Deep Learning, SUMMER 2022
LAB3 Diabetic Retinopathy Detection

K.S. Chen

July 27, 2022

1 Introduction

In this lab, we learned to implement pretrained and without pretrained ResNet models to classify diabetic retinopathy grading. We should be familiar with the architecture of ResNet18 and ResNet50. Furthermore, confusion matrix is adopted to analysis the predictions.

2 Experiment set up

2.1 The detail of your model(ResNet)

ResNet is a model invented to overcome the vanishing gradient problem of traditional CNN models. Its residual blocks connect the output of the previous layer to that of the current layer and add them together before entering activation function. By doing so, the current layer will only output the residual of the weights, making it easier for the residual to reach zero when the current layer fail to achieve lower loss. In other words, the weights will remain the same even if the network is very deep and layers cannot improve the weights. As a results, the training loss will not increase as the network become deeper.

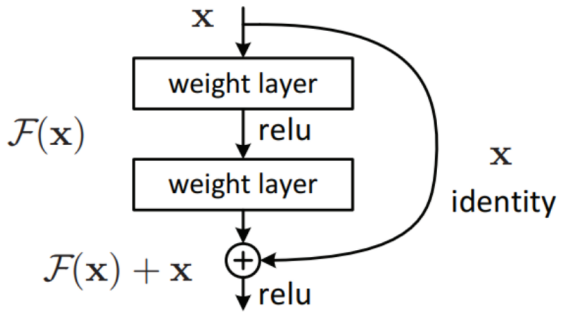


Figure 1: The basic residual block of ResNet.

2.2 The details of your Dataloader

Figure 2 shows the detail implementation of my dataloader. RandomVeticalFlip and RandomRotation is used to provide diversity to the training dataset. Since the dataset is already large, data augmentation might not be necessary. Therefore, after these two transformation, the images is converted into tensors by the ToTensor() function. Note that we do not want to further increase the difficulty of testing data, so no image transformation is applied.

2.3 Describing your evaluation through the confusion matrix

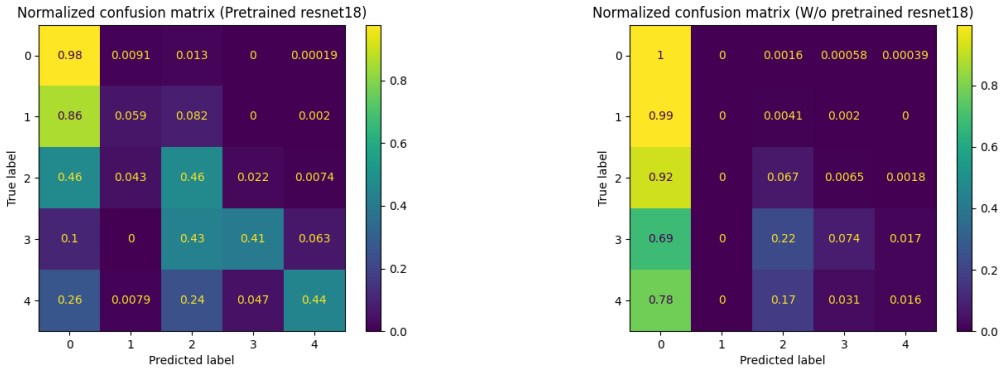
Figure 3-a, 3-b, 4-a and 4-b display the confusion matrix of four different models we evaluate in this lab. As you can see from the color distributions, pretrained models(Figure 3-a and 4-a) generally

```python
class RetinopathyLoader(Dataset):
    def __init__(self, root, mode):
        self.root = root
        self.img_name, self.label = getData(mode)
        self.mode = mode

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

    def __getitem__(self, index):
        PATH = os.path.join(self.root, self.img_name[index] + ".jpeg")
        label = torch.from_numpy(np.array(self.label[index]))
        if self.mode == "train":
            transform = transforms.Compose([
                transforms.RandomVerticalFlip(p=0.5),
                transforms.RandomRotation((-30,30)),
                transforms.ToTensor()])
        else:
            transform = transforms.Compose([transforms.ToTensor()])
        img = Image.open(PATH).convert("RGB")
        img = transform(img)
        return img, label
```

Figure 2: The detail of my dataloader.



(a) Pretrained ResNet18

(b) W/o pretrained ResNet18

Figure 3: Normalized confusion matrix of ResNet18

have higher correct prediction rate (diagonal elements). For the without-pretrained models(Figure 3-b and 4-b), the models tend to predict every label as 0, the most appeared one, probably due to the imbalanced data distribution. Although the prediction of pretrained Resnet18(Figure 3-a) and ResNet50(Figure 4-a) are similar, ResNet50 outperforms ResNet18 in four out of five labels. Thus, we can say that predtrained ResNet50 is the best model for this dataset.

3 Experimental results

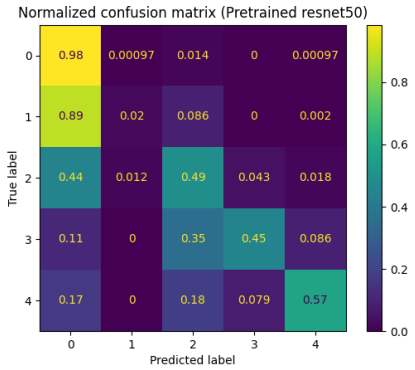
3.1 The highest testing accuracy

Model	Pretrained	W/O Pretrained
ResNet18	0.821	0.737
ResNet50	0.823	0.732

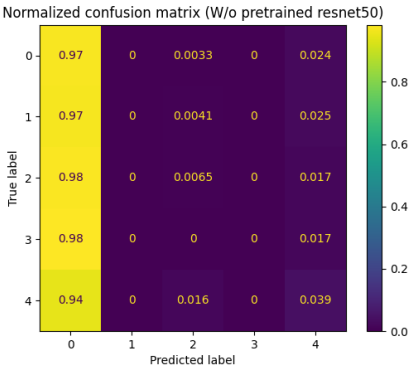
Table 1: Test accuracy for each model(20 epochs)

3.1.1 Screenshot

From Table 1, Figure 3-a and 3-b, it is pretty obvious that the pretrained ResNet18 has higher accuracy on both train and test data. However, the pretrained ResNet18 appears to be overfitting at around



(a) Pretrained ResNet50



(b) W/o pretrained ResNet50

Figure 4: Normalized confusion matrix of ResNet50

```
resnet18, pretrain
Epoch 0, train loss = 0.02705 train acc = 0.742 test acc = 0.776
Epoch 1, train loss = 0.02034 train acc = 0.783 test acc = 0.794
Epoch 2, train loss = 0.01917 train acc = 0.796 test acc = 0.782
Epoch 3, train loss = 0.01845 train acc = 0.803 test acc = 0.801
Epoch 4, train loss = 0.01773 train acc = 0.811 test acc = 0.811
Epoch 5, train loss = 0.01743 train acc = 0.814 test acc = 0.799
Epoch 6, train loss = 0.01696 train acc = 0.819 test acc = 0.813
Epoch 7, train loss = 0.01648 train acc = 0.826 test acc = 0.814
Epoch 8, train loss = 0.01640 train acc = 0.825 test acc = 0.815
Epoch 9, train loss = 0.01595 train acc = 0.830 test acc = 0.812
Epoch 10, train loss = 0.01557 train acc = 0.831 test acc = 0.815
Epoch 11, train loss = 0.01533 train acc = 0.834 test acc = 0.816
Epoch 12, train loss = 0.01498 train acc = 0.838 test acc = 0.817
Epoch 13, train loss = 0.01473 train acc = 0.840 test acc = 0.821
Epoch 14, train loss = 0.01443 train acc = 0.842 test acc = 0.818
Epoch 15, train loss = 0.01410 train acc = 0.844 test acc = 0.813
Epoch 16, train loss = 0.01376 train acc = 0.848 test acc = 0.820
Epoch 17, train loss = 0.01352 train acc = 0.850 test acc = 0.811
Epoch 18, train loss = 0.01322 train acc = 0.854 test acc = 0.804
Epoch 19, train loss = 0.01318 train acc = 0.853 test acc = 0.811
```

(a) Pretrained ResNet18(Epoch 13)

```
resnet18, w/o
Epoch 0, train loss = 0.03032 train acc = 0.731 test acc = 0.734
Epoch 1, train loss = 0.02728 train acc = 0.735 test acc = 0.734
Epoch 2, train loss = 0.02721 train acc = 0.735 test acc = 0.733
Epoch 3, train loss = 0.02703 train acc = 0.735 test acc = 0.734
Epoch 7, train loss = 0.02667 train acc = 0.735 test acc = 0.731
Epoch 8, train loss = 0.02669 train acc = 0.734 test acc = 0.733
Epoch 9, train loss = 0.02660 train acc = 0.735 test acc = 0.733
Epoch 10, train loss = 0.02653 train acc = 0.735 test acc = 0.734
Epoch 11, train loss = 0.02649 train acc = 0.735 test acc = 0.732
Epoch 12, train loss = 0.02640 train acc = 0.734 test acc = 0.734
Epoch 13, train loss = 0.02633 train acc = 0.735 test acc = 0.734
Epoch 14, train loss = 0.02616 train acc = 0.735 test acc = 0.729
Epoch 15, train loss = 0.02598 train acc = 0.735 test acc = 0.734
Epoch 16, train loss = 0.02574 train acc = 0.735 test acc = 0.735
Epoch 17, train loss = 0.02558 train acc = 0.735 test acc = 0.735
Epoch 18, train loss = 0.02530 train acc = 0.736 test acc = 0.734
Epoch 19, train loss = 0.02505 train acc = 0.737 test acc = 0.737
```

(b) W/o pretrained ResNet18(Epoch 19)

Figure 5: Highest Test Accuracy of ResNet18

epoch 15, right after it reach the highest test accuracy.

On the contrary, the test accuracy of without-pretrained ResNet18 is still going up. Intuitively, it might not outperformed the pretrained ResNet18, but it can reach higher accuracy if there are more epochs.

3.1.2 Anything you want to present

ResNet50 is indeed a tough model to train comparing to ResNet18. It consumes several times the GPU RAM ResNet18 needed and the training time is very long. Even if I set "torch.backends.cudnn.benchmark" and " torch.backends.cudnn.enabled" to True.

In addition, when training ResNet18, I can set the number of workers in dataloader to 12 or more. However, when it comes to ResNet50, one worker is maximum. Otherwise, "GPU out of memory" error will pop up. To sum up. ResNet50 takes twice the time to process same dataset, so better start training earlier.

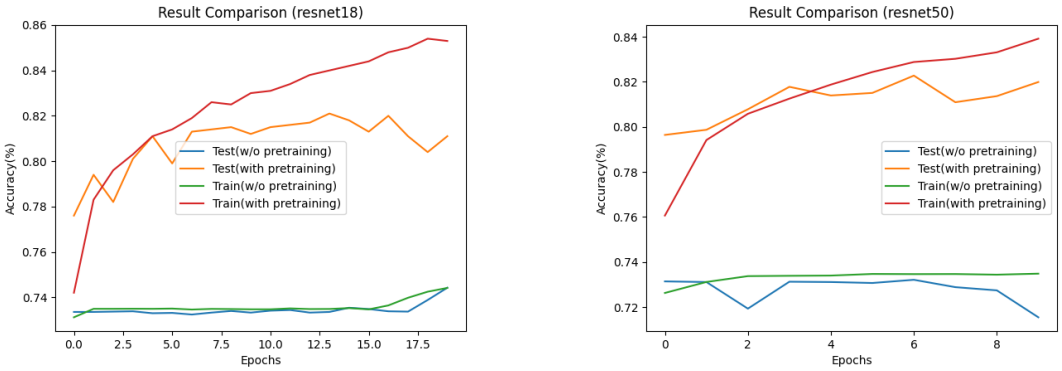
3.2 Comparing figures

Figure 7 and 8 show that pretrained ResNet models perform significantly better in both train and test accuracy. Both pretrained ResNet18 and ResNet50 reach 82% of the accuracy, while without-pretrained models remain less than 75%.

The interesting thing is that ResNet50 models are trained twice longer per epoch but only half the number of epochs. From the time aspect, both kinds of models have similar performance. Nontheless,

```
resnet50, pretrain, start time 2022-07-27 13:59:56.496181
Epoch 0, train loss = 0.04829 train acc = 0.761 test acc = 0.796
Epoch 1, train loss = 0.03906 train acc = 0.794 test acc = 0.799
Epoch 2, train loss = 0.03668 train acc = 0.806 test acc = 0.808
Epoch 3, train loss = 0.03550 train acc = 0.813 test acc = 0.818
Epoch 4, train loss = 0.03402 train acc = 0.819 test acc = 0.814
Epoch 5, train loss = 0.03302 train acc = 0.824 test acc = 0.815
Epoch 6, train loss = 0.03251 train acc = 0.829 test acc = 0.823
Epoch 7, train loss = 0.03156 train acc = 0.830 test acc = 0.811
Epoch 8, train loss = 0.03099 train acc = 0.833 test acc = 0.814
Epoch 9, train loss = 0.03002 train acc = 0.839 test acc = 0.820
resnet50, w/o
Epoch 0, train loss = 0.06101 train acc = 0.726 test acc = 0.731
Epoch 1, train loss = 0.05662 train acc = 0.731 test acc = 0.731
Epoch 2, train loss = 0.05559 train acc = 0.734 test acc = 0.719
Epoch 3, train loss = 0.05517 train acc = 0.734 test acc = 0.731
Epoch 4, train loss = 0.05498 train acc = 0.734 test acc = 0.731
Epoch 5, train loss = 0.05476 train acc = 0.735 test acc = 0.731
Epoch 6, train loss = 0.05448 train acc = 0.735 test acc = 0.732
Epoch 7, train loss = 0.05446 train acc = 0.735 test acc = 0.729
Epoch 8, train loss = 0.05445 train acc = 0.734 test acc = 0.727
Epoch 9, train loss = 0.05420 train acc = 0.735 test acc = 0.715
```

Figure 6: With and w/o pretrained ResNet50: Highest Testing Accuracy.(Epoch 6, Epoch 6)



(a) ResNet18 (b) ResNet50

Figure 7: Result Comparison

from the number of epoch aspect, ResNet50 has more potential.

4 Discussion

This lab makes me realize the importance of large GPU RAM and high CPU GHz. The dataloading part bottleneck the whole process. Large RAM enables larger batches to be store on GPU, therefore, more workers can be set in the parameter of the Dataloader function, which reduce the elapse time of each epoch by 40%. On the other hand, higher GHz directly speed up the process of dataloading.