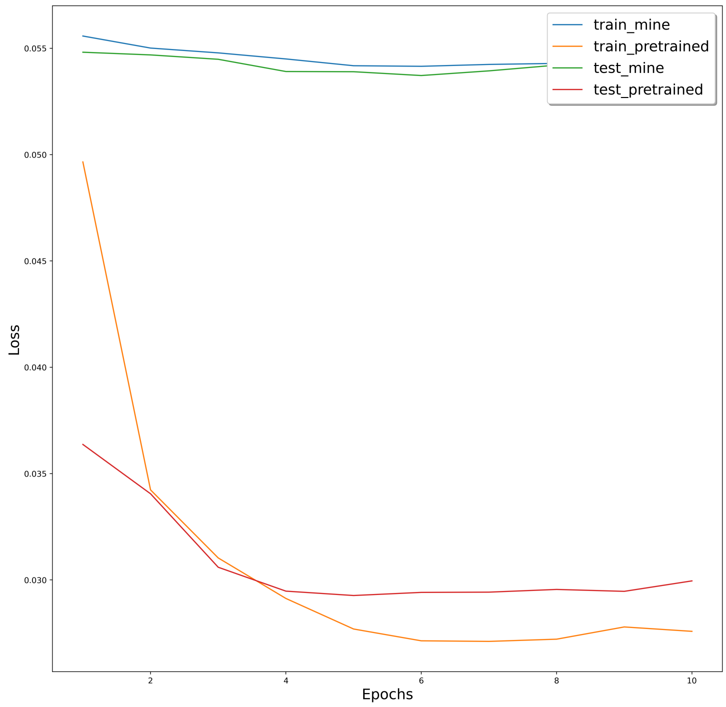

Figure 3.4: Loss of ResNet-50 (with and without pretrained).

3.2 Confusion matrix

From Figures 3.5 and 3.7, we can see that the model without pretrained does not work, it's same as previous section, the accuracy does not improve, the loss does not decrease, and the confusion matrices are obviously bad.
And the confusion matrices of pretrained models are much better (See Figures 3.6 and 3.8), but also not good enough, since the percentage of correct labels is still not dominant.

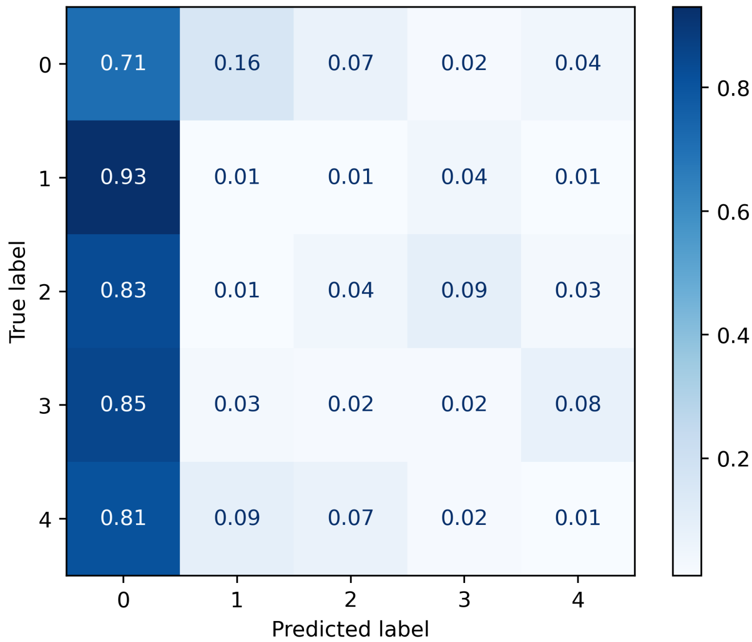

Figure 3.5: Confusion matrix of ResNet-18 (without pretrained).

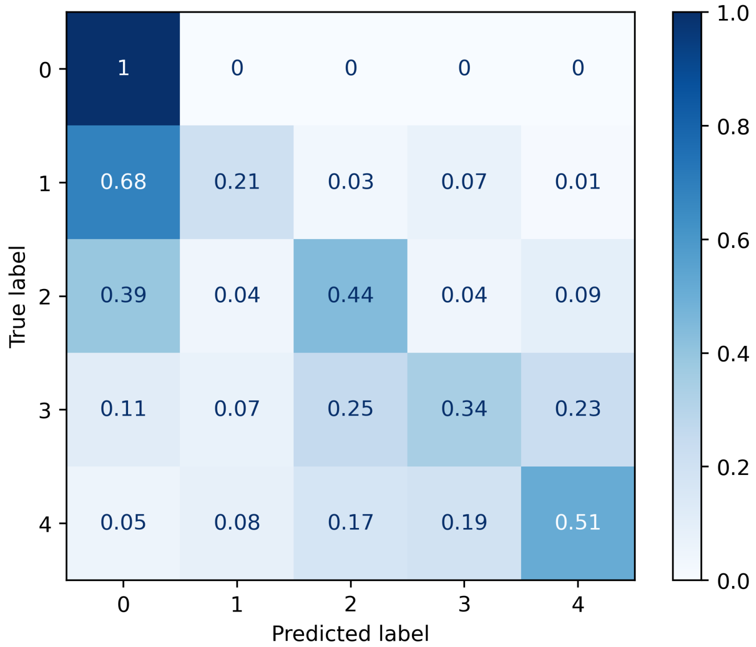

Figure 3.6: Confusion matrix of ResNet-18 (pretrained).

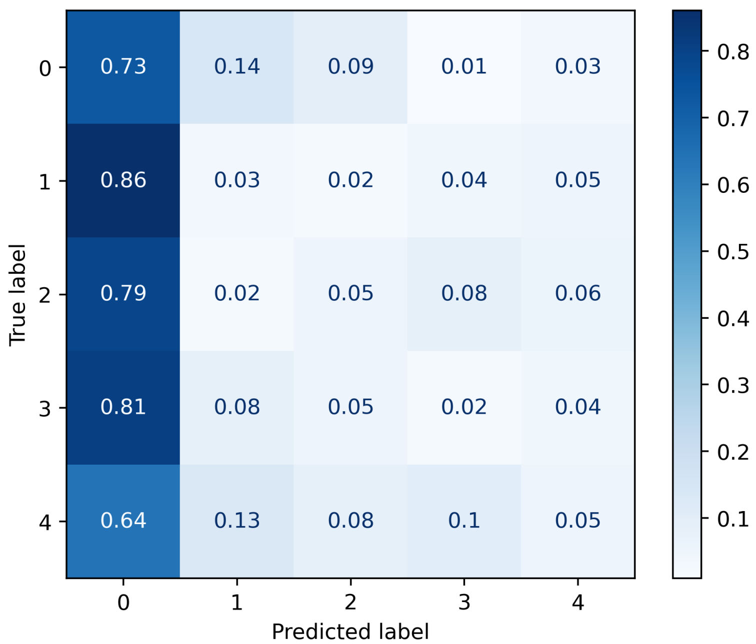

Figure 3.7: Confusion matrix of ResNet-50 (without pretrained).

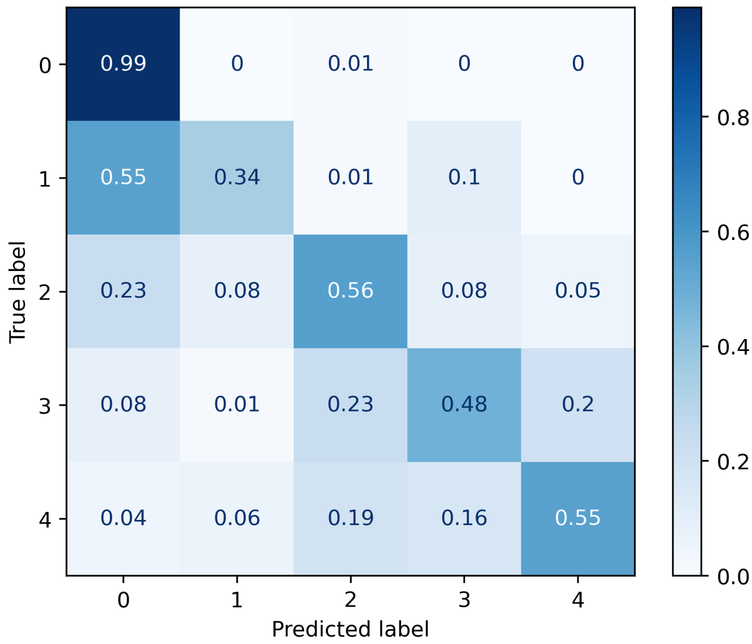

Figure 3.8: Confusion matrix of ResNet-50 (pretrained).

Chapter 4

Conclusion

After training ResNets, I found that the model without pretrained does not work, even if the accuracy is about **73%**. With pretrained weights, the model has learned some visual patterns, which is maybe from another dataset, and the knowledge can be used to solve this task. Thus, pretrained models are really useful.