DLP-HW4-2

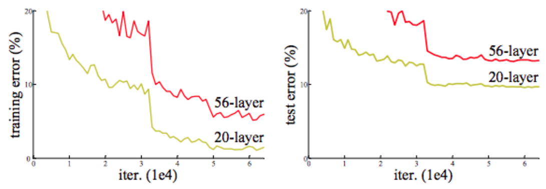
李啟安　機器人學程　310605015

I. Introduction

We will implement ResNet18 and ResNet50 and try to compare the difference between loading pretrained weight or not. We create our own dataloader, model and confusion matrix.

A. How deep can we go?

Although the error rate will decrease as we go deeper ina network because the model will be more robust . However, after attaining a minimum value, the error rate starts increasing again. This happens due to the exploding and vanishing gradient descent problem.



B. ResNet introduction

ResNet[1] [2] or we can say residual network have proved to be quite efficient to solve this problem. They implement a ”shortcut” and this allow us to take activation from one layer and feed it to another layer while sustaining the learning parameters of the network in deeper layers.

II. Experiment setups

A. The details of your model (ResNet)

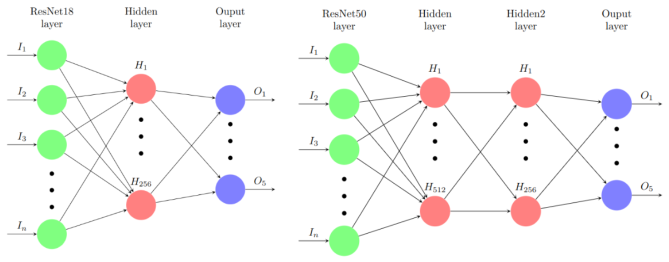
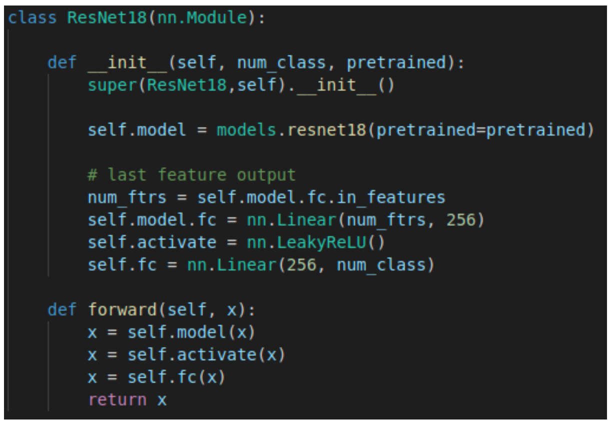


Fig. 1. ResNet18 and ResNet50 Fintune Architecture

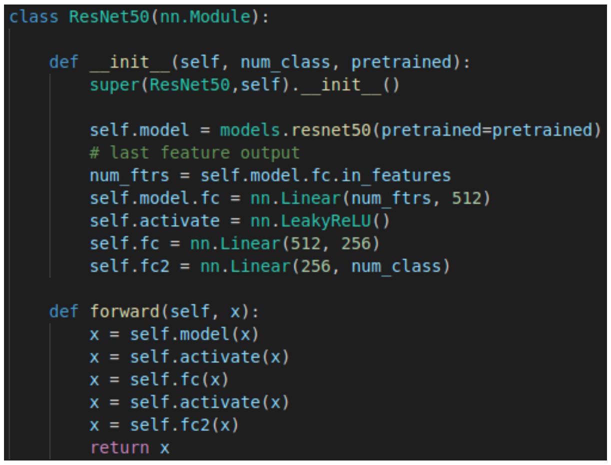
In this experiment, we use two models ResNet18 and ResNet50. We can directly take model architecture from torchvision. If we set the **pretrained** parameters to **True**, then we can use use the pretrained weight. However, we should design the final layer by ourself to meet to output classes.

1) ResNet18: I build a hidden layer after the ResNet18 output layer. The hidden layer has 256 neurons. The output layer has 5 neurons. The activation function is **LeakyReLU**.



```python
class ResNet18(nn.Module):

    def __init__(self, num_class, pretrained):
        super(ResNet18,self).__init__()

        self.model = models.resnet18(pretrained=pretrained)

        # last feature output
        num_ftrs = self.model.fc.in_features
        self.model.fc = nn.Linear(num_ftrs, 256)
        self.activate = nn.LeakyReLU()
        self.fc = nn.Linear(256, num_class)

    def forward(self, x):
        x = self.model(x)
        x = self.activate(x)
        x = self.fc(x)
        return x
```

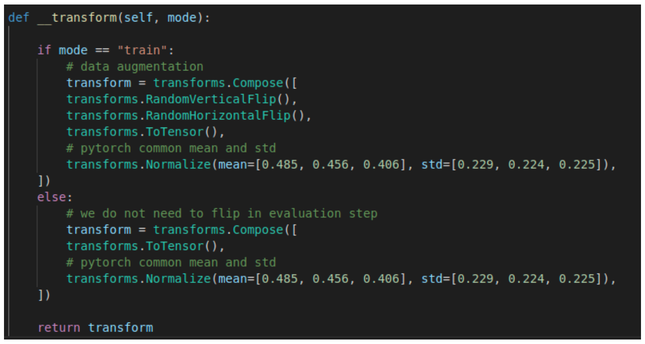
2) ResNet50: I build two hidden layers after the ResNet50 output layer. The hidden layer1 has 512 neurons while hidden layer2 has 256 neurons .The output layer has 5 neurons. The activation function is **LeakyReLU**.



```python
class ResNet50(nn.Module):

    def __init__(self, num_class, pretrained):
        super(ResNet50,self).__init__()

        self.model = models.resnet50(pretrained=pretrained)
        # last feature output
        num_ftrs = self.model.fc.in_features
        self.model.fc = nn.Linear(num_ftrs, 512)
        self.activate = nn.LeakyReLU()
        self.fc = nn.Linear(512, 256)
        self.fc2 = nn.Linear(256, num_class)

    def forward(self, x):
        x = self.model(x)
        x = self.activate(x)
        x = self.fc(x)
        x = self.activate(x)
        x = self.fc2(x)
        return x
```

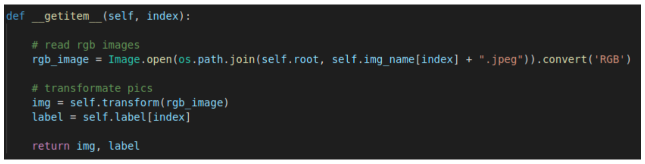
B. *The details of your Dataloader*

The most challenging part in **RetinopathyLoader** is how to transform the images. Function-**transform** can let image rotate, normalize and also convert it to tensor type. Function-**getitem** will read the rgb images. And transform all the images. Then return the data with label. Function-**len** returns the size of dataset. Function-**getData** will return the dataset and labels depending on the mode you choose (train or test).



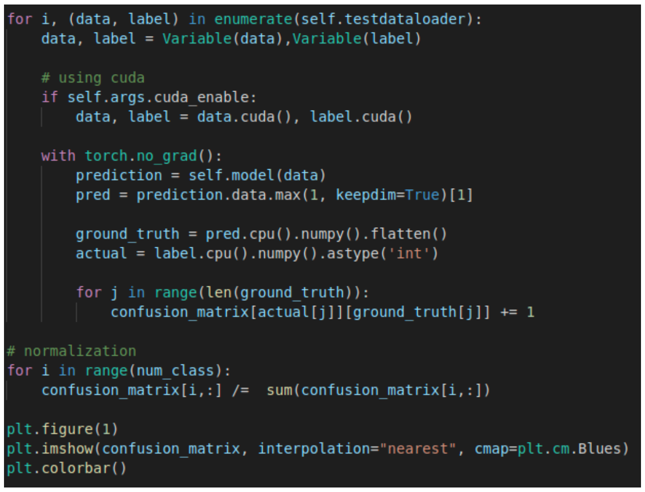
```python
def __transform(self, mode):

    if mode == "train":
        # data augmentation
        transform = transforms.Compose([
        transforms.RandomVerticalFlip(),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        # pytorch common mean and std
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ])
    else:
        # we do not need to flip in evaluation step
        transform = transforms.Compose([
        transforms.ToTensor(),
        # pytorch common mean and std
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ])

    return transform
```

Fig. 2. Image transform



```python
def __getitem__(self, index):

    # read rgb images
    rgb_image = Image.open(os.path.join(self.root, self.img_name[index] + ".jpeg")).convert('RGB')

    # transformate pics
    img = self.transform(rgb_image)
    label = self.label[index]

    return img, label
```

Fig. 3. Pytorch - getitem



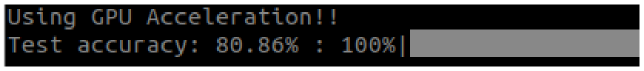
```python
for i, (data, label) in enumerate(self.testdataloader):
    data, label = Variable(data),Variable(label)

    # using cuda
    if self.args.cuda_enable:
        data, label = data.cuda(), label.cuda()

    with torch.no_grad():
        prediction = self.model(data)
        pred = prediction.data.max(1, keepdim=True)[1]

        ground_truth = pred.cpu().numpy().flatten()
        actual = label.cpu().numpy().astype('int')

        for j in range(len(ground_truth)):
            confusion_matrix[actual[j]][ground_truth[j]] += 1

# normalization
for i in range(num_class):
    confusion_matrix[i,:] /=  sum(confusion_matrix[i,:])

plt.figure(1)
plt.imshow(confusion_matrix, interpolation="nearest", cmap=plt.cm.Blues)
plt.colorbar()
```

Fig. 4. Confusion matrix code

C. *Describing your evaluation through confusion matrix*

Because we need to output 5 different class, so we use a 5*5 table to visualize label and prediction. Then, we will do the regularization. Each row should sum up to be 1.

III. Experimental results



```
Using GPU Acceleration!!
Test accuracy: 80.86% : 100%|
```

Fig. 5. Accuracy Screenshot

A. *The highest testing accuracy*

1) *Screenshot:*

The best accuracy is achieved by **ResNet18 pretrained** model. The result is 80.86 percent. See Fig 5.

By watching the confusion matrix, we can see that the best model is ResNet18 with pretrained model. If we train the network without the pretrained weight, the result will be very bad, see Fig 6 and Fig 8.

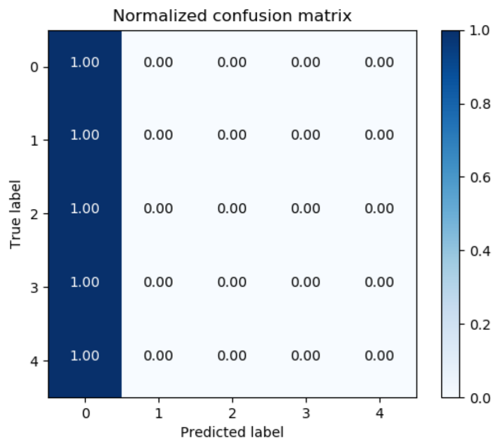


Fig. 6. ResNet18

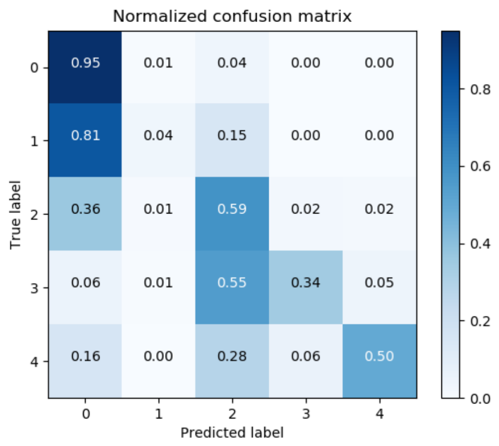


Fig. 7. ResNet18 with pretrained weight

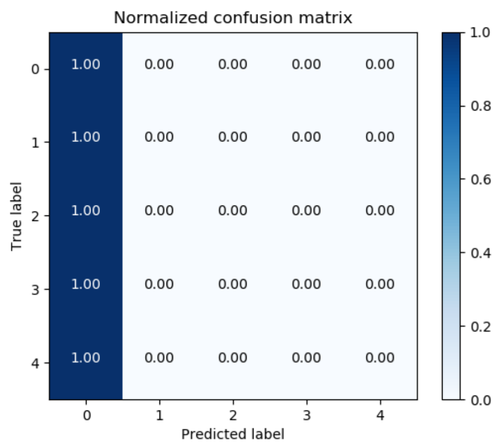


Fig. 8. ResNet50

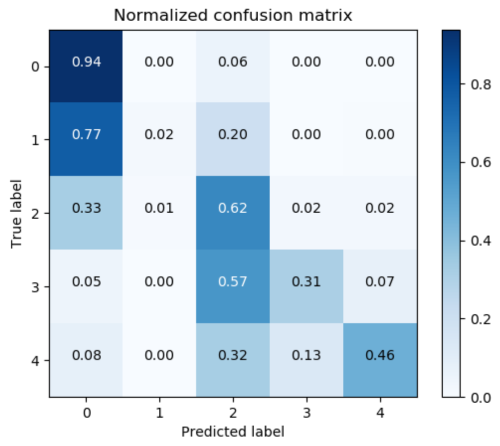


Fig. 9. ResNet50 with pretrained weight

2) Anything you want to present:
See **Table1** for more details. The hyperparameters are shown as follows: batch size = 12, learning rate=0.001 and epochs = 10.

B. *Comparison figures*

1) Plotting the comparison figures (ResNet18/50, with/ without pretraining):

Model	Train Acc	Test Acc
ResNet18	73.505 %	73.342 %
ResNet18(pretrained)	**84.157 %**	**80.857 %**
ResNet50	73.509 %	73.356 %
ResNet50(pretrained)	84.260 %	80.145 %

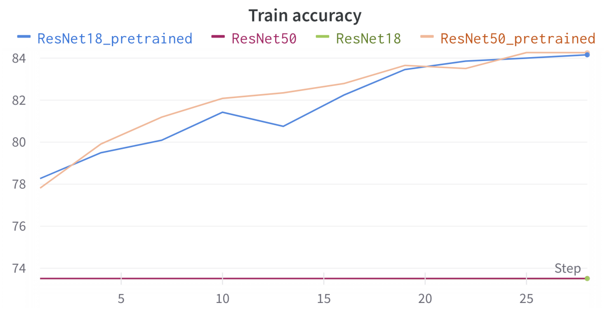


Fig. 10. Training comparison

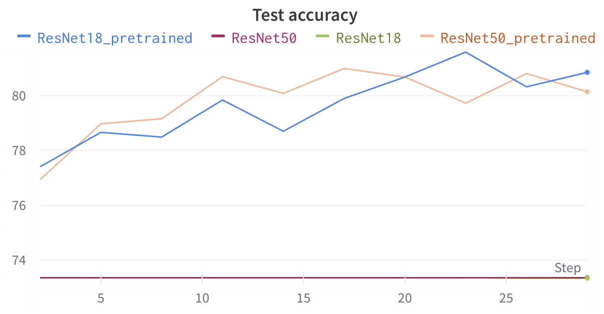


Fig. 11. Testing comparison

IV. Discussion

A. *Anything you want to share*

可以看到有 pretrained model 的模型效果好非常多。分析原因可能是 ResNet 的 pretrained weight 是訓練很久看過很多不同 data 的網路，足夠泛化，可以 extract 很多不同的 feature ，來讓我們的 classification 可以做的更好。然而如果沒有使用 pretrained weight 就會變成有一個很大網路從頭訓練，資料量不夠豐富或是不平衡的資料集都會讓效果變得很差。

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, “Identity mappings in deep residual networks,” in *European conference on computer vision*, pp. 630–645, Springer, 2016.

[2] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.