DLP LAB3 Report

0756138 黃伯凱

April 18, 2019

1 Introduction

我們都知道，Deep Learning的優點之一：**較深的神經網路架構可以提升準確率**。但是其實一般的神經網路如果不停地增加模型深度，是會讓準確度出現飽和、甚至是下降的。

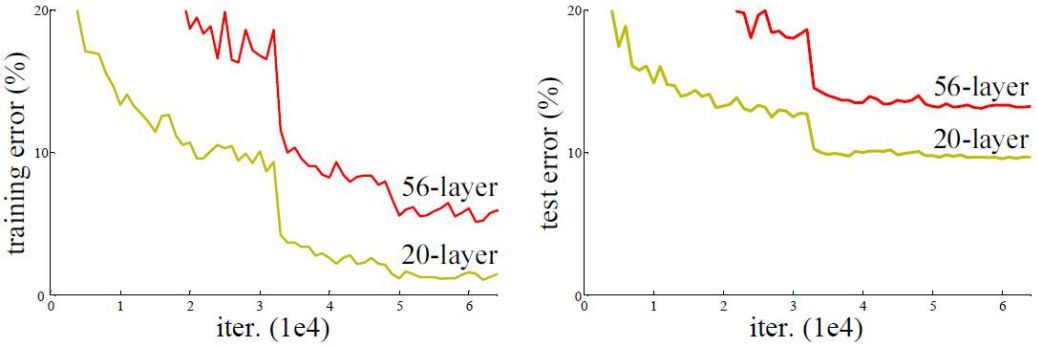


Figure 1: 20層與56層網路在CIFAR-10上的誤差

以56層的網路架構來說，造成這種結果的原因可能是因為：太多層hidden layer導致在backpropagation的時候可能會讓更新值變成0，使得model accuracy比較淺的神經網路還差。有了這樣的假設之後，何凱明博士就提出了Deep Residual Network (ResNet)這樣的架構。

ResNet在2015年被提出，2016年獲得CVPR的最佳論文獎。與過去比較著名的CNN架構：AlexNet、VGG、GoogleNet比起來，152-layer ResNet的網路深度比它們高出了近十倍，而準確率也比過去的CNN架構來的好很多。

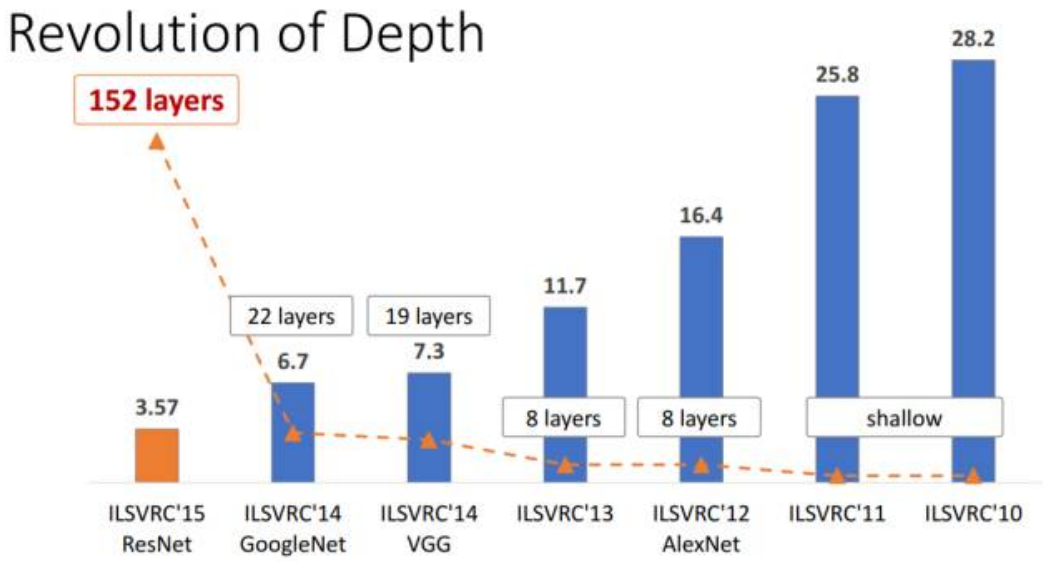


Figure 2: ImageNet分類Top-5誤差

ResNet的架構還有分成：18、34、50、101、152 layers等不同架構，這些數字代表的是這個網路裡convolution layer + fully connected layer的數量總和。這一次的Lab就是要討論這個在CNN有重大突破的網路架構，而我會針對18 layers和50 layers做介紹，並且會有pretained model的實驗以及without pretrain model的實驗結果。

這次的DataSet來自2015年Diabetic Retinopathy Detection的一個競賽。由於糖尿病患者有機率罹患糖尿病性視網膜病變，而且這個病變很難及時發現，所以希望可以交由Machine Learning、Deep Learning來及早發現症狀。這個DataSet分成5個class，詳細資料：Diabetic Retinopathy Detection。

Paper連結：Deep Residual Learning for Image Recognition

2 Deep Residual Network

2.1 Residual Block

在介紹我的ResNet架構以前，要先介紹ResNet為何會有如此驚人的突破。如同在Introduction中介紹的，一個deep neural network的Accuracy有可能會比shallow neural network還低，這是因為在Backpropagation的時候會出現**Degration**的問題。在backpropagation的時候，多餘的hidden layer會讓gradient趨近於零，所以ResNet有了一套方法解決無法更新weight的問題：**Residual Block** (Figure 3)。

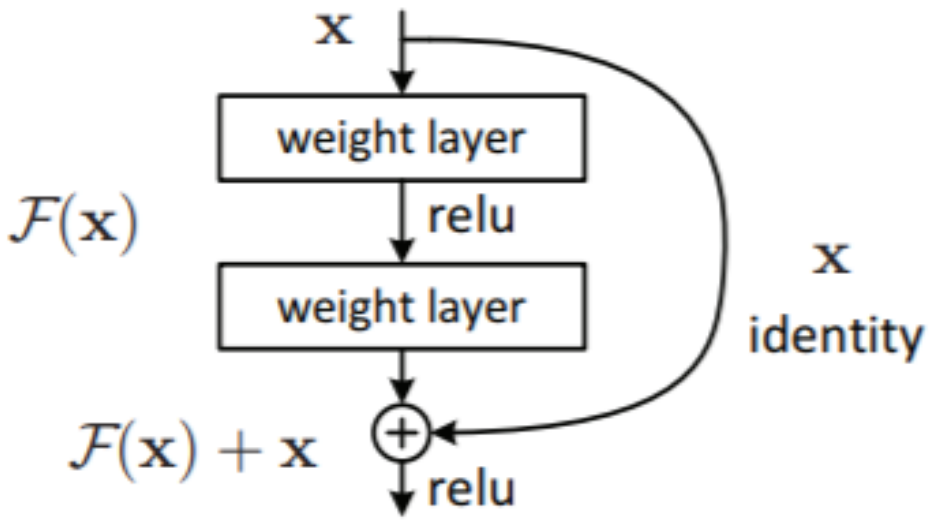


Figure 3: Residual Block

Figure 3中有一條彎曲的線將前面layer的值不經過中間的layer而直接傳到後面，這樣的連接方式稱為**Identity Mapping**，而有經過中間layer的稱為**Residual Mapping**。他假設Residual Mapping的output為F(x)，而Identity Mapping的值仍為x，最後兩個會加總再一起經過ReLU，這樣的連接方式可以在backpropagation時不至於讓gradient消失。

現在假設我們在看第l個Residual Block，他的input是x_l，output就是$y_l = F(x_l, \{W_i\}) + x_l$ (以Figure 3為例，$F(x_l, \{W_i\}) = W_2\sigma(W_1 x_l)$，$\sigma$是ReLU Function)。要特別注意的是，Identity Mapping前後必須要是同樣的Dimension，所以必要時Identity Mapping可以加上Linear Projection W_s：

$$y_l = F(x_l, \{W_i\}) + W_s x_l \tag{1}$$

有了Eq.1之後就可以來看為什麼這樣的架構可以防止gradient消失。假設現在從第l一直到第L個Residual Block的output為：

$$x_L = x_l + \sum_{i=l}^{L-1} F(x_i, \{W_i\}) \tag{2}$$

那麼在backpropagation的時候：

$$\frac{\partial loss}{\partial x_l} = \frac{\partial loss}{\partial x_L} \cdot \frac{\partial x_L}{\partial x_l} = \frac{\partial loss}{\partial x_L} \cdot (1 + \frac{\partial}{\partial x_L} \sum_{i=l}^{L-1} F(x_i, \{W_i\}) \tag{3}$$

在Eq.3小括號中的常數1就是Identity Mapping來的，這個1可以避免gradient通通變成0，進而讓更新weight更加容易。

2.2 Building Block

由於ResNet是架構很深的網路，所以一定會有非常多的parameter必須學會。為了避免太多參數得學，在ResNet的架構中也有了Bottleneck。他的目的就是為了要**降低參數數量**。

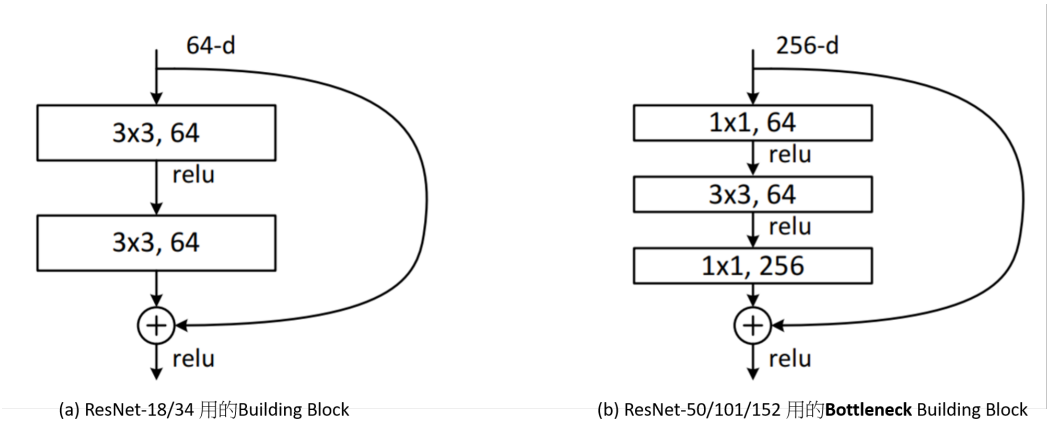


Figure 4: Building Block

Figure 4中的兩個架構通稱為一個**building block**，ResNet架構中用了圖中的這兩種building block，每一個building block是由Convolution Layer和ReLU組成。Figure 4(b)也稱為一個bottleneck。他與Figure 4(a)不同在於他把2層拆成3層來學。

至於為何他可以省參數數量，就得設想若Figure 4(b)也是兩層，且各自是(3x3, 256)的話，那麼總共有$(3 \times 3 \times 256 \times 256 \times 2) = 1179648$個參數要學。這邊用的bottleneck則只需要$(1 \times 1 \times 256 \times 64) + (3 \times 3 \times 64 \times 64) + (1 \times 1 \times 64 \times 256) = 69632$個參數，差了16.94倍。

ResNet的架構是由好幾個Building Block組合而成，ResNet-18/34/50/101/152的差異就在於使用不同的building block以及building block的數量不同。下一個段落就會討論ResNet的整體架構。

2.3 ResNet Architecture

如前面提到的，ResNet有分成ResNet-18/34/50/101/152等數種不同的架構。而他們的差異其實就只在於使用不同的building block以及building block的數量不同。

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 5: ResNet Architecture

Figure 5中的每一個conv_x都是數個相同的building block組成的layer。從圖中可以看出Section 2.2中提過的：**ResNet只用了兩種building block的結構**，各自為2層和3層convolution layer的building block。

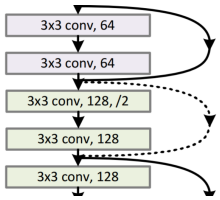


Figure 6: Downsampling

要特別注意的是，**每一個conv_x的第一個convolution layer的stride=2**，這是拿來降低dimension用的(Figure 6中stride=2的layer)。而因為有了這個downsampling，如果Identity Mapping剛好跨越不同dimension的layer時，這一個Mapping就得再加上一個Linear Projection跟著一起降維(Figure 6中的虛線)。

3 Experiment

3.1 Model Detail

3.1.1 Building Block

如上一個章節提到的，ResNet的基本架構是Building Block，而我在這邊把Building Block分成BasicBlock和Bottleneck。BasicBlock是ResNet-18/34使用的Building Block (Figure 7)，而Bottleneck是ResNet-50/101/152使用的(Figure 8)。

```python
class BasicBlock(nn.Module):
    expansion = 1  # filter num difference within a block

    def __init__(self, in_filterNum, o_filterNum, stride=1, downsample=None):
        '''
        Args :
            in_filterNum : Input filter Num.
            o_filterNum : Output filter Num.
        '''
        super(BasicBlock, self).__init__()
        # Both self.conv1 and self.downsample layers downsample the input when stride != 1
        self.conv1 = conv3x3(in_filterNum, o_filterNum, stride)
        self.bn1 = nn.BatchNorm2d(o_filterNum)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(o_filterNum, o_filterNum)
        self.bn2 = nn.BatchNorm2d(o_filterNum)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x  # Identity Mapping

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out
```

Figure 7: Code Snippet for Basic Building Block

```python
class Bottleneck(nn.Module):
    expansion = 4  # filter num difference within a block

    def __init__(self, in_filterNum, o_filterNum, stride=1, downsample=None):
        '''
        Args :
            in_filterNum : Input filter Num.
            o_filterNum : Output filter Num.
        '''
        super(Bottleneck, self).__init__()
        # Both self.conv2 and self.downsample layers downsample the input when stride != 1
        self.conv1 = conv1x1(in_filterNum, o_filterNum)
        self.bn1 = nn.BatchNorm2d(o_filterNum)
        self.conv2 = conv3x3(o_filterNum, o_filterNum, stride)
        self.bn2 = nn.BatchNorm2d(o_filterNum)
        self.conv3 = conv1x1(o_filterNum, o_filterNum * self.expansion)
        self.bn3 = nn.BatchNorm2d(o_filterNum * self.expansion)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        identity = x  # Identity Mapping

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out
```

Figure 8: Code Snippet for Bottleneck

這兩段都有用到conv1x1和conv3x3，這些其實只是一般的convolution layer (Figure 9)，只是為了程式簡潔而拆成不同的function來寫。

```python
# convolution layer with kernel size = 3
def conv3x3(in_planes, out_planes, stride=1):
    return nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride, padding=1, bias=False)

# convolution layer with kernel size = 1
def conv1x1(in_planes, out_planes, stride=1):
    return nn.Conv2d(in_planes, out_planes, kernel_size=1, stride=stride, bias=False)
```

Figure 9: Code Snippet for conv1x1 and conv3x3

3.1.2 ResNet

我的ResNet架構如下，如果是ResNet-18就是ResNet(BasicBlock, [2,2,2,2])，ResNet-50則是ResNet(Bottleneck, [3,4,3,3])。這個4-int list就是每一層conv_中Building Block的數量。

```
class ResNet(nn.Module):

    def __init__(self, block, blockNum, num_classes=5, width_per_group=64):
        '''
        Args :
            block : BasicBlock for ResNet-18/34, Bottleneck for ResNet-50/101/152.
            blockNum : 4-int list of the number of building blocks in conv1_~conv4_
            num_classes : Number of classes
            width_per_group : The number of filter difference between each conv_
        '''
        super(ResNet, self).__init__()
        filterNums = [int(width_per_group * 2 ** i) for i in range(4)]  # [64,128,256,512]
        self.in_filterNum = filterNums[0]  # Initial filter num = 64

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

        # construct conv1_~conv4_
        self.layer1 = self._make_layer(block, filterNums[0], blockNum[0])
        self.layer2 = self._make_layer(block, filterNums[1], blockNum[1], stride=2)
        self.layer3 = self._make_layer(block, filterNums[2], blockNum[2], stride=2)
        self.layer4 = self._make_layer(block, filterNums[3], blockNum[3], stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(filterNums[3] * block.expansion, num_classes)
```

Figure 10: Code Snippet-1 for ResNet

_make_layer就是用來建構一個conv_，有兩種情況要downsample：

1. 當stride不為1時，也就是每一個conv_的第一個building block都要downsample。
2. 當input和output dimension不一樣時，也就是Identity Mapping遇到此狀況要downsample。

```python
def _make_layer(self, block, filterNum, blockNum, stride=1):
    '''
    Args :
        block : BasicBlock for ResNet-18/34, Bottleneck for ResNet-50/101/152.
        filterNum : The filter num in current conv_ layer.
        blockNum : The num of building block in current conv_layer.
    '''
    downsample = None

    # When there are different dimension, then downsample
    if stride != 1 or self.in_filterNum != filterNum * block.expansion:
        downsample = nn.Sequential(
            conv1x1(self.in_filterNum, filterNum * block.expansion, stride),
            nn.BatchNorm2d(filterNum * block.expansion),
        )

    layers = []
    layers.append(block(self.in_filterNum, filterNum, stride, downsample))
    self.in_filterNum = filterNum * block.expansion  # Update filter num by different building block
    for _ in range(1, blockNum):
        layers.append(block(self.in_filterNum, filterNum))

    return nn.Sequential(*layers)
```

Figure 11: Code Snippet-2 for ResNet

```python
def forward(self, x):
    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.maxpool(x)

    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    x = self.layer4(x)

    x = self.avgpool(x)
    x = x.view(x.size(0), -1)
    x = self.fc(x)

    return x
```

Figure 12: Code Snippet-3 for ResNet

3.2 DataLoader Detail

3.2.1 Data Augmentation

我有將這一次的training data做data augmentation。前面幾個都只是resize、旋轉或者翻轉的function。transforms.ToTensor()是將PIL image或者numpy轉成tensor、將dimension (H x W x C)轉成(C x H x W)，且將[0,255]變成[0,1]的function。比較重要的是normalize，本來沒加需要15 epoch左右才能跑到82%，但加上normalize之後只需要5 epoch左右。

```
imgSize = 512
rotateAngle = 30

train_preprocess = transforms.Compose([
    transforms.Scale(imgSize),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.RandomRotation(rotateAngle),
    transforms.ToTensor(),
    transforms.Normalize((0.3749, 0.2601, 0.1856), (0.2526, 0.1780, 0.1291)),
])

test_preprocess = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.3749, 0.2601, 0.1856), (0.2526, 0.1780, 0.1291)),
])
```

Figure 13: Code Snippet for Data Augmentation

3.2.2 Get Data from DataSet

這個部分比較需要注意的是data class裡面的preprocess function。他會將一個batch的image都丟到preprocess的function裡面，training data會經過data augmentation，testing data只會轉成tensor且normalize。

```
def getData(mode):
    if mode == 'train':
        img = pd.read_csv('data\\csv\\train_img.csv')
        label = pd.read_csv('data\\csv\\train_label.csv')
        return np.squeeze(img.values), np.squeeze(label.values)
    elif mode == 'test':
        img = pd.read_csv('data\\csv\\test_img.csv')
        label = pd.read_csv('data\\csv\\test_label.csv')
        return np.squeeze(img.values), np.squeeze(label.values)
```

Figure 14: Code Snippet-1 for getting Data

```
class dataset(Data.Dataset):
    def __init__(self, root=None, mode=None):
        self.mode = mode
        self.img_names, self.labels = getData(mode)
        print('> Found %d datas...' % (len(self.img_names)))

    def __getitem__(self, index):
        img_path = 'data\\imgs\\' + self.img_names[index] + '.jpeg'
        img = Image.open(img_path)
        if(self.mode=='train'):
            img = train_preprocess(img)
        elif(self.mode=='test'):
            img = test_preprocess(img)
#         img = np.transpose(img, (2,0,1))
        return img, self.labels[index]

    def __len__(self):
        ''' Return the size of dataset '''
        return len(self.labels)
```

Figure 15: Code Snippet-2 for Data

我將batch size設為16，也就是說當我在訓練神經網路時，一次都是處理16張image，Figure 15中的__getitem__會一次拿出16張images一起處理。我也有將training data的順序打亂，不按照順序訓練神經網路。

```
BATCH = 16
EPOCH = 20
lr = 0.001
momentum = 0.9
weight_decay = 5.0e-4

trainData = dataset(mode='train')
train_loader = Data.DataLoader(
    dataset = trainData,
    batch_size = BATCH,
    shuffle = True,
    )

testData = dataset(mode='test')
test_loader = Data.DataLoader(
    dataset = testData,
    batch_size = BATCH,
    )
```

Figure 16: Code Snippet for DataLoader

3.3 Confusion Matrix

有時並不能簡單的鑑別出一個model的好或壞，所以我們需要用一些指標去判定它，也作為我們挑選model的依據。Confusion Matrix是用於分類問題的一種常用的方法，它很簡單也很容易實現，這邊我們就來討論confusion matrix的運作模式。

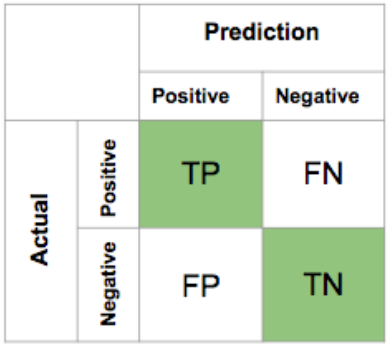


Figure 17: Confusion Matrix

Figure 17是2個class的confusion matrix，橫軸是predict value，縱軸是ground truth。每一格裡面的T、F各自代表True和False；P、N各自代表Positive和Negative。只要有T就代表prediction與ground truth相符，F則為不相符；P和N則代表預測值是1還是0。

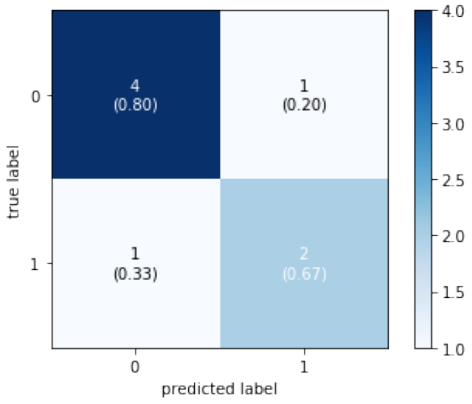


Figure 18: Confusion Matrix Example-1

Figure 18是multiple class confusion matrix的範例。每個格子裡代表該結果出現的次數，括號裡的數值則是normalize後的數值。以此圖為例，總共predict 8次中有6次預測是正確的，6次中的4次是預測結果為0而且預測正確。

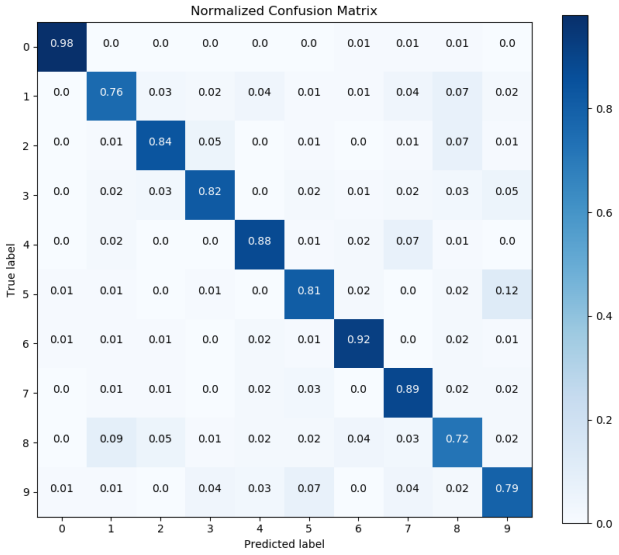


Figure 19: Confusion Matrix Example-2

Figure 19是多個class且已經normalize的範例。Normalize的算法為：

(預測出為Class j的次數) / (實際值為Class i的數量)

所以如果對角線上的值都越接近1，那麼這個model可以視為比較好的model。

4 Result

4.1 Highest Accuracy

ResNet-18的最高test accuracy為82.54%：

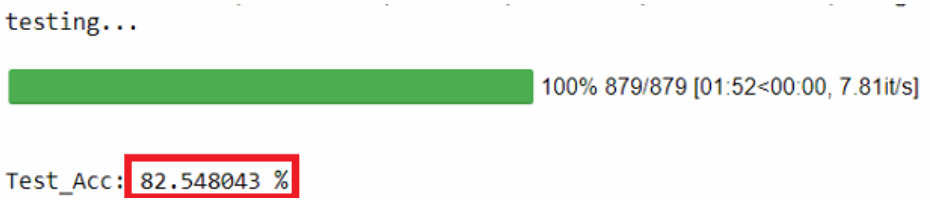


Figure 20: ResNet-18 Best Accuracy

ResNet-50的最高Accuracy為82.90%，但因為我只有存最後一個epoch的model，而最後一個epoch正好並不是最高的accuracy，所以只這邊附上我每一個epoch存下來的testing accuracy：

	Test
0	77.42349
1	80.34164
2	81.20996
3	81.40925
4	80.35587
5	81.97865
6	82.07829
7	82.22064
8	82.74733
9	82.90391
10	82.40569
11	82.14947
12	81.60854
13	81.8363
14	82.23488
15	82.7331
16	82.36299
17	81.73665
18	82.80427
19	81.90747

Figure 21: ResNet-50 Best Accuracy

4.2 Comparison Figure

有pretrain model ResNet-18/50的結果比較如下：

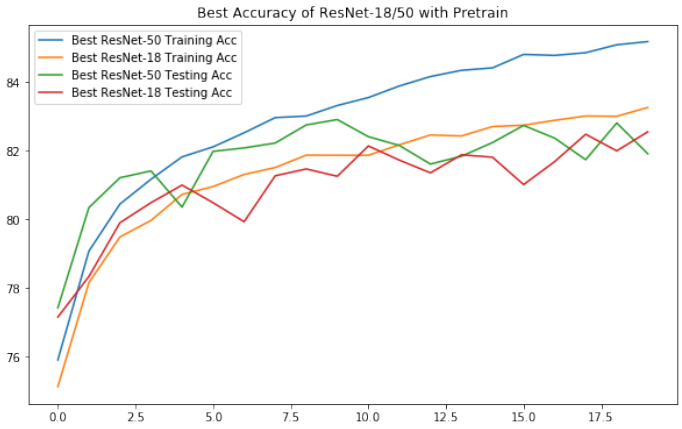


Figure 22: Best Accuracy Comparison with Pretrain

沒有pretrain model ResNet-18/50的結果比較如下：

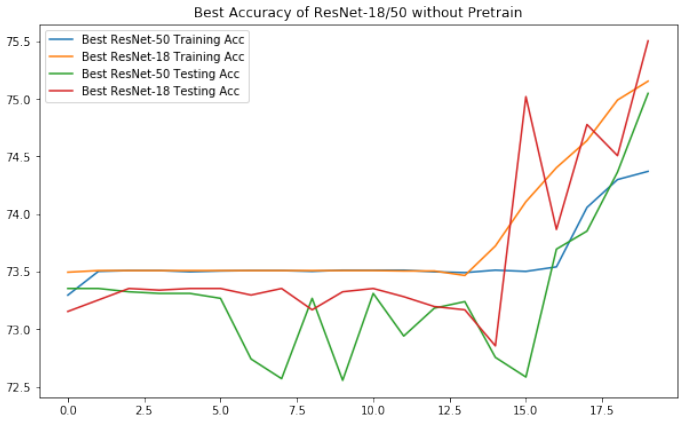


Figure 23: Best Accuracy Comparison without Pretrain

ResNet-18的結果：

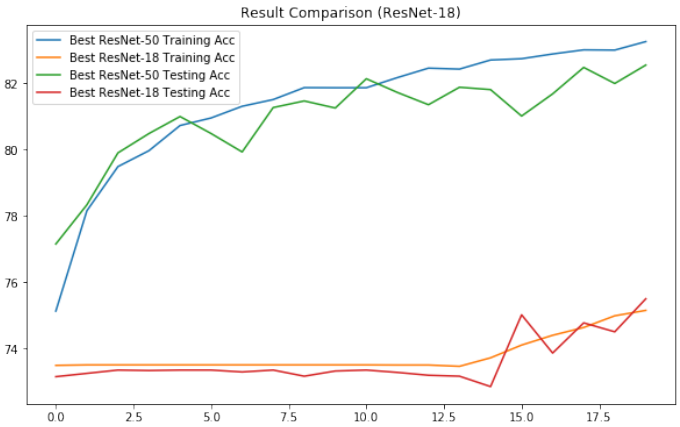


Figure 24: ResNet-18 Comparison Graph

ResNet-50的結果：

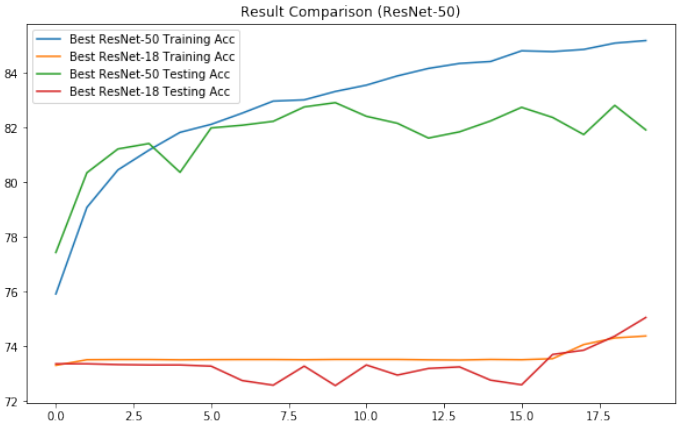
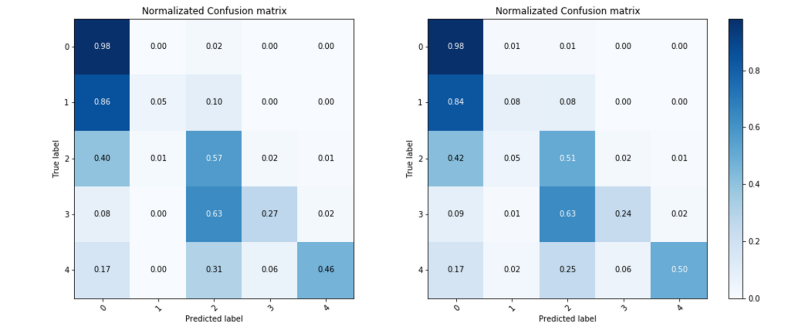
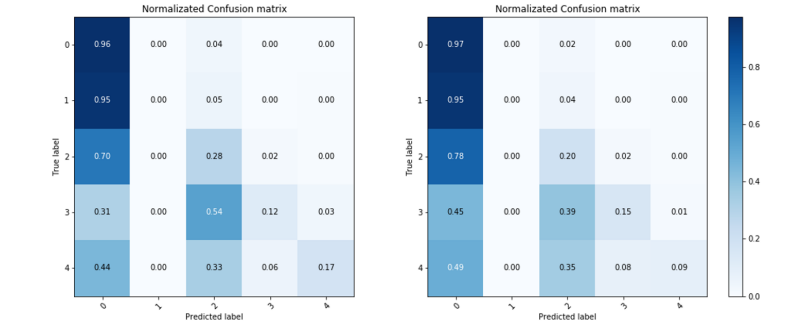


Figure 25: ResNet-50 Comparison Graph



(a) ResNet-18 Confusion Matrix (b) ResNet-50 Confusion Matrix

Figure 26: ResNet Confusion Matrix Comparison with Pretrain



(a) ResNet-18 Confusion Matrix (b) ResNet-50 Confusion Matrix

Figure 27: ResNet Confusion Matrix Comparison without Pretrain

上面幾張圖當中，只要是without pretrain畫出來的結果，10 epoch以內的testing accuracy幾乎都沒有什麼變化，一直到10幾epoch才開始慢慢上升。有可能是在10幾epoch之後ResNet突然學會該怎麼調整參數才有這樣的變化。如果提升epoch數量有可能還可以再提升accuracy。至於Confusion Matrix幾乎都predict 0的情況，我會在下一個章節討論。

5 Discussion

5.1 Data Normalization

剛開始在train的時候沒有注意到可以先對training data normalize，發現幾乎都要快10個epoch才會將test accuracy提升到82%。自從我在data augmentation加入normalization之後，幾乎只要5個epoch就可以上升到82%。

但也不一定完全是因為normalization，因為其實ResNet架構裡面也有BatchNormalization可以幫我做Normalize。只不過我的batch size也才16，所以有先將data normalize應該還是會比較好。

5.2 Confusion Matrix Problem

Figure 26和Figure 27都非常容易預測出0這個結果。一開始還以為是我的程式寫錯，但是仔細觀察training data之後才發現原來有73%的data label都是0，而testing data中也有73%的label是0，所以他當然很容易會把資料誤認成class 0。

```
label_count = np.zeros(5)
for data in tqdm(train_loader):
    data = data[1].numpy()
    for i in range(5):
        label_count[i] += len(data[data==i])
print(label_count)
```

```
[20655.  1955.  4210.   698.   581.]
```

(a) Train Label Count

```
test_label_count = np.zeros(5)
for data in tqdm(test_loader):
    data = data[1].numpy()
    for i in range(5):
        test_label_count[i] += len(data[data==i])
print(test_label_count)
```

```
[5153.  488. 1082.  175.  127.]
```

(b) Test Label Count

Figure 28: Code Snippet for Label Count