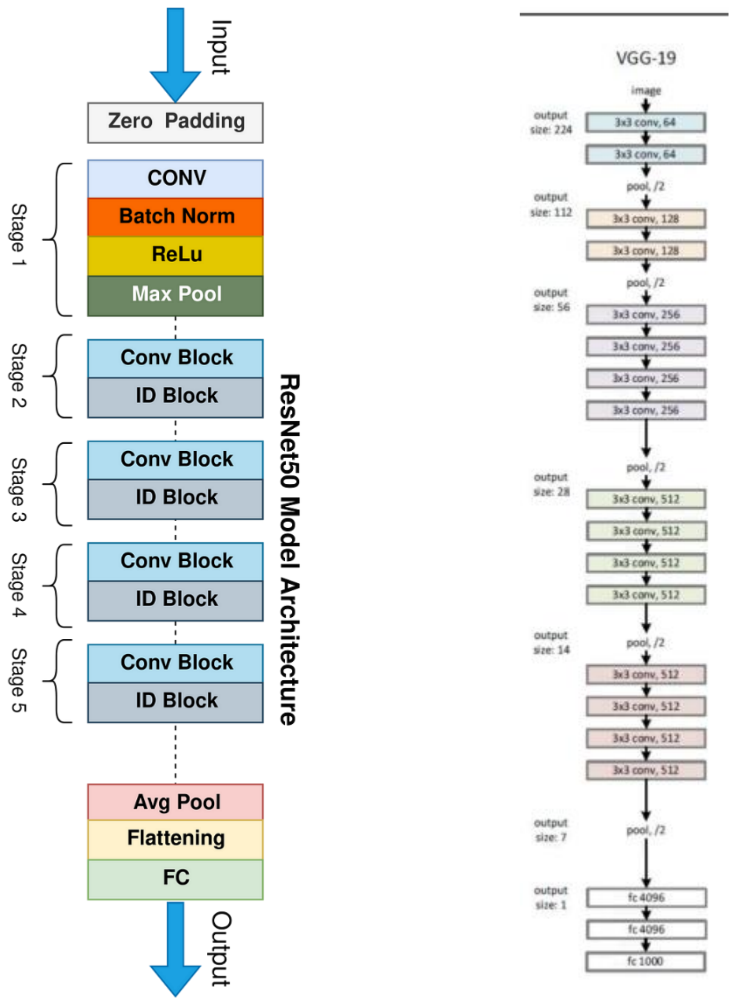
NYCU DL

Lab2 – Butterfly & Moth Classification

312554006 羅名志

1. Introduction

Lab 2 involves implementing an image classification task using two renowned architectures: VGG19 and ResNet50, as illustrated in the following figure. By manually crafting the details of both models, we aim to have a better understanding of the mechanisms of convolutional neural networks and acquire valuable techniques for processing image data.



[1] Architectures of ResNet50 and VGG19 we need to implement in this lab.

2. Implementation Details

A. The details of your model (VGG19, ResNet50)

a. VGG19

```python
def __init__(self, in_channels = 3, num_classes=100):
    self.features = nn.Sequential(
        nn.Conv2d(in_channels, 64, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(64, 64, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),

        nn.Conv2d(64, 128, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(128, 128, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),

        nn.Conv2d(128, 256, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(256, 256, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(256, 256, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(256, 256, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),

        nn.Conv2d(256, 512, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(512, 512, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(512, 512, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(512, 512, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Conv2d(512, 512, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(512, 512, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(512, 512, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(512, 512, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),
    )
```

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Based on the architecture of VGG19 (right graph), I construct the model to completely adhere to each layer's settings. Furthermore, the layers of the output classifier (below graph) also follow the official VGG19 settings.

```python
self.avgpool = nn.AdaptiveAvgPool2d((7, 7))

self.classifier = nn.Sequential(
    nn.Linear(512 * 7 * 7, 4096),
    nn.ReLU(inplace=True),
    nn.Dropout(),
    nn.Linear(4096, 4096),
    nn.ReLU(inplace=True),
    nn.Dropout(),
    nn.Linear(4096, num_classes),
)
```

b. ResNet50

The architecture of ResNet50 ([2] below graph) is composed of 50 layers, which follow a specific repetitive pattern

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

The overall model structure is show below, following the specifications of ResNet50.

```python
def __init__(self, layer_list, in_channels = 3, out_class = 100):
    super().__init__()
    # initial block input channels: 64, 128, 256, 512 (with expansion = 4)
    self.in_c = 64
    # in layer0, first two layers: 7x7, 64, stride = 2 and 3x3 max pool, stride = 2
    self.layer0 = nn.Sequential(
        nn.Conv2d(in_channels, 64, kernel_size = 7, stride = 2, padding = 3, bias = False),
        nn.BatchNorm2d(64),
        nn.ReLU(inplace = True),
        nn.MaxPool2d(kernel_size = 3, stride = 2, padding = 1)
    )

    # following layers = setting of ResNet 50 with each layer num: 3, 4, 6, 3
    self.layer1 = self.make_layer(out_c = 64, layer_num = layer_list[0], stride = 1)
    self.layer2 = self.make_layer(out_c = 128, layer_num = layer_list[1], stride = 2)
    self.layer3 = self.make_layer(out_c = 256, layer_num = layer_list[2], stride = 2)
    self.layer4 = self.make_layer(out_c = 512, layer_num = layer_list[3], stride = 2)

    # for calssification task
    self.classifier = nn.Sequential(
        nn.AdaptiveAvgPool2d((1,1)),
        nn.Flatten(),
        nn.Linear(2048, out_class)
        #nn.Linear(2048, 512),
        #nn.ReLU(inplace = True),
        #nn.Dropout(0.4),
        #nn.Linear(512, out_class)
    )
```

Because the basic block is repeated with only the output size varying, I construct the basic block (class Bottleneck) and utilize the make_layer function to efficiently build the overall architecture.

* The Bottleneck

```
def __init__(self, in_c, out_c, downsample, stride = 1):
    super().__init__()

    self.conv1 = nn.Conv2d(in_c, out_c, kernel_size = 1, stride = 1, padding = 0)
    self.batch_norm1 = nn.BatchNorm2d(out_c)
    self.conv2 = nn.Conv2d(out_c, out_c, kernel_size = 3, stride = stride, padding = 1)
    self.batch_norm2 = nn.BatchNorm2d(out_c)
    self.conv3 = nn.Conv2d(out_c, out_c*4, kernel_size = 1, stride = 1, padding = 0)
    self.batch_norm3 = nn.BatchNorm2d(out_c*4)

    self.downsample = downsample
    self.relu = nn.ReLU(inplace = True)
```

To be more specific, the make_layer function is responsible for creating various components of layers based on different output sizes. Additionally, when the input size differs from the output size, the addition of the residual part may result in mismatched sizes. To address this, the downsample function is utilized to ensure that both sizes remain matched.

* The make_layer function

```
def make_layer(self, out_c, layer_num, stride):
    # every first layer need downsampling to match the size b/w x and identity(residual) size
    downsample = nn.Sequential(
        nn.Conv2d(self.in_c, out_c*4, kernel_size = 1, stride = stride),
        nn.BatchNorm2d(out_c*4)
    )
    layers = []
    layers.append(Bottleneck(self.in_c, out_c, downsample, stride))
    self.in_c = out_c*4
    for i in range(layer_num-1):
        layers.append(Bottleneck(self.in_c, out_c, None))
    return nn.Sequential(*layers)
```

B. The details of your Dataloader

```python
def getData(mode):
    if mode == 'train':
        curr_path = os.getcwd()
        path = os.path.join(curr_path, 'dataset/train.csv')
        print(path)
        df = pd.read_csv(path)
        path = df['filepaths'].tolist()
        label = df['label_id'].tolist()
        return path, label
    elif mode == 'valid':
        curr_path = os.getcwd()
        path = os.path.join(curr_path, 'dataset/valid.csv')
        print(path)
        df = pd.read_csv(path)
        path = df['filepaths'].tolist()
        label = df['label_id'].tolist()
        return path, label
    else:
        curr_path = os.getcwd()
        path = os.path.join(curr_path, 'dataset/test.csv')
        print(path)
        df = pd.read_csv(path)
        path = df['filepaths'].tolist()
        label = df['label_id'].tolist()
        return path, label
```

First define the location of each image file under different modes, and return each file path and its label.

```python
img_path = self.root + '/dataset/' + self.img_name[index]
img = Image.open(img_path)

# transform the data
if self.mode == 'train':
    transform = transforms.Compose([
        transforms.RandomRotation(10),  # Randomly rotate
        transforms.RandomHorizontalFlip(),  # Randomly fli
        transforms.RandomVerticalFlip(),  # Randomly flip
        # transforms.RandomResizedCrop(224),  # Randomly c
        # transforms.ColorJitter(brightness=0.2, contrast=
        # transforms.RandomApply([transforms.GaussianBlur(
        transforms.ToTensor(),
    ])

else:
    transform = transforms.ToTensor()
img = transform(img)

label = torch.tensor(self.label[index], dtype=torch.long)

return img, label
```

Get the data after data augmentation and transform to tensor format.

3. Data Preprocessing

A. How you preprocessed your data?

For the training data, I experimented with several data augmentation methods. However, implementing all these methods simultaneously can lead to slower convergence of the model, which is expected. Therefore, I only applied rotation and flipping during the preprocessing step. For the testing and validation sets, the only necessary transformation is to convert them into tensor format.

```
# transform the data
if self.mode == 'train':
    transform = transforms.Compose([
        transforms.RandomRotation(10),  # Randomly rotate the image by 10 degrees
        transforms.RandomHorizontalFlip(),  # Randomly flip the image horizontally
        transforms.RandomVerticalFlip(),  # Randomly flip the image vertically
        # transforms.RandomResizedCrop(224),  # Randomly crop a portion of the image and resize it to 224x224
        # transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),  # Randomly adjust brightness, contrast, saturation, and hue
        # transforms.RandomApply([transforms.GaussianBlur(kernel_size=3)], p=0.5),  # Randomly apply Gaussian blur with a probability
        transforms.ToTensor(),
    ])

else:
    transform = transforms.ToTensor()
img = transform(img)
```

B. What makes your method special?

I think my preprocessing approach isn't particularly special. The only distinguishing factor is my integration of various augmentation methods and experimentation with different combinations. This is quite different from the simplest case, where the input data is merely transformed into tensor format.

4. Experiment results

A. The highest testing accuracy

The highest testing accuracy in my model is VGG19 with learning rate=0.0001/epochs=60/batch_size=64. The result is displayed below.

```
Epoch [57/60], Train Loss: 0.0994, Training accuracy: 96.9350
Epoch [57/60], Validation Loss: 0.7599, Validation accuracy: 0.8340
Epoch [58/60], Train Loss: 0.0975, Training accuracy: 97.0859
Epoch [58/60], Validation Loss: 0.9675, Validation accuracy: 0.8340
Epoch [59/60], Train Loss: 0.0948, Training accuracy: 97.1336
Epoch [59/60], Validation Loss: 1.0197, Validation accuracy: 0.8260
Epoch [60/60], Train Loss: 0.0998, Training accuracy: 96.9430
Epoch [60/60], Validation Loss: 0.8213, Validation accuracy: 0.8320
Test Loss: 0.6276, Test Accuracy: 0.8800
```

```
model: VGG19
Train Loss: 0.0401, Train Accuracy: 98.9281%
Test Loss: 0.6276, Test Accuracy: 88.0000%
```

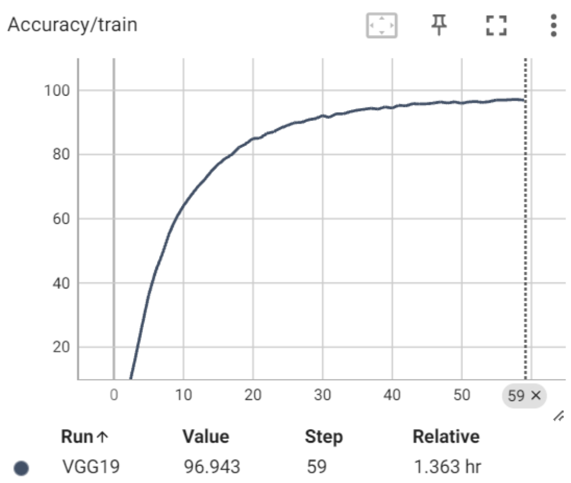
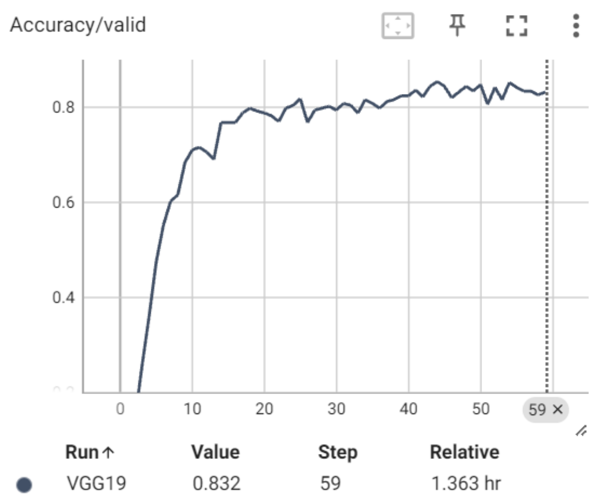
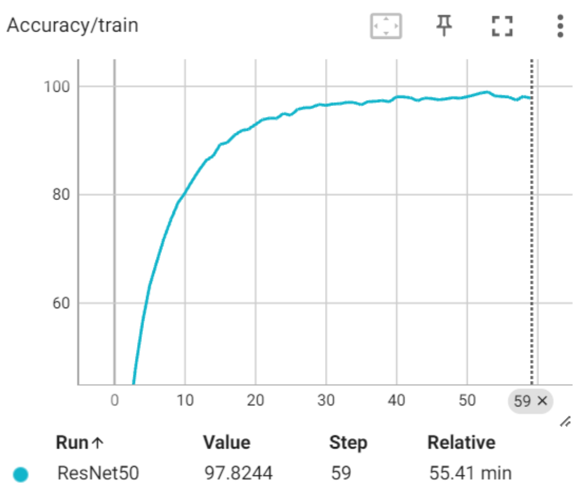
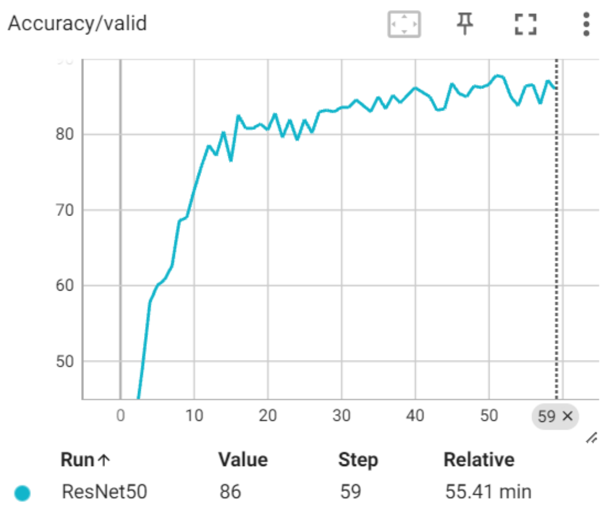
Whie, the result of ResNet50 is also comparable with VGG19 under same hyperparameters setting. The result is shown below:

```
Epoch [57/60], Train Loss: 0.0643, Training accuracy: 98.0149%
Epoch [57/60], Validation Loss: 0.6833, Validation accuracy: 86.6000%
Epoch [58/60], Train Loss: 0.0793, Training accuracy: 97.4353%
Epoch [58/60], Validation Loss: 0.7404, Validation accuracy: 84.0000%
Epoch [59/60], Train Loss: 0.0573, Training accuracy: 98.1340%
Epoch [59/60], Validation Loss: 0.6773, Validation accuracy: 87.2000%
Epoch [60/60], Train Loss: 0.0641, Training accuracy: 97.8244%
Epoch [60/60], Validation Loss: 0.7397, Validation accuracy: 86.0000%
Test Loss: 0.7065, Test Accuracy: 84.0000
```

```
model: ResNet50
Train Loss: 0.2542, Train Accuracy: 92.7267%
Test Loss: 0.7065, Test Accuracy: 84.0000%
```

In summary, the VGG19 model achieved the highest testing accuracy of 88%, whereas the ResNet50 model reached 84%. It's worth noting that these results are based on a sample, and upon multiple runs, both models consistently yield similar accuracies.

B. Comparison figures

<table>
<tr><td></td><td>Training</td><td>Validation</td></tr>
<tr><td>VGG19</td><td>
</td><td>
</td></tr>
<tr><td>ResNet50</td><td>
</td><td>
</td></tr>
</table>

1. Both VGG19 and ResNet50 begin to converge after 20 epochs.
2. While the training and validation accuracies of ResNet50 are higher than those of VGG19, the testing results in this case are slightly worse than those of VGG19.

5. Discussion

1. During the experiment, I observed that ResNet50 is much easier to train, as indicated by its training/validation accuracy and convergence speed. However, the testing results are similar to those of VGG19. This could be due to ResNet50 having more layers than VGG19, potentially leading to unstable results in the test cases.
2. I discovered that increasing the use of data augmentation does not necessarily result in higher accuracy. When the combinations of augmentations become too complex, the model may struggle to converge and learn useful patterns.
3. Since the goal of this lab is not to achieve 100% accuracy, I still obtained a relatively worse result, around 88%. The design of the network and hyperparameters may affect performance, and I am still investigating the reasons behind this.