Deep Learning and Practice Report
Lab4-2: Diabetic Retinopathy Detection

1 Introduction

Diabetic retinopathy is a complication of diabetes. The back of the eye (retina) could be damaged due to the high blood sugar levels. It could even lead to blindness if left untreated. Therefore, detecting this disease at early stage becomes an important task. In this lab, we are asked to utilize ResNet for diabetic retinopathy detection. Also, we need to write our own customized DataLoader through PyTorch framework, and we have to calculate the confusion matrix to evaluate the performance.

2 Experiment Setups

2.1 The details of your model

ResNet (Residual Network) is a deep convolutional neural network proposed in 2015. By adding a skip connection between the input and the output after several weight layers, training deeper neural networks has become more practical. This approach not only makes the network easier to optimize, but also alleviates vanishing / exploding gradient problem. The team that proposed ResNet has won the 1st place on several competitions, including the ILSVRC 2015 classification task, ImageNet detection, ImageNet localization, COCO detection and COCO Segmentation.

```python
class resnet(nn.Module):
    def __init__(self, model, pretrained, num_class):
        super(resnet, self).__init__()

        self.model_name = model
        self.num_class = num_class

        if self.model_name == "resnet18":
            self.resnet = models.resnet18(pretrained=pretrained)
        elif self.model_name == "resnet50":
            self.resnet = models.resnet50(pretrained=pretrained)

        ## Reinitialize the last layer
        fc_in_dim  = self.resnet.fc.in_features
        fc_out_dim = self.num_class
        self.resnet.fc = nn.Linear(fc_in_dim, fc_out_dim)

    def forward(self, x):
        x = self.resnet(x)
        return x
```

Figure 1: Code of my ResNet model.

In this lab, we need to apply both ResNet18 and ResNet50 to classify diabetic retinopathy grading. Figure 1 shows the code of my ResNet model. I initialize the model architecture from `torchvision`, and a boolean variable `pretrained` is used to decide whether to load the pre-trained model weights or not. Besides, the last layer of ResNet is a fully-connected layer with more than 1000 output features, while there are only 5 labels in our dataset. Therefore, I reinitialize the last layer with 5 output features in order to adjust the model for our task.

2.2 The details of your dataloader

```python
def __getitem__(self, index):
    img_path = "{}/data/{}.jpeg".format(self.root, self.img_name[index])
    label_gt = self.label[index]

    ## Define transformations
    transformations = transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomVerticalFlip(),
        transforms.ToTensor(), ## Scales the pixel to the range [0, 1]
        transforms.Normalize((0.3749, 0.2602, 0.1857),(0.2526, 0.1780, 0.1291)) ## Normalization
    ])

    img = Image.open(img_path)
    img = transformations(img)

    return img, label_gt
```

Figure 2: Code of the function `__getitem__` in my dataloader.

Figure 2 shows the code of the function `__getitem__` in my customized dataloader. The function will be automatically called when iterating through all batches of data. In this function, I first load and convert each image into numpy array with the python package **Pillow**. Several transformations such as random flipping and normalization are then performed on each image. I also convert each numpy array into pytorch tensor so that it can be put into GPU and accelerate the training process. Finally, the function returns the processed image and the corresponding label.

2.3 Describe your evaluation through the confusion matrix

```python
def plot_confusion_matrix(args, labels, preds):
    """
    Plot confusion matrix (Need to implement by myself!).
    Used when test only.
    """
    print("Plotting confusion matrix...")

    ## Calculate confusion matrix
    num_class = len(set(labels.tolist()))
    confusion_matrix = np.zeros((num_class, num_class))
    for idx in range(len(labels)):
        confusion_matrix[labels[idx]][preds[idx]] += 1

    ## Normalize
    confusion_matrix = confusion_matrix / confusion_matrix.sum(axis=1)[np.newaxis].T

    ## Plot
    textcolors = ("black", "white")
    fig, ax = plt.subplots()
    img = ax.imshow(confusion_matrix, cmap=plt.cm.Blues)
    for i in range(confusion_matrix.shape[0]):
        for j in range(confusion_matrix.shape[1]):
            ax.text(
                j, i, "{:.2f}".format(confusion_matrix[i, j]),
                ha="center", va="center",
                color=textcolors[confusion_matrix[i, j] > 0.5]
            )

    plt.colorbar(img)
    plt.title("Normalized Confusion Matrix")
    plt.xlabel("Predicted Label")
    plt.ylabel("True Label")

    pretrained_str = "w" if args.pretrained else "wo"

    plt.savefig("./cm_{}_{}_pretrained.png".format(args.model, pretrained_str))
```

Figure 3: Code of the function `plot_confusion_matrix`.

Learning Rate	Pretrain	ResNet18		ResNet50	
		Test Acc.	F1-Macro	Test Acc.	F1-Macro
1.0×10^{-3}	✗	73.4%	17.56%	73.21%	17.19%
1.0×10^{-3}	✓	82.16%	51.98%	**82.28%**	**53.4%**
5.0×10^{-4}	✗	75.67%	31.69%	74.31%	23.88%
5.0×10^{-4}	✓	81.27%	48.96%	80.16%	45.79%

Table 1: Testing results under different settings.

Figure 3 shows the code of the function `plot_confusion_matrix`. This function receives the ground-truth labels and the model's predictions as inputs. Since the dataset consists of 5 classes, a 2D array with shape $(5, 5)$ is then created to represent the confusion matrix. Specifically, for $i, j \in \{0, 1, 2, 3, 4\}$, the entry (i, j) of this matrix denotes the number of samples with ground-truth label i, but being classified as label j by the model. Each entry is obtained by looping and making comparison over all the ground-truth labels and the model's predictions.

3 Experimental Results

3.1 The highest testing accuracy

3.1.1 Screenshot

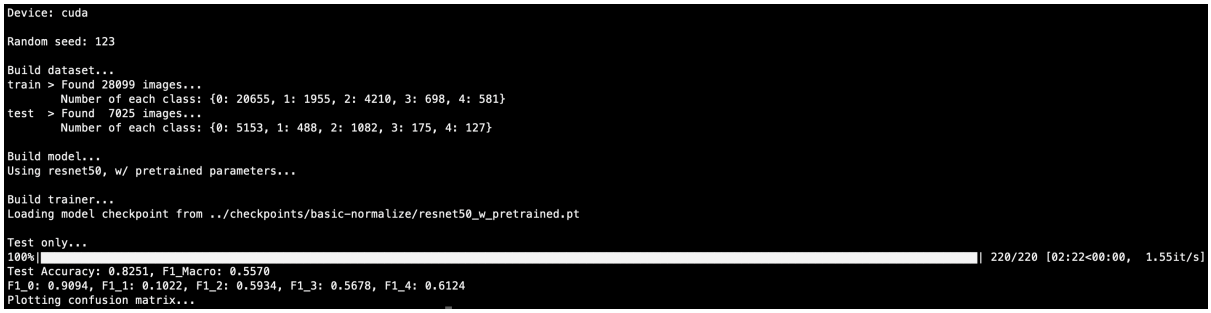

Figure 4: Screenshot of highest testing accuracy.

3.1.2 Anything you want to present

Table 1 displays the overall testing results under different experiment settings. All models are trained for 20 epochs and the one with highest testing accuracy is reported. Additionally, I report the macro-F1 score as well since this dataset exists severe class imbalance issue (this can be observed in the information displayed in figure 4). The macro-F1 score is the average F1 of each class, which can give us a brief overview on how well the model performs on each class. Among all models, ResNet50 with pre-trained model weights achieves highest testing accuracy and macro-F1. We can see that models with pre-trained weights outperforms those without pre-trained weights on both scores, especially on macro-F1.
For models trained from scratch, they tend to predict most samples into the majority class, thus having a low macro-F1. However, I find that using a smaller learning rate for them can improve both scores.

3.2 Comparison figures

3.2.1 Learning curves

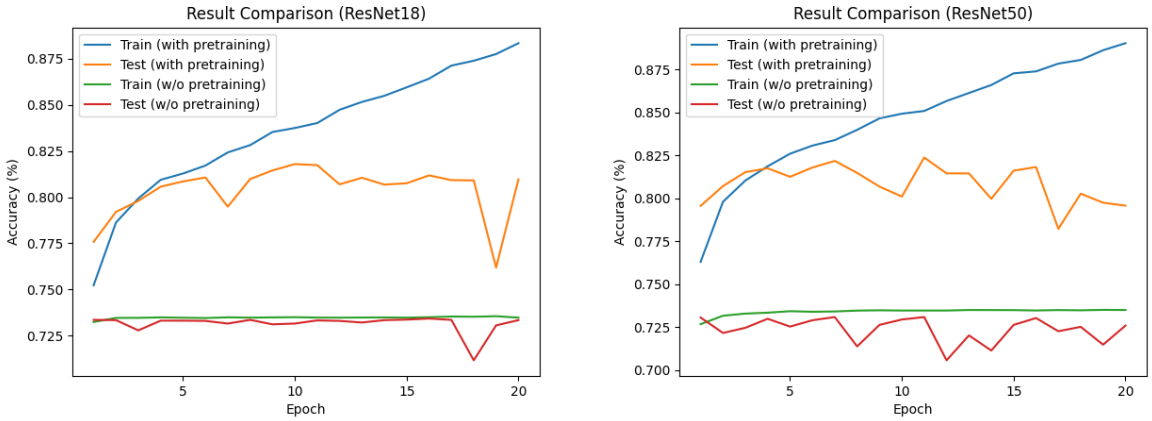

Figure 5: Learning curves for ResNet18 and ResNet50.

Figure 5 shows the learning curves of all models. As we can see, models without pre-trained weights fail to achieve higher accuracy. On the other hand, models with pre-trained weights can somehow learn to discriminate different classes of images correctly.

3.2.2 Confusion matrix

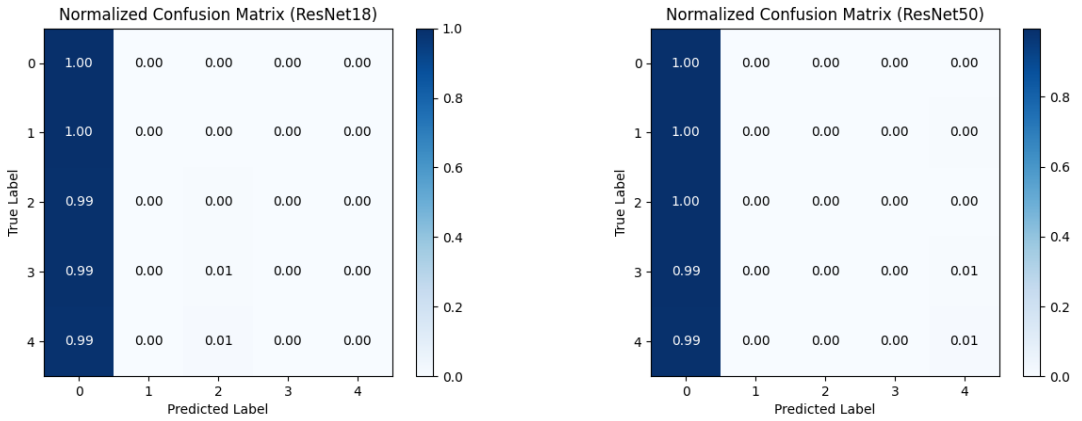

Figure 6: Confusion matrix without pre-trained model weights.

Figure 6 shows the confusion matrix of ResNet18 and ResNet50 that are trained without pre-trained model weights. It is obvious that almost all samples have been classified as class 0, which is the majority of this dataset. This indicates that the models fail to learn the features of other classes.
However, as shown in figure 7, we can see that the phenomenon described above is significantly mitigated by training the models with pre-trained weights. Although the models still have poor performance on class 1, the predictions are much more balanced now. From these results, we can conclude that using large-scale pre-trained models indeed strengthen the models' ability on fitting the imbalanced data.

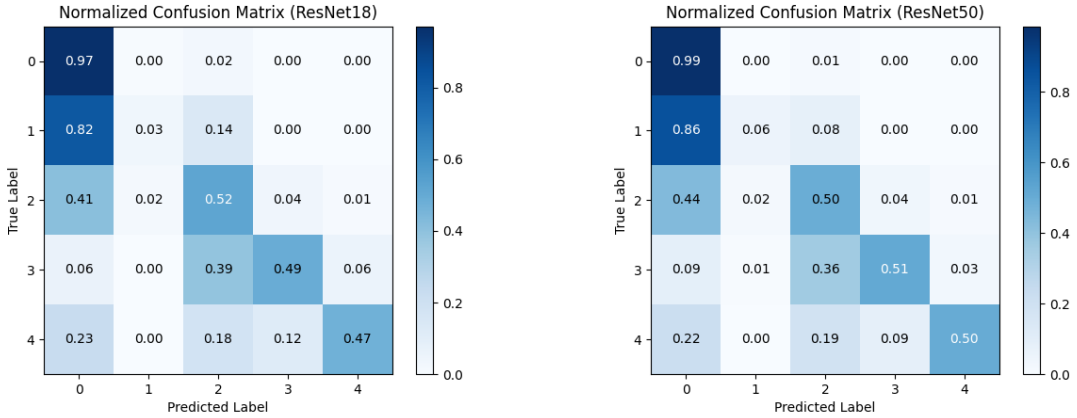

Figure 7: Confusion matrix with pre-trained model weights.

Learning Rate	Normalize	Weighted-Loss	ResNet18		ResNet50	
			Test Acc.	F1-Macro	Test Acc.	F1-Macro
1.0×10^{-3}	✗	✗	82.16%	51.98%	82.28%	53.4%
1.0×10^{-3}	✓	✗	81.79%	53.87%	**82.38%**	**54.63%**
1.0×10^{-3}	✗	✓	79.36%	51.46%	77.3%	53.15%

Table 2: Testing results with pre-trained model weights under different settings.

4 Discussion

4.1 Anything you want to share

In this section, some additional experiment results are provided. Table 2 shows the results under different settings. The model is trained with pre-trained weights. First of all, I discovered that doing normalization on the images has little impact on the model's performance.

Weighted-Loss	Model	F1-0	F1-1	F1-2	F1-3	F1-4
✗	ResNet18	90.96%	2.75%	60.3%	51.15%	54.74%
	ResNet50	91.03%	13.49%	60.54%	44.79%	57.14%
✓	ResNet18	89.58%	12.15%	55.03%	45.65%	54.89%
	ResNet50	88.53%	**20.48%**	57.07%	46.95%	52.7%

Table 3: F1 score of each class with pre-trained model weights under different settings.

Also, as I mentioned in previous sections, the class distribution in this dataset is strongly imbalanced. The data is mostly dominated by class 0. Consequently, simply classifying all samples as class 0 can roughly achieve the accuracy of 73%, which matches the worst result shown in table 1. However, this is not the situation we desire since the model learns nothing. Therefore, I also trained the models with weighted-loss. I set the loss weights to $(1, 10, 5, 10, 10)$ in order to encourage the models to pay more attention on other classes. The testing accuracy and macro-F1 score are shown in table 2, and table 3 demonstrates the detailed F1 score on each class. We can see that training with weighted-loss indeed makes the model focus on minor classes more, though it may require more epochs to attain the same accuracy score as the original training process.