HW3: Diabetic Retinopathy Detection

Ting-Hsuan, Wang (0756045)

April 17, 2019

1 Introduction

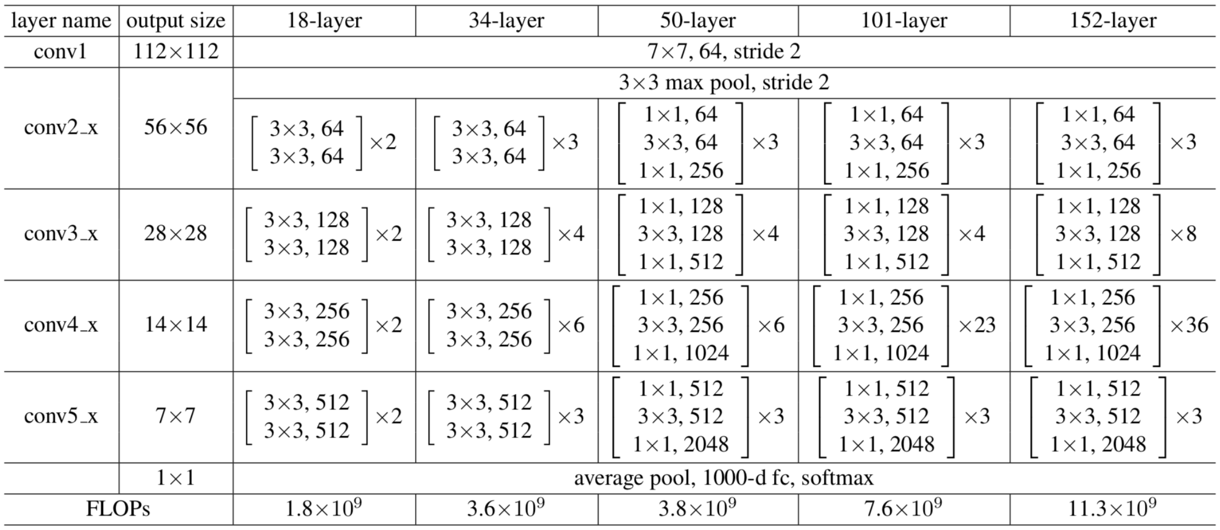
Deep neural networks often suffer from worse results due to vanishing or exploding gradient problem. Neurons that are farther from output layers updates using extremely little or large gradient. This is because during the phase of backpropagation, gradients are multiplied together. Large gradient becomes larger and larger while small gradient becomes smaller and smaller. Thus, the network models can't learn well. Several normalization methods have been proposed to alleviate this problem. The most commonly used one is called batch normalization. With batch normalization as intermediate layer and normalized initialization, networks with tens of layers are able to start converging.
However, when neural networks go beyond dozens of layers (very deep networks), the accuracies during both training and testing phases do not efficiently decrease as expected. Even worse, the accuracies are lower than shallower networks. This is caused by "degradation problem" rather than overfitting. Degradation happens when the neural network has more layers than it needs to model the problem, and hence latter added layers try to learn identity mapping. However, identity mapping is not easy task for layers of non-linear layer to learn. Very deep neural network therefore loses its advantage of having more parameters to model the problem. Fortunately, ResNet, published in 2015, is a sliver lining of training very deep neural network. ResNet proposed a network architecture called residual network, which uses the idea of shortcut connection to solve both the degradation and gradient vanishing/exploding problem. Though shortcut connection has been studied for a long time, it hasn't really achieved impressing results. Nevertheless, the combination of shortcut connection and residual block/bottleneck architecture of ResNet make shortcut connection great again. Many neural network architectures come after ResNet also imitate its usage of shortcut connection, such as unet, densenet and inception v4.
In this homework, we have to implement several image classification models of ResNet variants: ResNet18 and ResNet50 architecture with and without pretrained model. The dataset we use here is Diabetic Retinopathy(DR) dataset, which originated from kaggle competition. DR is an eye disease that a lot people with diabetes have. Early detection of DR can slow down or even averted the progression in vision impairment. Therefore, our goal is to classify different stages of DR. Additionally, we have to evaluate our models with confusion matrices.

2 Experiment setups

2.1 Details of ResNet

2.1.1 Network architecture

In Figure 1, we can see the ResNet variants' network architecture. We'll go through the design of shortcut connection and basic block/bottelneck architecture used in ResNet18 and ResNet50 respectively. Note that the output sizes of our model are different from what Figure 1 show, since our input images is sized 512x512 (ignoring the cropped image size during dataloader). As a result, the correct output sizes are: 256x256(conv1), 128x128(conv2_x), 64x64(conv3_x), 32x32(conv4_x), 16x16(conv5_x), and lastly, 1x1 for final output layers. The FLOPs in the last column indicate the number of variable used in ResNet. FLOPs can be varied due to different kinds of implementation of shortcut connection.



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				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 1: Network architecture of ResNet variants

2.1.2 Residual learning

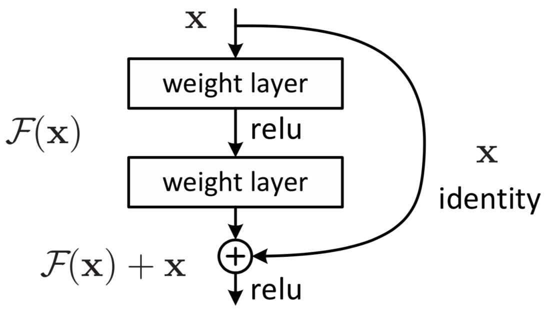


Figure 2: Residual Learning

ResNet introduces a deep residual learning framework as shown in Figure 2. We defined $H(x)$ as the desired underlying mapping and $F(x)$ as the mapping that the framework learns. Based on traditional deep learning model, our goal is to minimize $L(F(x), H(x))$ such that $F(x)$ is possibly familiar with $H(x)$. Yet in residual learning framework, we modify the learned mapping as $F(x) + x$. The $+x$ is called shortcut connection. It connects the input with the last output layer. The idea behinds this is that we want the framework to learn more easily if the mapping is nearly identity.

2.1.3 Shortcut connection

$$y_l = h(x_l) + F(x_l, W_l) \tag{1}$$

$$x_{l+1} = f(y_l) \tag{2}$$

Shortcut connection resolves the problem of degradation and gradient vanishing/exploding as discussed in "Introduction". Here, we're digging into the mathematical part of shortcut connection to fully understand the principle behinds it.
As denoted in the previous section, our goal is to learn $F(x) + x$ for residual learning. The meaning of symbols in equation 1 and equation 2 are as follows: y_l is the output of layer $lh(x_l)$ is an identity mapping and f is a ReLU function applied after addition of short cut and the orginal block. We can further rewrite these two equations as equation 3 by assume today f is also an identity mapping.

$$x_{l+1} = x_l + F(x_l + W_l) \tag{3}$$

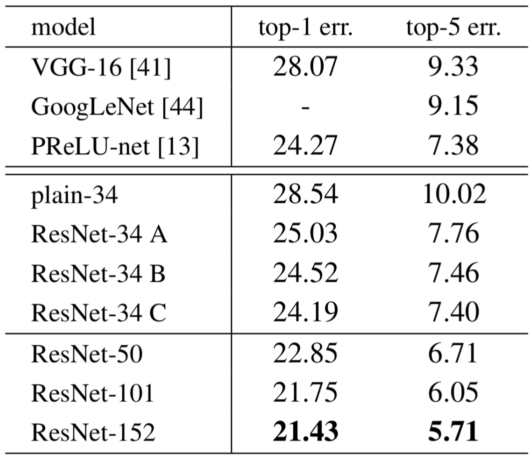
Therefore, the expanding the equations above from layer begins with l to up to L, we get:

$$x_L = x_l + F(x_l + W_l) = x_{l-1} + F(x_{l-1} + W_{l-1}) + F(x_l + W_l) = ... = x_l + \sum_{i=l}^{L-1} F(x_i, W_i) \tag{4}$$

We can now clearly see that any deeper unit L and any shallower unit l are connected together. In other words, no matter how blocks of residual layers we have, the input can always reaches the last layer in forward pass. As for backpropagation, after we conduct derivative to layer x_l, we get:

$$\frac{\partial \epsilon}{\partial x_l} = \frac{\partial \epsilon}{\partial x_L}\frac{\partial x_L}{\partial x_l} = \frac{\partial \epsilon}{\partial x_L}(1 + \frac{\partial}{\partial x_l}\sum_{i=l}^{L-1} F(x_i, W_i)) \tag{5}$$

The result we get consist of two part, $\frac{\partial \epsilon}{\partial x_L}$ and $\frac{\partial \epsilon}{\partial x_L}\frac{\partial}{\partial x_l}\sum_{i=l}^{L-1} F(x_i, W_i)$. The first part ensures that any shallower unit l can get gradients from any deeper unit L. The second part propagates the through the weight terms. From now, we can say that skip connection can solve gradient vanishing/exploding and degradation problem under the condition that $h(x_l)$ and $f(x_l)$ are both identity mapping.
However, in reality, we have many chooses and ways to design skip connection and residual



model	top-1 err.	top-5 err.
VGG-16 [41]	28.07	9.33
GoogLeNet [44]	-	9.15
PReLU-net [13]	24.27	7.38
plain-34	28.54	10.02
ResNet-34 A	25.03	7.76
ResNet-34 B	24.52	7.46
ResNet-34 C	24.19	7.40
ResNet-50	22.85	6.71
ResNet-101	21.75	6.05
ResNet-152	**21.43**	**5.71**

Figure 3: Error rate(10-crop testing) on ImageNet validation.

block design. For simplicity, we only discuss the design of skip connection here. Three ways are mentioned in the ResNet paper to implement shortcut connection. (A) zero-padding shortcuts are used for increasing dimensions, and all shortcuts are parameter-free (B) projection shortcuts are used for increasing dimensions, and other shortcuts are identity; and (C) all shortcuts are projections. Experiment results on ImageNet validation are showed in Figure 8. As the ResNet paper suggest, (A) is suitable for comparing to plain deep network since it doesn't add any parameters. (B) and (C) have comparable results with (C) performs a little bit better. Nonetheless, using (C) to implement shortcut connection will lead to increasing huge amount of parameters. To reduce time complexity and model sizes, the paper authors choose to implement ResNet with (B). We thus follow their suggestion and implement shortcut connection using the same way.

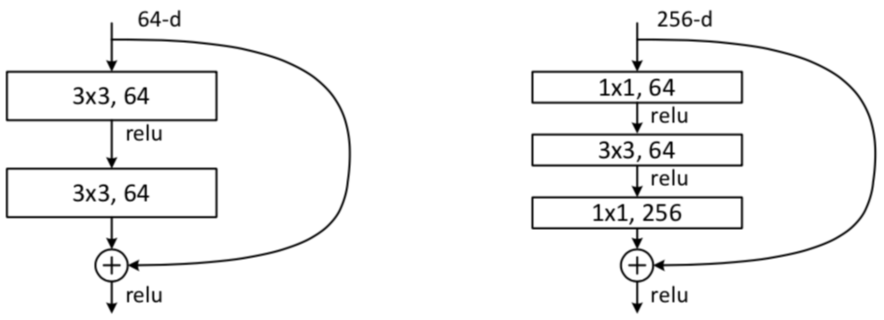


Figure 4: The design of shallower network and deeper network in ResNet. They're called block and bottleneck architecture respectively.

2.1.4 Basic block architecture

Basic block architecture on the left of Figure 4 is used to implement ResNet18 and ResNet34, which are relatively shallower than ResNet50 and above. On the first layer of a basic block, a 3x3 convolution with stride 2 is applied to shrink the output size. Since the input size and output size of first basic block are not the same, when adding the short cut term at the last layer of the block, we have to do projection on the short cut. We implement the projection as a 1x1 convolution with kernel size and stride both equal to 1.

2.1.5 Bottleneck architecture

Bottleneck architecture on the right of Figure 4 is used to implement ResNet50 above. Actually, we can implement ResNet50 and above using basic block architecture described before, yet the number of parameters arises a lot which is not we want. We thus modify the basic block architecture by putting 1x1 convolution at the start and end of the residual function with a 3x3 convolution in the middle. The 1x1 convolution is used to reduce and then restore the dimensions. Otherwise, 3x3 convolution will have larget input/output dimensions. It is worth noticing that the time complexity of basic block and bottleneck architecture are similar.

2.2 Details of Dataloader

Dataloader in pytorch is mainly used to preprocess the data and to dispatch the data into batches. We've tried several ways to do data augmentation and preprocessing, including: random cropping, random cropping and resizing, random flipping, color jittering, random rotate, normalization. From our experiments, we find that random cropping in size 480, random flipping and normalization performs the best. Other preprocess methods do not really help or even impede the model from learning properly. Nevertheless, to increase the speed of training and testing, we set number of worker threads to 4 and batch size to 36 and 10 for ResNet18 and ResNet50 respectively for both training and testing dataloaders. See Figure 5 for more information.

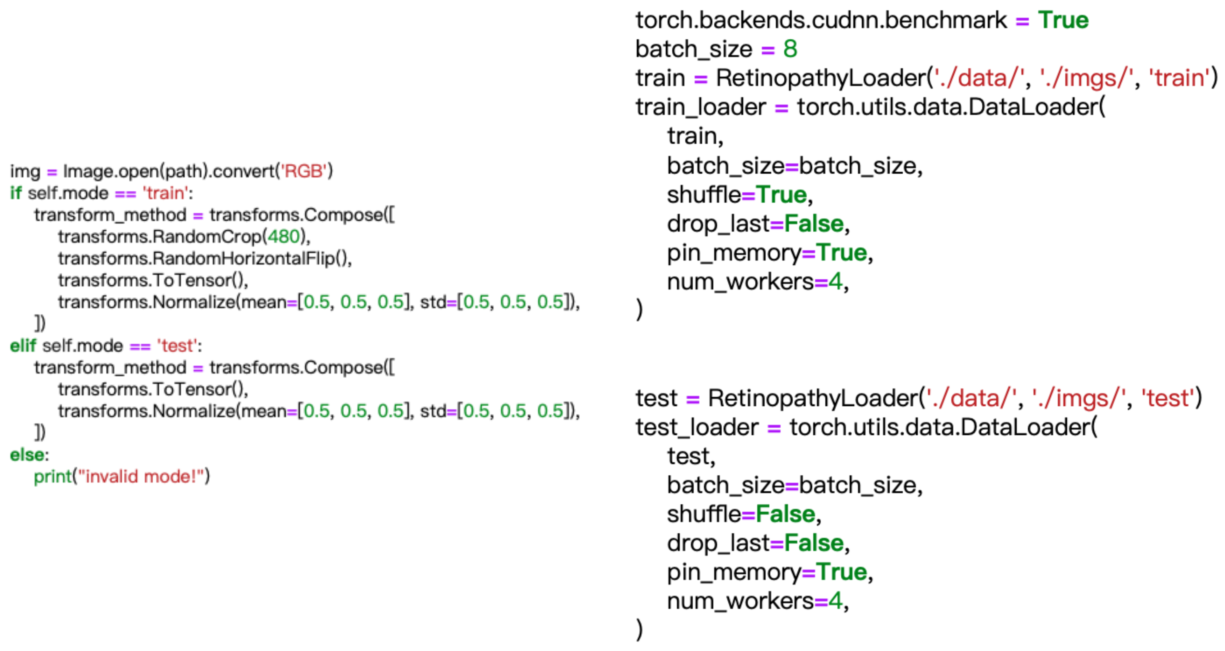
2.3 Evaluation through the confusion matrix

Confusion matrix is useful for calculating recall, precision, specificity and accuracy. We can see that with models without pretraining, ResNet18 and ResNet50, perform a lot worse than the pretrain models, ResNet18_pretrain and ResNet50_pretrain. We can also say that ResNet18 and ResNet50 nearly do not have any ability to classify. They always guess class 0 on all data.

3 Experimental results

3.1 Testing accuracy

See Figure 8 for the highest testing accuracies.



```
img = Image.open(path).convert('RGB')
if self.mode == 'train':
    transform_method = transforms.Compose([
        transforms.RandomCrop(480),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
    ])
elif self.mode == 'test':
    transform_method = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
    ])
else:
    print("invalid mode!")
```

```
torch.backends.cudnn.benchmark = True
batch_size = 8
train = RetinopathyLoader('./data/', './imgs/', 'train')
train_loader = torch.utils.data.DataLoader(
    train,
    batch_size=batch_size,
    shuffle=True,
    drop_last=False,
    pin_memory=True,
    num_workers=4,
)


test = RetinopathyLoader('./data/', './imgs/', 'test')
test_loader = torch.utils.data.DataLoader(
    test,
    batch_size=batch_size,
    shuffle=False,
    drop_last=False,
    pin_memory=True,
    num_workers=4,
)
```

Figure 5: Training dataloader and testing dataloader.

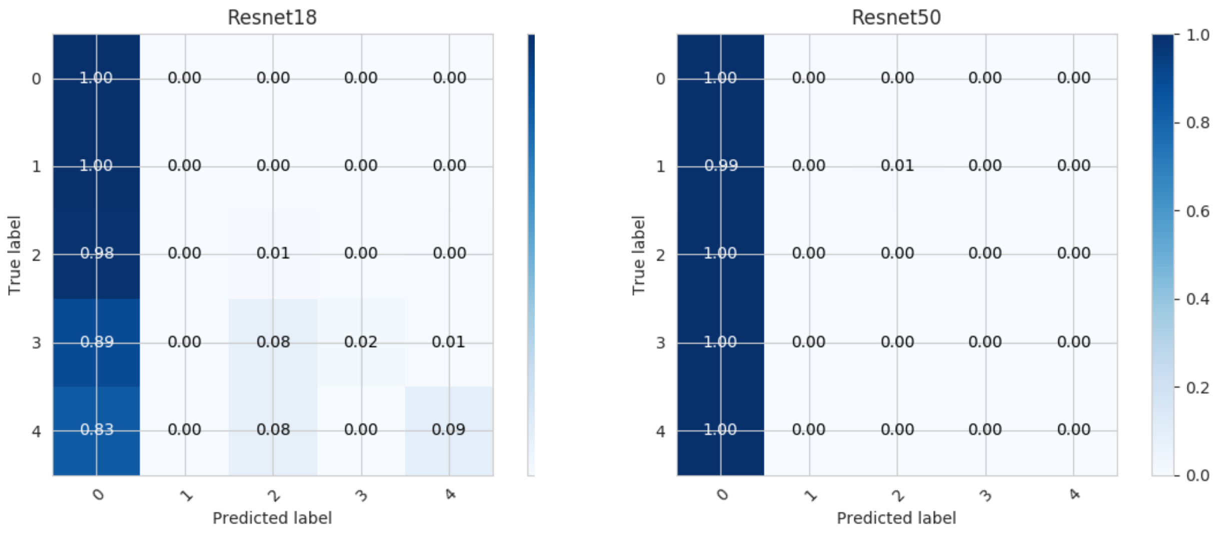
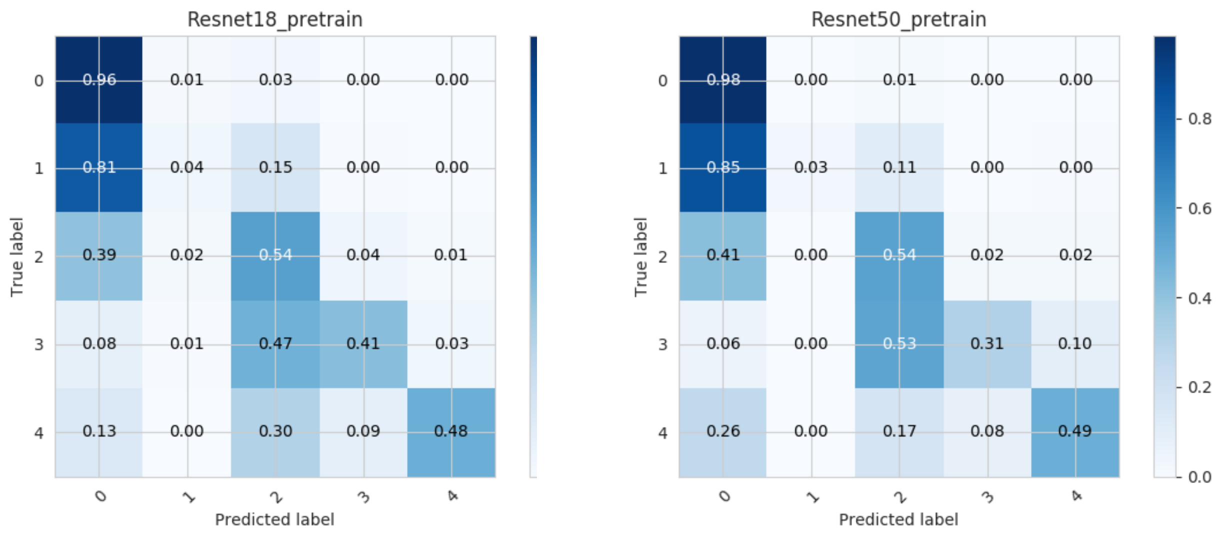


Figure 6: Confusion matrix of non-pretrained ResNet.

3.2 Comparison figures

From Figure 9, we can see that ResNet50 with pretrained model gain the highest accuracy, while the models without pretraining seem to stuck at local minimum. Note that overfitting occurs when the models are trained for too many epochs. See "Discussion" for more information and observation on overfitting.



True label	0	1	2	3	4
0	0.96	0.01	0.03	0.00	0.00
1	0.81	0.04	0.15	0.00	0.00
2	0.39	0.02	0.54	0.04	0.01
3	0.08	0.01	0.47	0.41	0.03
4	0.13	0.00	0.30	0.09	0.48

True label	0	1	2	3	4
0	0.98	0.00	0.01	0.00	0.00
1	0.85	0.03	0.11	0.00	0.00
2	0.41	0.00	0.54	0.02	0.02
3	0.06	0.00	0.53	0.31	0.10
4	0.26	0.00	0.17	0.08	0.49

Figure 7: Confusion matrix of pretrained ResNet.

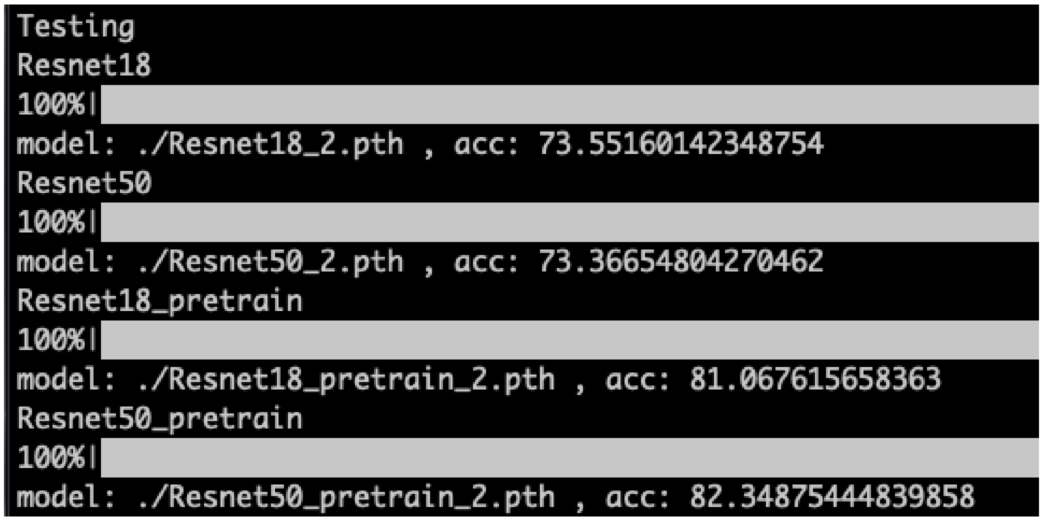
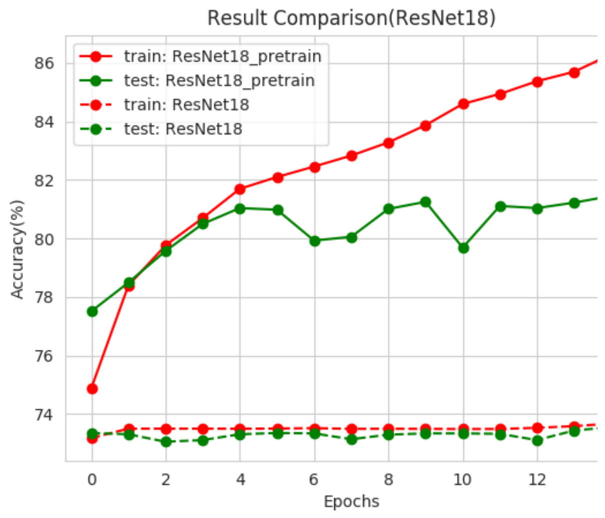

```
Testing
Resnet18
100%|
model: ./Resnet18_2.pth , acc: 73.55160142348754
Resnet50
100%|
model: ./Resnet50_2.pth , acc: 73.36654804270462
Resnet18_pretrain
100%|
model: ./Resnet18_pretrain_2.pth , acc: 81.067615658363
Resnet50_pretrain
100%|
model: ./Resnet50_pretrain_2.pth , acc: 82.34875444839858
```

Figure 8: Highest testing accuracies of all ResNet variants.

4 Discussion

4.1 The problem of overfitting and imbalanced data

Overfitting is quite common in this classification task. For ResNet18 and ResNet50 pre-trained model, overfitting occurs around 15 epoch and 8 epoch respectively. When overfitting occurs, our training accuracies keep arising while testing accuracies sadly going down. We further inspect the dataset in Figure 10, we find that the data is severely unbalanced. Class 0 takes over 70 percent of the all the data in both training and testing data, while class 1, 2 and 4 take much lower percentage. Data augmentation can be applied on those classes with little amount of data to enhance overall performance.



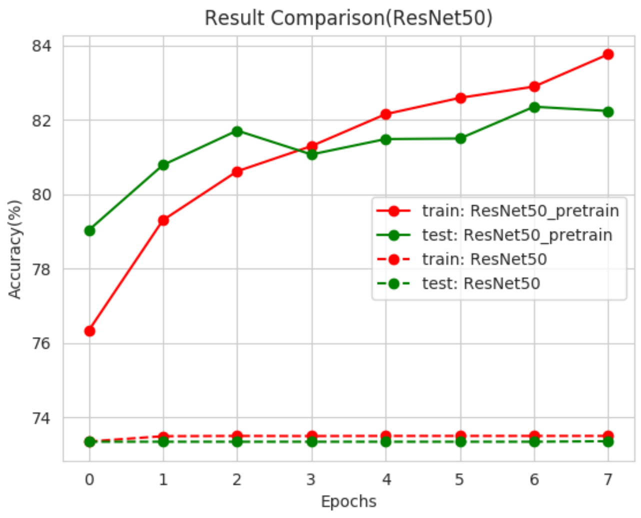
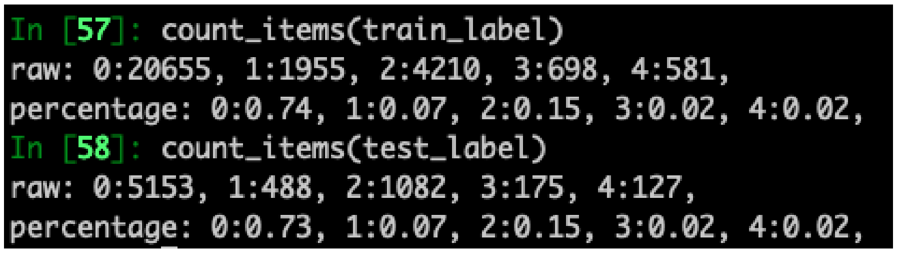


Figure 9: Accuracies during training and testing phases of variants of ResNet.



```
In [57]: count_items(train_label)
raw: 0:20655, 1:1955, 2:4210, 3:698, 4:581,
percentage: 0:0.74, 1:0.07, 2:0.15, 3:0.02, 4:0.02,
In [58]: count_items(test_label)
raw: 0:5153, 1:488, 2:1082, 3:175, 4:127,
percentage: 0:0.73, 1:0.07, 2:0.15, 3:0.02, 4:0.02,
```

Figure 10: The label counts in training and testing data.

4.2 Future work

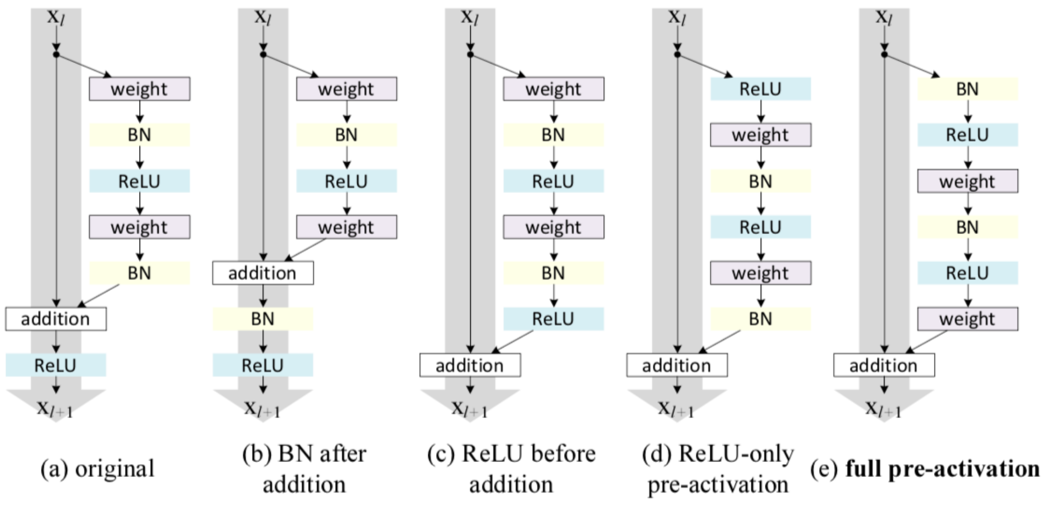


Figure 11: Various usages of activation in paper Identity Mappings in Deep Residual Networks"".

The implementation of ResNet residual block is post-activation. That is, adding ReLU and batch normalization term after convolution layers. Also, another ReLU is performed after the addition of short cut and block/bottleneck architecture. A later paper published by the same author of ResNet in Figure 11 in 2016 shows that, full pre-activation is more helpful for the ResNet to update. This kind of architecture not only converges faster but also avoids overfitting in ResNet1001 compare to the original architecture. We'll may try this in the future.

5 Reference

1. Deep Residual Learning for Image Recognition
2. Identity Mappings in Deep Residual Networks
3. Pytorch ResNet
4. Much thanks to Jxcode for providing many useful information and tricks on this homework.