Butterfly & Moth Classification
Lab Report # 2

By
312581020
許瀚丰

Deep Learning
Spring 2024
Date Submitted: April 10, 2024

Contents

Listings

1 Introduction

1.1 Problem Overview

本次實驗是實作兩個經典的影像辨識架構，分別為VGG19 與ResNet50，並將其訓練於有100個Class的Butterfly & Moths Dataset上。而在本次實驗中，我也另外嘗試比較兩個模型分別使用Pretrained Weight與不使用Pretrained Weight在訓練上是否有何差異。

2 Implementation Details

2.1 The details of your model

2.1.1 VGG19

由於VGG19的每個block都是由好幾個Convolution加上ReLU後接起來，並在每個Block的最後加上Maxpooling，因此首先我將Convolution與ReLU的組合先建立出來，並在其中加上了Batch Normalization以幫助訓練，如程式碼1所示。

```
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=3, stride=1, padding=1):
        super(ConvBlock, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
        )
    def forward(self, x):
        return self.block(x)
```

Listing 1: Convolution Block

整體VGG19的模型是由五個數量分別為2, 2, 4, 4, 4 的ConvBlock所組合起來的Block，並在每個Block後皆會接上一個Maxpooling，最後再由三層fc作為Classifier來輸出結果。因此我在實作上透過build_layer這個function來建立了block1至block5，由於第一個Conv的in_channels是由上一組block而來，因此只有第一個ConvBlock需另外

處理，而其他的則使用迴圈建立即可，之後再加上Maxpooling。而最後只需要透過將輸出結果使用flatten拉平，並經過三個Linear輸出結果即可，如程式碼2所示。

```
class VGG19(nn.Module):
    def __init__(self, num_class=100):
        super(VGG19, self).__init__()
        self.block1 = self.build_layer(3, 64, 2)
        self.block2 = self.build_layer(64, 128, 2)
        self.block3 = self.build_layer(128, 256, 4)
        self.block4 = self.build_layer(256, 512, 4)
        self.block5 = self.build_layer(512, 512, 4)
        self.flatten = nn.Flatten()
        self.fc = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_class),
        )
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
                nn.init.kaiming_normal_(m.weight, mode="fan_out", nonlinearity="relu")
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)

    def build_layer(self, in_channels, out_channels, num_blocks):
        layer = [ConvBlock(in_channels, out_channels)]
        for _ in range(1, num_blocks):
            layer.append(ConvBlock(out_channels, out_channels))
        layer.append(nn.MaxPool2d(kernel_size=2, stride=2))
        return nn.Sequential(*layer)

    def forward(self, x):
        h = self.block1(x)
        h = self.block2(h)
        h = self.block3(h)
        h = self.block4(h)
        h = self.block5(h)
        h = self.flatten(h)
        h = self.fc(h)
        return h
```

Listing 2: VGG19

2.1.2 ResNet50

在ResNet50中，最重要的就是中間連續四層的Block，其內部分別為不同數量的Bottleneck Block所組合而成，其作法是讓輸入與輸出的都使用1×1的Conv，而中間的Conv的維度會比輸入與輸出還小，因此相比於原本的Residual Bock就可以在增加層數的同時降低運算量。在實作上，由於每個block的輸入是由上層的block而來，因此就需要注意在做skip connection時確認輸入的維度與輸出的維度是否相同，若不相同則需要先使用一個1×1 Conv將其升維，之後就與上面所提到的Bottleneck Block的輸出加總後回傳即可，如程式碼3所示。

```python
class BottleneckBlock(nn.Module):
    def __init__(self, in_channels, out_channels, expansion=4, down=False):
        super(BottleneckBlock, self).__init__()
        if in_channels != (out_channels * expansion):
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels * expansion, kernel_size=1, stride=2 if down else 1, padding=0, bias=False),
                nn.BatchNorm2d(out_channels * expansion)
            )
        else:
            self.shortcut = nn.Identity()
        self.block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=1, padding=0, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=2 if down else 1, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
            nn.Conv2d(out_channels, out_channels * expansion, kernel_size=1, stride=1, padding=0, bias=False),
            nn.BatchNorm2d(out_channels * expansion)
        )
        self.ReLU = nn.ReLU()
    def forward(self, x):
        return self.ReLU(self.block(x) + self.shortcut(x))
```

Listing 3: Bottleneck

而整體的ResNet50大致上可以分成三個部分，分別為一開始的init conv，接著中間由連續四組由不同數量的Bottleneck所組合出來的block連接，最後的一層則是用fc作為Classifier，一開始我直接使用nn.Sequential來建立conv1，而conv2至conv5與VGG19的實作類似，使用build_layer來特別處理第一層，需要注意的是其他層數都是使用out_channels * 4來作為in_channels的大小，因此在BottleneckBock function中的shortcut就不需要額外升維，如程式碼4。

```python
class ResNet50(nn.Module):
    def __init__(self, num_class=100):
        super(ResNet50, self).__init__()
        self.expansion = 4
        self.channels = [64, 128, 256, 512]
        self.num_blocks = [3, 4, 6, 3]
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(),
        )
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.conv2 = self.build_layer(self.channels[0], self.channels[0] , self.num_blocks[0], False)
        self.conv3 = self.build_layer(self.channels[0] * self.expansion, self.channels[1], self.num_blocks[1], True)
        self.conv4 = self.build_layer(self.channels[1] * self.expansion, self.channels[2], self.num_blocks[2], True)
        self.conv5 = self.build_layer(self.channels[2] * self.expansion, self.channels[3], self.num_blocks[3], True)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.flatten = nn.Flatten()
        self.fc = nn.Linear(self.channels[3] * self.expansion, num_class)
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
                nn.init.kaiming_normal_(m.weight, mode="fan_out", nonlinearity="relu")
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)

    def build_layer(self, in_channels, out_channels, num_blocks, down):
        layer = [BottleneckBlock(in_channels, out_channels, self.expansion, down)]
        for _ in range(1, num_blocks):
            layer.append(BottleneckBlock(out_channels * self.expansion, out_channels, self.expansion))
        return nn.Sequential(*layer)
```

```python

    def forward(self, x):
        x = self.conv1(x)
        x = self.maxpool(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.conv5(x)
        x = self.avgpool(x)
        x = self.flatten(x)
        x = self.fc(x)
        return x
```

Listing 4: ResNet50

2.2 The details of your Dataloader

2.2.1 Dataset

在資料的處理上，首先我使用了PIL所提供的Image來開啓圖片檔案，並透過PyTorch所提供的transform來做data Augmentation（實作方式會在下個section提到），如程式碼5所示。

```python
class ButterflyMothLoader(data.Dataset):
    def __init__(self, root, mode):
        self.root = root
        self.img_name, self.label = getData(mode)
        self.mode = mode
        print("> Found %d images..." % (len(self.img_name)))
    def __len__(self):
        return len(self.img_name)
    def __getitem__(self, index):
        img = Image.open(self.root + self.img_name[index])
        label = self.label[index]
        if self.mode == "train":
            img = train_transform(img)
        else:
            img = transform(img)

        return img, label
```

Listing 5: Dataset

2.2.2 Dataloader

對於資料的載入，我則是在main.py中使用PyTorch所提供的DataLoader，並設定Batch Size為128來進行訓練，如程式碼6所示。

```
from dataloader import ButterflyMothLoader
from torch.utils.data import DataLoader
train_data = ButterflyMothLoader(root="dataset/", mode="train")
train_loader = DataLoader(train_data, batch_size=128, shuffle=True)
valid_data= ButterflyMothLoader(root="dataset/", mode="val")
val_loader = DataLoader(valid_data, batch_size=128, shuffle=False)
```

Listing 6: Dataloader

3 Data Preprocessing

3.1 How you preprocessed your data?

在Data Preprocessing的部分，我使用PyTorch所提供的transform來實作，並在Training時對每張照片做以下操作：

- Resize成224×244
- 隨機左右翻轉
- 隨機上下翻轉
- 隨機旋轉$\pm$15度
- 隨機調整畫面的HSI
- 隨機擷取畫面一部分並Resize成224×244
- 依照ImageNet 影像的統計結果來對資料做Normalization

```
from torchvision import transforms
train_transform = transforms.Compose([
                transforms.Resize((224, 224)),
                transforms.RandomHorizontalFlip(),
```

```
                transforms.RandomVerticalFlip(),
                transforms.RandomRotation(degrees=15),
                transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2,
    hue=0.2),
                transforms.RandomResizedCrop(size=(224, 224), scale=(0.8, 1.0)),
                transforms.ToTensor(),
                transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
    0.225])
            ])
transform = transforms.Compose([
                transforms.Resize((224, 224)),
                transforms.ToTensor(),
                transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
    0.225])
            ])
```

Listing 7: Data Preprocessing

3.2 What makes your method special?

我的作法是參考助教所給的Hints，其中我認為較為特別的是我加上了ColorJitter與參考了ImageNet所統計出來的mean與std來做Normalization。前者可以在不改變畫面內容物位置的情況下透過調整畫面的HSI來增加資料的多樣性，而後者與直接將資料Normalize成-1到1的方法更能符合實際影像上的結果，期望因此能提升模型的泛化能力。

4 Experimental results

4.1 Hyperparameter settings

以下為本次實驗的超參數設定：

- **Batch Size**: 128
- **Loss Function**: CrossEntropyLoss
- **Optimizer**: Adam
- **Epoch**: 200
- **Learning Rate**: $1 * 10^{-3}$

4.2 The highest testing accuracy

實驗結果如表1所示，Test Acc為預測結果機率最大的class是正確的class的比例，Top3 Acc則是預測結果前三大的class內有正確的class的比例。從結果可以觀察到兩個模型的預測準確率皆有88%以上，而ResNet50不論是在Test Acc與Top 3 Acc皆比VGG19有更好的結果。

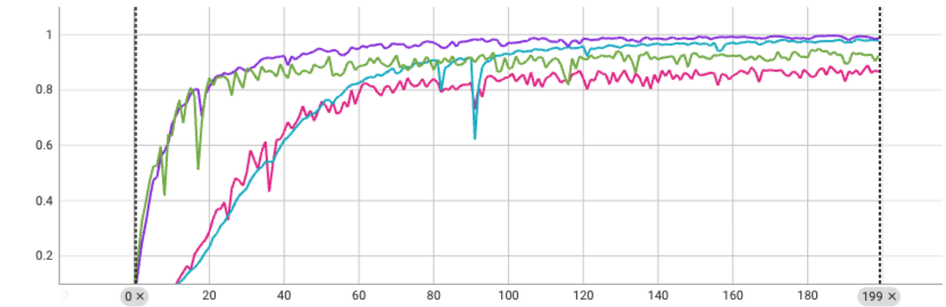
VGG19 Accuracy	ResNet50 Accuracy
> Found 500 images... Testing on VGG19 Test Loss: 0.7242, Test Acc: 89.0000%, Top3 Acc: 96.2000%	> Found 500 images... Testing on ResNet50 Test Loss: 0.2641, Test Acc: 95.4000%, Top3 Acc: 99.2000%

Table 1: Testing Accuracy

4.3 Comparison figures

訓練結果的Comparison figure如表2所示，兩個模型的訓練超參數皆相同，可以觀察到ResNet50相比於VGG19在前期更容易提升。另外我們也可以觀察到，兩者訓練到最後Training的Accuracy雖然只相差1%左右，然而在Validation上卻差了接近6%，可見ResNet50在此任務上的泛化能力是更好的。

Accuracy



Run ↑	Min	Max	Start Value	End Value	△Value	△%	Start Step	End Step
ResNet50/Acc_train	0.0507	0.9963	0.0507	0.9863	↑0.9356	↑1847%	0	199
ResNet50/Acc_valid	0.102	0.95	0.102	0.924	↑0.822	↑806%	0	199
VGG19/Acc_train	0.0112	0.9823	0.0112	0.9781	↑0.9669	↑8636%	0	199
VGG19/Acc_valid	0.012	0.89	0.012	0.866	↑0.854	↑7117%	0	199

Table 2: Comparison figure

5 Discussion

5.1 Train with Petrained model

在本次實驗中，我也嘗試使用了Pretrained的VGG19與ResNet50來做訓練，兩者分別都需要將最後的Classifier做一些簡單的修改，使其符合本次實驗的class個數，Pretrained VGG19與ResNet50的實作分別為程式碼8與9。

```
import torch
import torch.nn as nn
from torchvision.models import vgg19_bn, VGG19_BN_Weights

class VGG19_pretrained(nn.Module):
    def __init__(self, num_class=100):
        super(VGG19_pretrained, self).__init__()
        self.vgg = vgg19_bn(weights=VGG19_BN_Weights.IMAGENET1K_V1)
        self.vgg.classifier[-1] = nn.Linear(4096, num_class)

    def forward(self, x):
        x = self.vgg(x)
        return x
```

Listing 8: Pretrained VGG

```python
import torch
import torch.nn as nn
from torchvision.models import resnet50, ResNet50_Weights

class ResNet50_pretrained(nn.Module):
    def __init__(self, num_class=100):
        super(ResNet50_pretrained, self).__init__()
        self.ResNet50 = resnet50(weights=ResNet50_Weights.IMAGENET1K_V1)
        self.ResNet50.fc = nn.Linear(2048, num_class)

    def forward(self, x):
        x = self.ResNet50(x)
        return x
```

Listing 9: Pretrained ResNet50

5.2 The highest testing accuracy

實驗結果如表3所示，可以看到兩者的Accuracy相比於從隨機的權重開始訓的模型Accuracy皆提升了不少，而Pretrained VGG19與Pretrained ResNet50兩者最終的Testing Accuracy差異不大。

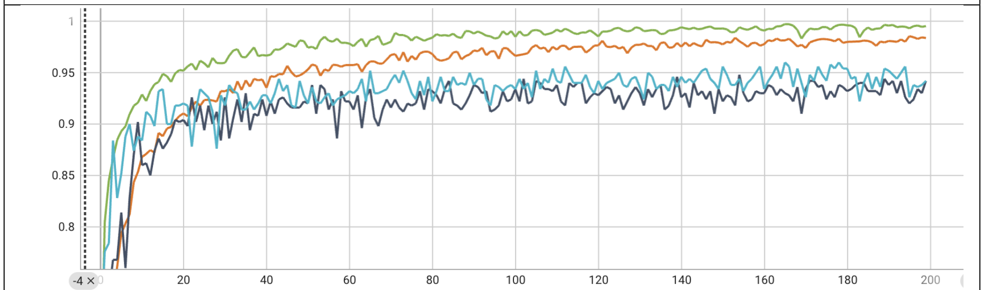
Pretrained VGG19 Accuracy	Pretrained ResNet50 Accuracy
> Found 500 images... Testing on VGG19_pretrained Test Loss: 0.3000, Test Acc: 95.8000%, Top3 Acc: 98.6000%	> Found 500 images... Testing on ResNet50_pretrained Test Loss: 0.1910, Test Acc: 96.2000%, Top3 Acc: 99.0000%

Table 3: Testing Accuracy on Pretrained Model

5.3 Comparison figures

訓練結果的Comparison figure如表4所示，兩個模型的訓練超參數皆相同，從圖中可以觀察到Pretrained ResNet50的Training Accuracy從頭到尾都比Pretrained VGG19好上一些些，而兩者在Validation Accuracy則相差不多。

Accuracy



Run	Min	Max	Start Value	End Value	ΔValue	Δ%	Start Step	End Step
ResNet50_pretrained/Acc_train	0.6162	0.9971	0.6162	0.9952	↑0.379	↑61%	0	199
ResNet50_pretrained/Acc_valid	0.684	0.96	0.684	0.942	↑0.258	↑38%	0	199
VGG19_pretrained/Acc_train	0.2403	0.9855	0.2403	0.9836	↑0.7434	↑309%	0	199
VGG19_pretrained/Acc_valid	0.38	0.948	0.38	0.942	↑0.562	↑148%	0	199

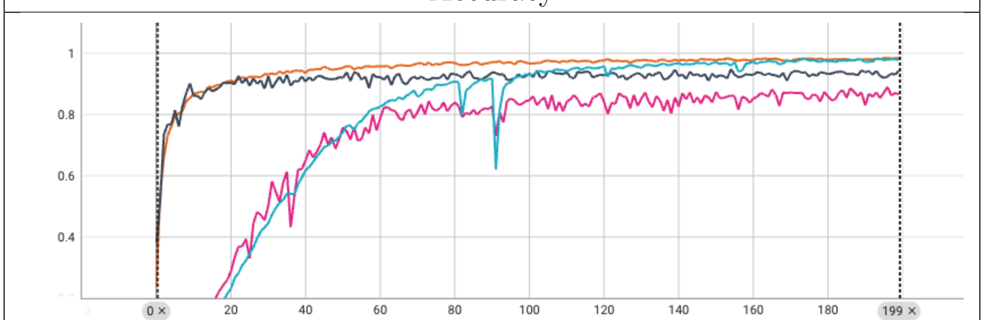
Table 4: Comparison figure on Pretrained Model

5.4 Compare with non-pretrained model

5.4.1 VGG19

實驗結果的如表5所示，可以看到在VGG19中是否有使用Pretrained Weights在前期的差異最為明顯，使用Pretrained Weights的Model收斂快了許多，且在最終的Validation Accuracy上的結果也優於未使用Pretrained Weights 的Model，可見使用Pretrained Weights也能使模型泛化能力更好。

Accuracy

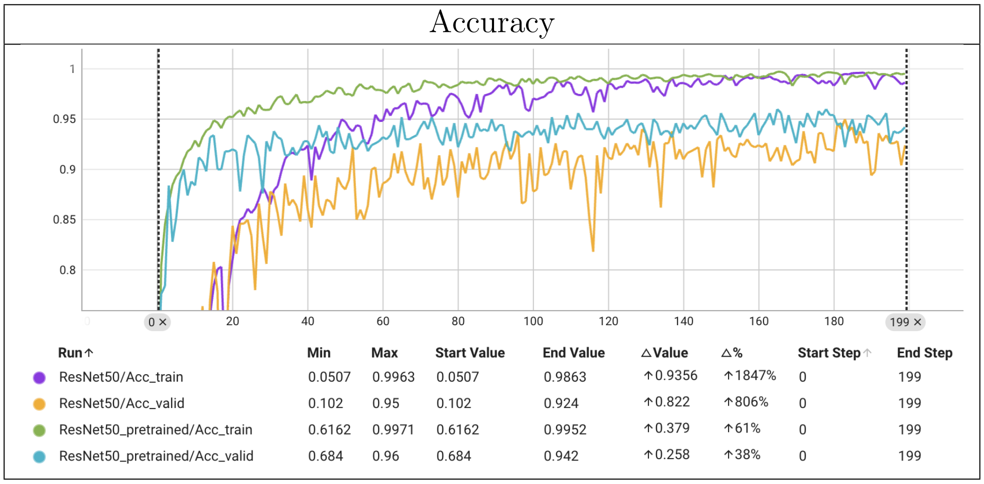


Run	Min	Max	Start Value	End Value	ΔValue	Δ%	Start Step	End Step
VGG19/Acc_train	0.0112	0.9823	0.0112	0.9781	↑0.9669	↑8636%	0	199
VGG19/Acc_valid	0.012	0.89	0.012	0.866	↑0.854	↑7117%	0	199
VGG19_pretrained/Acc_train	0.2403	0.9855	0.2403	0.9836	↑0.7434	↑309%	0	199
VGG19_pretrained/Acc_valid	0.38	0.948	0.38	0.942	↑0.562	↑148%	0	199

Table 5: Comparison on VGG19

5.4.2 ResNet50

實驗結果的如表6所示，相比於VGG19，ResNet50在是否有pretrained上的結果差異較小，但仍能觀察出在前期的收斂較快與後期模型的Validation Accuracy較高的趨勢。



Run	Min	Max	Start Value	End Value	△Value	△%	Start Step	End Step
ResNet50/Acc_train	0.0507	0.9963	0.0507	0.9863	↑0.9356	↑1847%	0	199
ResNet50/Acc_valid	0.102	0.95	0.102	0.924	↑0.822	↑806%	0	199
ResNet50_pretrained/Acc_train	0.6162	0.9971	0.6162	0.9952	↑0.379	↑61%	0	199
ResNet50_pretrained/Acc_valid	0.684	0.96	0.684	0.942	↑0.258	↑38%	0	199

Table 6: Comparison on ResNet50