DL Lab3 Diabetic Retinopathy Detection

0756079 陳冠聞

April 18, 2019

1 Introduction

Degradation problem is that when the number of layer grows, the accuracy of deep neural net will saturate, or even drop dramatically. The degradation problem is an obstruction to train a very deep neural network. In 2015, ResNet was proposed to solve the degradation problem. The idea of residual learning or shortcut connection make it possible to train a network with as deep as 150 layers.

In this lab, I implement the ResNet18 and ResNet50 using PyTorch package. In addition, I train the ResNet to do the task of *Diabetic Retinopathy Detection*, and compared the results between the model trained from scratch and the pretrained model. Finally, I discuss what I have learned from this lab, and the difficulties encountered in this lab.

1.1 Skip/Shortcut Connection

By traditional machine learning theory, the deeper of a neural net, the greater the model capacity it has. Intuitively, the deeper neural net should be as least as good as shallow neural net for training performance. However, increasing network depth does not work by simply stacking layers together. Deep networks are hard to train because of the **vanishing gradient problem** —as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient very small. As a result, as the network goes deeper, its performance gets saturated or even starts degrading rapidly, and it's called **degradation problem**.

The core idea of ResNet is introducing a so-called "identity shortcut connection" that skips one or more layers, as shown in the Figure 3. It provides an alternative for gradient to back propagation. Assume the input of a block is x, and output of origin block is $F(x)$, then the overall output is $F(x) + x$. The intuition behind this type of skip connection is that they have uninterrupted gradient flow from the first layer to the last layer, which tackles the vanishing gradient problem.

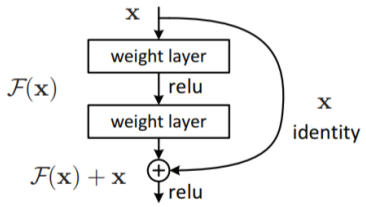


Figure 1: Residual Learning

From another perspective, the skip connections allows for later layers to also learn simple features that are captured in the earlier layers. And the residual block is to learn how to add the residual to the input, rather than need to learn the whole transformation. As shown in Figure 2, the loss surface is much more smoother for the model with skip connection, which means that the model with skip connection is easier to learn.

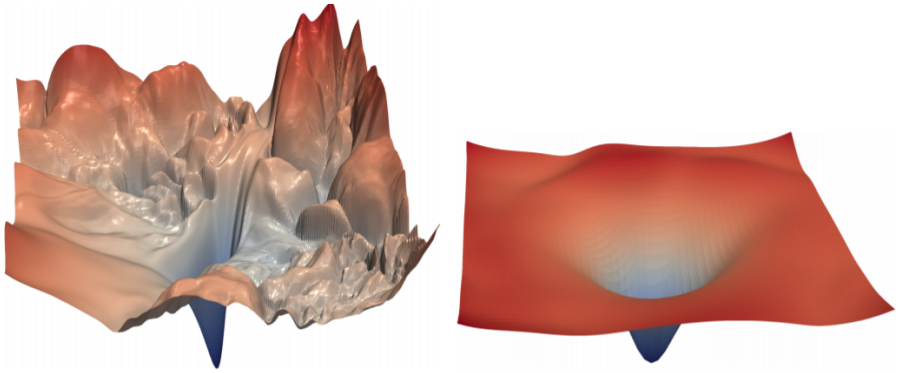


Figure 2: Loss Surface Comparison

2 Experimental Setup

2.1 Details of My Model

From Figure 3[1], We can see that in ResNet18 and ResNet50, there are two different building blocks. The building block for ResNet18 is named *Basic Block*, which is composed of two 3x3 convolution kernel. As for ResNet50, the building block is named as *Bottlenect Block*, which first use 1x1 convolution kernel to reduce the number of channels, then followed by a 3x3 convolution, and finally use 1x1 convolution to restore the number of channels. The implementation details are shown in Figure 4 and Figure 5.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				

Figure 3: ResNet Architecture

[1]Figure 1 and 3 is from *Deep Residual Learning for Image Recognition*
Figure 2 if from *Visualizing the Loss Landscape of Neural Nets*

```python
class BasicBlock(nn.Module):
    expansion = 1
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        # first conv_3x3 (if stride > 1, do stride in first conv_3x3)
        self.conv1 = conv_3x3(in_channels, out_channels, stride)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

        # second conv_3x3
        self.conv2 = conv_3x3(out_channels, out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # If have stride or in_channels does not match with out_channels, identity have to do subsampling
        self.downsample = None
        if stride != 1 or in_channels != out_channels:
            self.downsample = downsample_1x1(in_channels, out_channels, stride)
```

Figure 4: Code Snippet for Basic Block (used in ResNet18)

```python
class Bottleneck(nn.Module):
    expansion = 4
    def __init__(self, in_channels, reduced_channels, stride=1, groups=1):
        super(Bottleneck, self).__init__()
        self.relu = nn.ReLU(inplace=True)

        # Use conv_1x1 to reduce # channels first
        self.conv1 = conv_1x1(in_channels, reduced_channels)
        self.bn1 = nn.BatchNorm2d(reduced_channels)

        # Do conv_3x3 with reduced_channels (if stride > 1, do stride conv in conv_3x3)
        self.conv2 = conv_3x3(reduced_channels, reduced_channels, stride, groups)
        self.bn2 = nn.BatchNorm2d(reduced_channels)

        # Use conv_1x1 to expend # channels
        out_channels = reduced_channels * self.expansion
        self.conv3 = conv_1x1(reduced_channels, out_channels)
        self.bn3 = nn.BatchNorm2d(out_channels)

        # If have stride or in_channels does not match with out_channels, identity have to do subsampling
        self.downsample = None
        if stride != 1 or in_channels != out_channels:
            self.downsample = downsample_1x1(in_channels, out_channels, stride)
```

Figure 5: Code Snippet for Bottleneck Block (used in ResNet50)

After finishing the building blocks, I can build up the ResNet18 and ResNet50 using these building blocks by following the network architecture in Figure 3. The only difference is that origin ResNet is trained on ImageNet with 1,000 classes, and this dataset only has 5 classes. As a result, I use fully-connected

layer which is $R^c \mapsto R^5$ to replace the origin fully-connected layer which is $R^c \mapsto R^{1000}$, where $c = 512$ for ResNet18 and $c = 2048$ for ResNet50. The implementation detail is in Figure 6, and the code for loading pretrained model is in Figure 7. (For concise typesetting, these code snippets are incomplete. Please see the attachment code for more details.)

```python
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=5):
        super(ResNet, self).__init__()
        # Number of channels in each layer
        channels = [int(64 * (2 ** i)) for i in range(4)]
        self.in_channels = channels[0]

        # Do conv_7x7 first
        self.conv1 = nn.Conv2d(3, channels[0], kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(channels[0])
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        # Resiudal Layers (stride=2 after second layer)
        self.layer1 = self._make_layer(block, channels[0], num_blocks[0])
        self.layer2 = self._make_layer(block, channels[1], num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, channels[2], num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, channels[3], num_blocks[3], stride=2)

        # Use avgpool to make the shape become (c,1,1)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))

        # Fully Connected Layer
        out_channels = channels[-1] * block.expansion
        self.fc = nn.Linear(out_channels, num_classes)

        # Initize model weight
        self._init_weight()
```

```python
def resnet18():
    net = ResNet(BasicBlock, num_blocks=[2, 2, 2, 2])
    return net

def resnet50():
    net = ResNet(Bottleneck, num_blocks=[3, 4, 6, 3])
    return net
```

Figure 6: Code Snippet for ResNet

```python
def pretrained_resnet18():
    model_ft = models.resnet18(pretrained=True)
    model_ft.avgpool = nn.AdaptiveAvgPool2d((1,1))
    model_ft.fc = nn.Linear(512, 5)
    return model_ft
```

```python
def pretrained_resnet50():
    model_ft = models.resnet50(pretrained=True)
    model_ft.avgpool = nn.AdaptiveAvgPool2d((1,1))
    model_ft.fc = nn.Linear(2048, 5)
    return model_ft
```

Figure 7: Code Snippet for Loading Pretrained Model

2.2 Details of My Dataloader

I implement the dataloader using the PyTorch class *torch.utils.data.Dataset* and *torch.utils.data.DataLoader*, the detail of implementation is shown in Figure 8.

Data augmentation and normalization is common technique used in machine learning. In general, the more data, the better our ML models will be. Unfortunately, number of data is limited in real world datasets. Data augmentation is a technique to increase the number of data points to make the ML model perform better. On the other hand, data normalization gives the error surface a more spherical shape, which can help the gradient descent to converge more quickly and stably.

In this lab, I use random flip along both horizontal and vertical direction, as well as random rotation within 90 degree as data augmentation. In addition, I compute mean and variance of training data, and normalize it to make it become zero mean and unit variance for each channel. For implementation, I use the *torchvision.transforms* class, and the detail is shown in Figure 9.

```
class RetinopathyDataset(data.Dataset):
    def __init__(self, root, mode, transform=None):
        self.root = root
        self.transform = transform
        self.img_names, self.labels = getData(mode)

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

    def __getitem__(self, index):
        img_name = self.img_names[index] + ".jpeg"
        label = self.labels[index]
        img_path = os.path.join(self.root, img_name)
        img = Image.open(img_path)
        if self.transform!= None:
            img = self.transform(img)
        return img, label
# Datasets
data_root = "data"
train_dataset = RetinopathyDataset(data_root, "train", train_transform)
test_dataset = RetinopathyDataset(data_root, "test", test_transform)

# Loaders
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Figure 8: Code Snippet for Dataloader

```
# Transform
img_size = (512,512)
train_transform = transforms.Compose([
    transforms.Resize(img_size),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.RandomRotation(90),
    transforms.ToTensor(),
    transforms.Normalize((0.3749, 0.2602, 0.1857), (0.2526, 0.1780, 0.1291)),
])

test_transform = transforms.Compose([
    transforms.Resize(img_size),
    transforms.ToTensor(),
    transforms.Normalize((0.3749, 0.2602, 0.1857), (0.2526, 0.1780, 0.1291)),
])
```

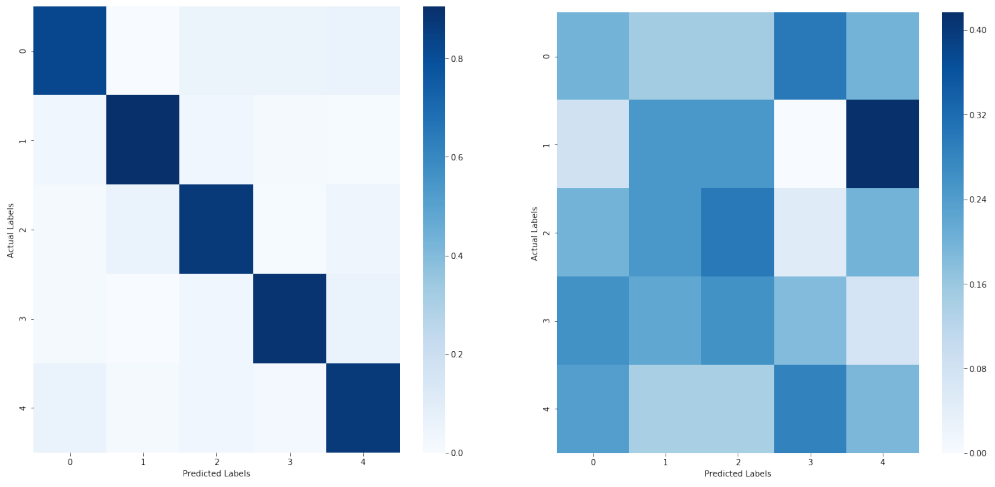
Figure 9: Code Snippet for Data Augmentation and Normalization

2.3 Confusion Matrix

To decide the which model is superior among a group of machine learning models, we must have some metrics to evaluate models. For classification task, accuracy is the most straightforward one. However, sometime the accuracy is not is not a reliable metric, especially for the unbalanced classification task. For example, most results of the medical tests are negative. In such situations, a model that always predict negative will yield high accuracy, but it fact, it should be a very bad model.

One commonly used metric is **confusion matrix**, which is a specific table layout that allows visualization of the performance of a model. Each column of the matrix represents the instances in a predicted class while each row represents the instances in an actual class. Besides for good visualization, confusion matrix can also be used to compute the **AUC-ROC** curve, which is a more reliable metric than accuracy.

In this lab, because the data is unbalanced, I will compare the confusion matrix between models. For better visualization, I use the weighted confusion matrix, whose sum of each row is 1. I will visualize the matrix where the larger value will have dark blue. Due to the fact that the better model should have values closer to 1 in each diagonal element, so the model have darker blue in diagonal element is the better one.



(a) better model (b) worse model

Figure 10: Confusion Matrix Visualiziation

3 Experimental Result

I trained both ResNet18 and ResNet50 using the same hyper-parameters setting. The optimizer is SGD with momentum 0.9, learning rate is set to 0.001 as well as weight decay 0.0005. The batch size is 32 distributed in two 1080Ti GPU, and trained for 10 epochs in total.

3.1 Highest Accuracy

The highest test accuracy 82.60% is achieved by the pretrained ResNet50. As we can observe from Table 1, pretrained models outperform the models trained from scratch about 10%, which is a big gap.

Table 1: Accuracy Comparsion

	ResNet18	ResNet50
From Scratch	73.12%	72.24%
Pretrained	81.52%	82.60%

3.2 Comparison Figures

As shown in Figure 11 and Figure 13, both training and test accuracy of the model trained from scratch can not improve after first epoch, which means that models basically can not learn anything from data after first epoch. Furthermore, we can observe in Figure 12 and Figure 14 that the model trained from scratch almost predict all data as class 0. It's because that the data is highly unbalanced, where about 73% of data belong to class 0, so the model trained from scratch just learn to classify all data as class 0 can still yield about 73% accuracy. Unfortunately, this kind of model is useless in real world. As for the pretrained model, we can see that although it only perform well on class 0, but it not just classify all data as class 0 and perform much better than the model trained from scratch for the other classes, which means it indeed learned something from data.

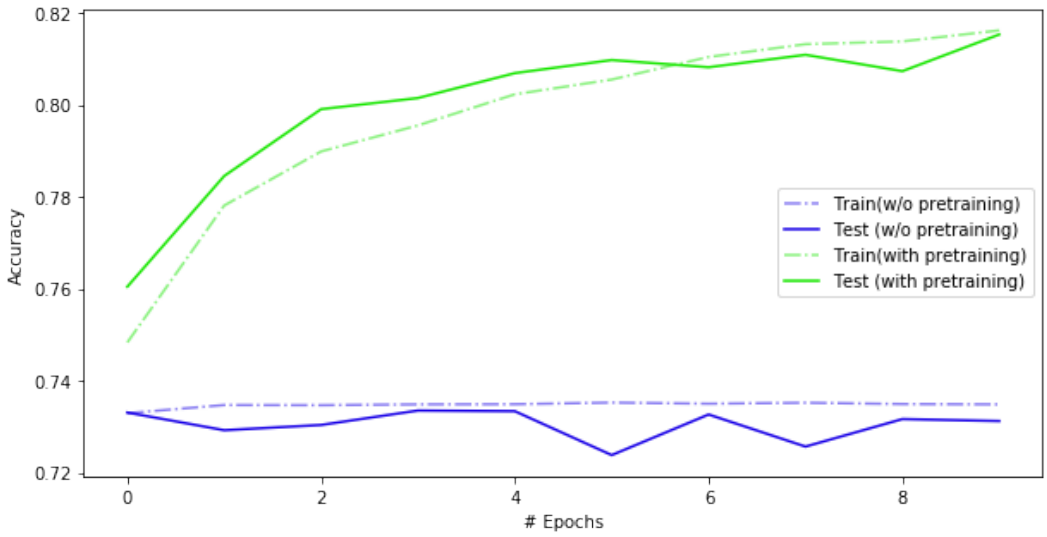
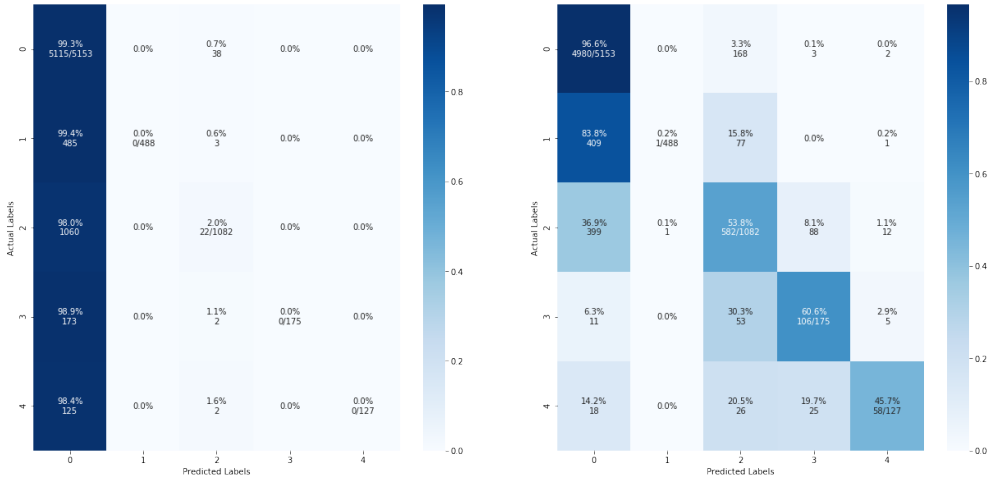


Figure 11: Accuracy Trend for ResNet18



(a) trained from scratch (b) pretrained and finetune

Figure 12: Confusion Matrix of ResNet18

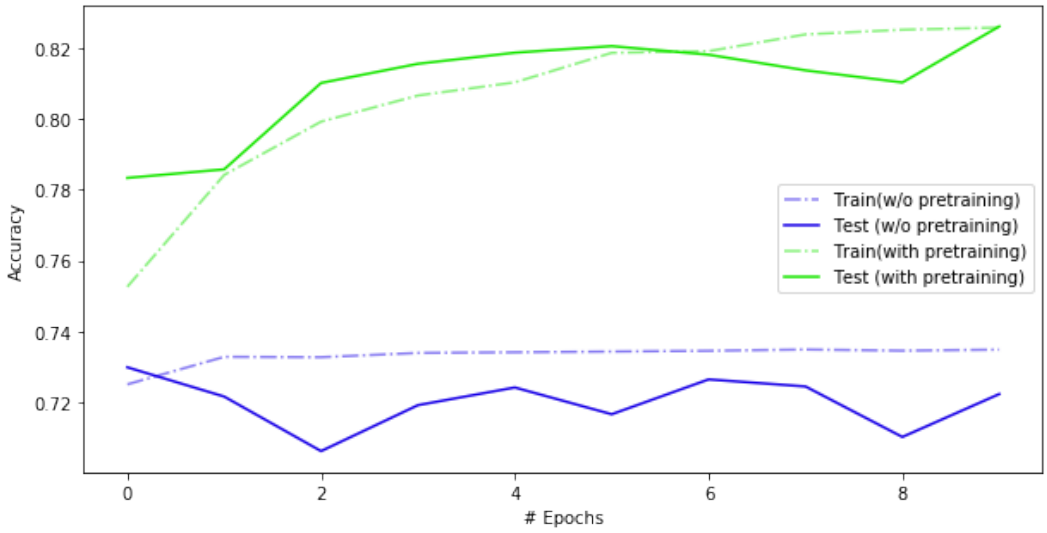
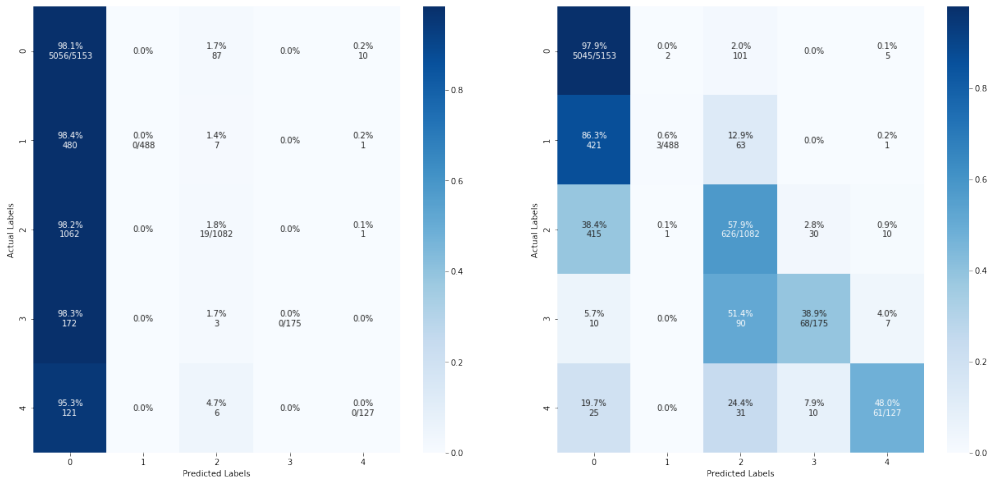


Figure 13: Accuracy Trend for ResNet50



(a) trained from scratch (b) pretrained and finetune

Figure 14: Confusion Matrix of ResNet50

4 Discussion

4.1 Pretrained Models

From the experimental result above, we can see that the pretrained model is actual better than the model trained from scratch in both accuracy as well as confusion matrix. In general, the model pretrained on large dataset then fintune on small dataset will perform better than the model trained from scratch, and I think there are two reasons for that, one is the bad local minimum problem, and another is the problem of overfitting.

First, the learning of the neural network is a non-convex optimization, which the gradient descent approach may converge to local minimum, so the initial weight is critical. In this lab, due to the unbalanced data, the model trained from scratch might quickly learn to classify all data as class 0, and trap in bad local minimum. On the other hand, the pretrained model already have useful convolution kernel, which can extract important feature in the image, so the loss surface around pretrained weight might have better local minimum.

Second, trainging on small dataset is easy for overfitting. The deep neural net have considerable weights, leads to the model capacity is big enough to fit the noise in the training data. Set initial weight to the weight learned from large dataset and fintune on small dataset can be thought as a regularization, which reduce the risk of overfitting.

In this lab, the training accuracy of model trained from scratch can not exceed 75%, and the generalization gap is not big, which means that the model can not even fit the noise. So I think the major problem is that the model trained from scratch traped in bad local minimum.

4.2 Data Augmentation and Normalization

As mention in 2.2, data augmentation and normalization are commonly used in ML. To verify the effect of them, I do experiment with three setting. One is use both data augmentation and normalization, another is only use data augmentation, the last one is only use normalization. The training setting are the same as experimental above, except for the number of epoch is set to 20, and it's because I want to see accuracy trend to nearly converge in this experiment.

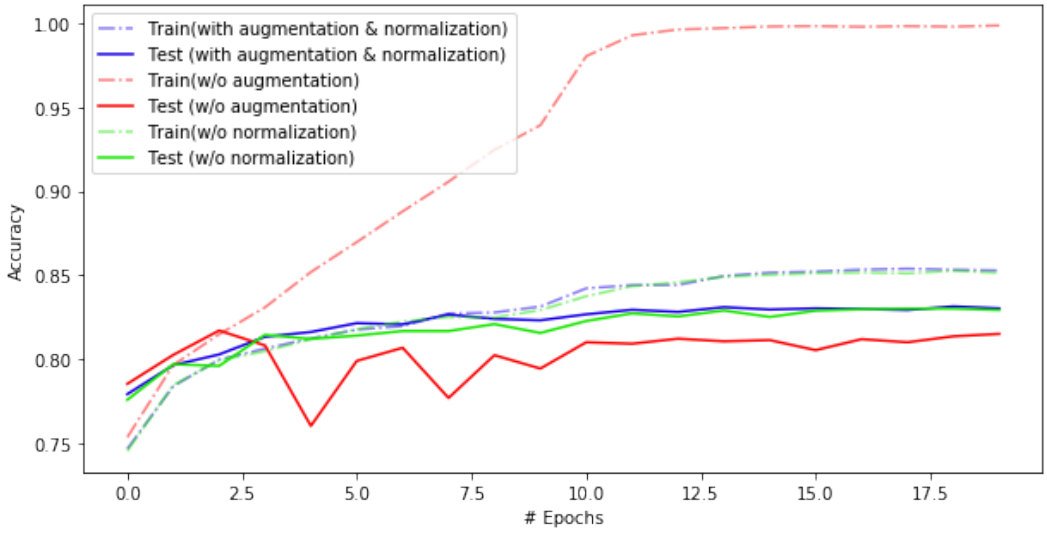


Figure 15: Accuracy Trend Comparison for ResNet50 (pretrained)

From Figure 15 we can see that without data augmentation, the training accuracy converges to 100%, but test accuracy stucks at 80%, which means the model fit the noise in the training data. On the other hand, The training accuracy of model with data augmentation can only reach 85%, but the generalization gap is much smaller, which means what the model have learned is meaningful, not the noise in the training data. For data normalization, we can barely see the difference between the model with and without normalization. I think it's because ResNet has batch-normalization layers. Even the the input data is unnormalized, the output of each layer will still be normalized, so the effect of data normalization is not obvious here.

4.3 Difficulties Encountered in This Lab

The major difficulty for me in this lab is to against the clock. Training for a ResNet on the *Diabetic Retinopathy* dataset for 10 epochs cost at least 3 hours on my machine, and it's not easy to verify the correctness of the implemented model without starting to train and see the result. What's

more, the experiments on different models or hyper-parameters is very time-consuming. Such a tight schedule stops me to do more exploration in this lab. For example, I was planning to deal with the unbalanced data using over-sampling or under-sampling technique, but the close deadline make it impossible.