DLP Lab2 Report

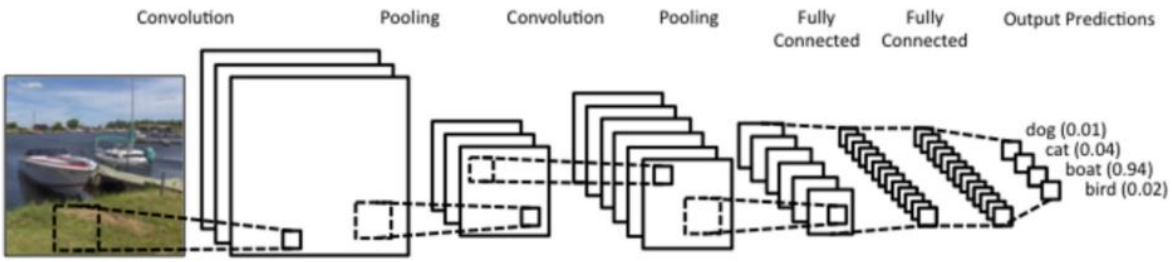
Butterfly and Moth Classification

Name: 丁祐承

ID: 0812212

1. Introduction

The main purpose of lab2 is to let us get to know some CNN based network and try to implement. There are two main networks for this lab to implement, the first one is VGG19 and the second one is ResNet50. After implement these two models, we need to train them on butterfly and moth dataset and compare the accuracy. For this lab, we need to know the whole procedure for training a model. Customize my own Dataloader, data preprocessing, training and evaluate.



CNN architecture

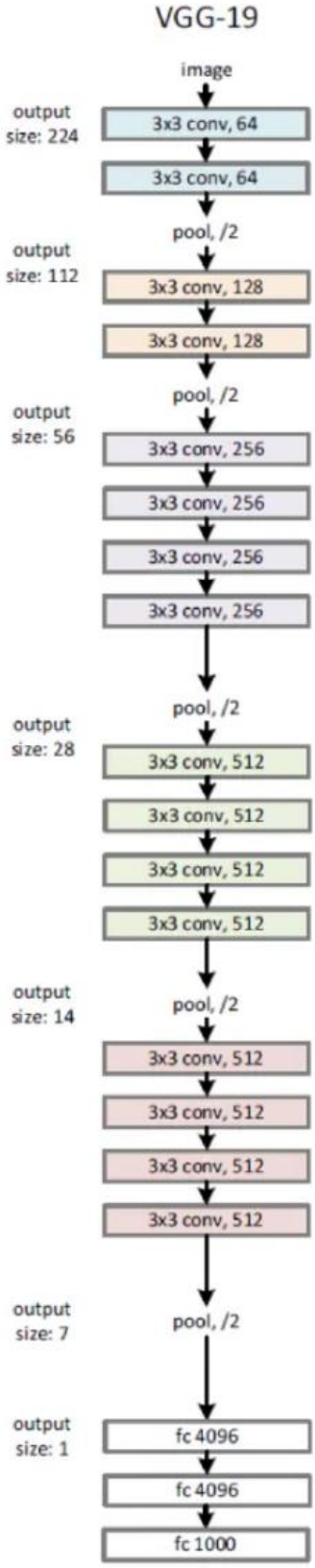
2. Implement Details

A. Model Details

VGG19:

VGG19, with its deep architecture, has been widely used in various image processing and computer vision tasks, including image classification, object detection, and more complex tasks such as image segmentation and face recognition. Due to its uniform architecture, it's also frequently used as a backbone for more

complex models and in feature extraction tasks where deep, hierarchical features
from images are useful.



Left side figure is the architecture of the VGG19. We can see that there are four main blocks with different dimensions and number of layers. For the first two blocks, their dimensions are 64 and 128 respectively with each two layers. For the rest three blocks, their dimensions are 256, 512, 512 respectively and with four layers of each. Each layer for VGG19 is a 3 by 3 kernel size convolutional layer. After each block, we add a pooling layer to downsample the model. After all the blocks, we flatten the model and map to the number of classes we want. For this lab, we map to 100 for butterfly and moth dataset.

```
# First block, output: 112
self.block1 = nn.Sequential(nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1),
                            nn.BatchNorm2d(64),
                            nn.ReLU(True),
                            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1),
                            nn.BatchNorm2d(64),
                            nn.ReLU(True),
                            nn.MaxPool2d(kernel_size=2, stride=2))
```

```
self.classifier = nn.Sequential(nn.Linear(in_features=512*7*7, out_features=4096, bias=False),
                                nn.ReLU(True),
                                nn.Dropout(p=0.5),
                                nn.Linear(in_features=4096, out_features=4096),
                                nn.ReLU(True),
                                nn.Dropout(p=0.5),
                                nn.Linear(in_features=4096, out_features=num_classes))
```

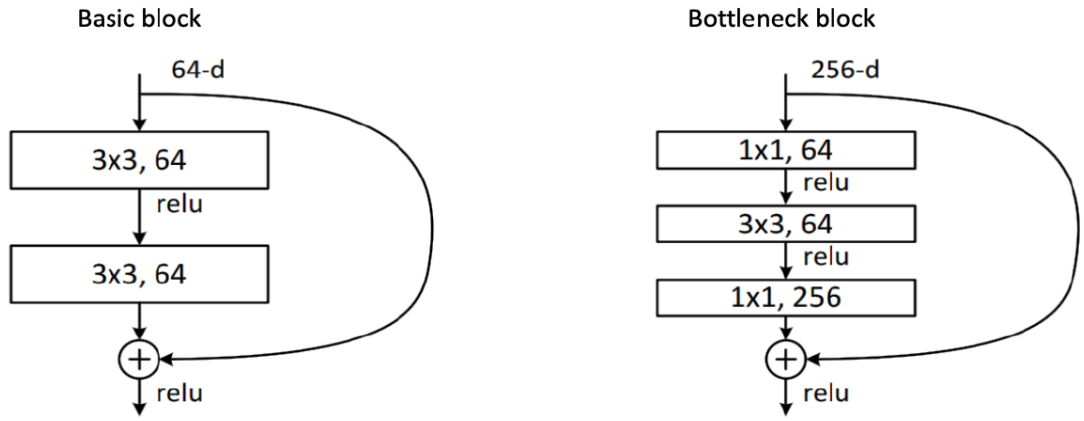
The first figure is the example of the block of VGG19 and the second figure is the final Fully Connected Layer for classifier.

ResNet50:

ResNet50 is a variant of ResNet (Residual Network) architecture, which is a type of convolutional neural network (CNN) that has been highly influential in the field of deep learning, especially in the context of image recognition and image classification tasks. The "50" in ResNet50 refers to the number of layers it contains, making it one of the several versions of ResNet, which also include ResNet18, ResNet34, ResNet101, and ResNet152, among others. The core innovation of ResNet, and by extension ResNet50, is the introduction of "residual blocks" that allow for the training of much deeper networks than were previously feasible.

For the residual block part, it is different from the ResNet18. ResNet34. Start from ResNet50 and more layer of ResNet, they use bottleneck as the residual block. The original residual block has two convolutional layer with each has 3 by 3 kernel size feature map. The bottleneck design has three convolutional layers. The first one and the last one have 1 by 1 kernel size feature map and the middle one has 3 by 3

kernel size feature map. The residual part is the same as the basic block. Original convolution based block aims to learn the F(x), where F meas the convolutional layer. The Residual block design now change the output into H(x) = F(x) + x, where F is the design of the residual block. By doing so, we can deal with the vanishing or exploiding gradients problems.




```python
def __init__(self, in_feature, config, connect, downsampling=False):
    super().__init__()
    self.in_feature = in_feature
    self.config = config
    self.connect = connect
    if downsampling:
        # To achieve downsampling, using stride 2
        self.block = nn.Sequential(nn.Conv2d(in_channels=in_feature, out_channels=config[0], kernel_size=1),
                                   nn.BatchNorm2d(config[0]),
                                   nn.ReLU(True),
                                   nn.Conv2d(in_channels=config[0], out_channels=config[1], kernel_size=3, stride=2, padding=1),
                                   nn.BatchNorm2d(config[1]),
                                   nn.ReLU(True),
                                   nn.Conv2d(in_channels=config[1], out_channels=config[2], kernel_size=1),
                                   nn.BatchNorm2d(config[2]))
    else:
        self.block = nn.Sequential(nn.Conv2d(in_channels=in_feature, out_channels=config[0], kernel_size=1),
                                   nn.BatchNorm2d(config[0]),
                                   nn.ReLU(True),
                                   nn.Conv2d(in_channels=config[0], out_channels=config[1], kernel_size=3, stride=1, padding=1),
                                   nn.BatchNorm2d(config[1]),
                                   nn.ReLU(True),
                                   nn.Conv2d(in_channels=config[1], out_channels=config[2], kernel_size=1),
                                   nn.BatchNorm2d(config[2]))
    self.relu = nn.ReLU()

def forward(self, x):
    # Residual connect
    residual = self.connect(x)
    x = self.block(x)
    x += residual
    x = self.relu(x)
    return x
```

Bottleneck Design

```python
class ResNet50(nn.Module):
    def __init__(self, num_classes=1000):
        super().__init__()
        # First part of the ResNet50 (conv1): 7*7, 64 stride 2
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=7, stride=2, padding=3)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu1 = nn.ReLU(True)
        # Second part of the ResNet50: 3*3 max pool, stride 2
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        # Conv2_x
        self.conv2 = self.generate_layer(num_layer=3, config=[64,64,256], in_feature=64, downsampling=False)
        # Conv3_x
        self.conv3 = self.generate_layer(num_layer=4, config=[128,128,512], in_feature=256, downsampling=True)
        # Conv4_x
        self.conv4 = self.generate_layer(num_layer=6, config=[256,256,1024], in_feature=512, downsampling=True)
        # Conv5_x
        self.conv5 = self.generate_layer(num_layer=3, config=[512,512,2048], in_feature=1024, downsampling=True)
        # Last part of the ResNet50: average pool, 1000-d fc, softmax
        self.avgpool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Linear(in_features=2048, out_features=num_classes)
```

Design of my ResNet50

B. DataLoader

```python
# Preparing dataset
print('Loading data')
train_data = ButterflyMothLoader('./dataset/', 'train')
valid_data = ButterflyMothLoader('./dataset/', 'valid')
test_data = ButterflyMothLoader('./dataset/', 'test')

train_loader = DataLoader(dataset=train_data, batch_size=24, shuffle=True)
valid_loader = DataLoader(dataset=valid_data)
test_loader = DataLoader(dataset=test_data)
```

```python
class ButterflyMothLoader(data.Dataset):
    def __init__(self, root, mode):
        """
        Args:
            mode : Indicate procedure status(training or testing)

            self.img_name (string list): String list that store all image names.
            self.label (int or float list): Numerical list that store all ground truth label values.
        """
        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 the size of dataset"""
        return len(self.img_name)
```

```python
class ButterflyMothLoader(data.Dataset):
    def __getitem__(self, index):
        """thing you should implement here"""

        """
           step1. Get the image path from 'self.img_name' and load it.
                  hint : path = root + self.img_name[index] + '.jpg'

           step2. Get the ground truth label from self.label

           step3. Transform the .jpg rgb images during the training phase, such as resizing, random flipping,
                  rotation, cropping, normalization etc. But at the beginning, I suggest you follow the hints.

                  In the testing phase, if you have a normalization process during the training phase, you only need
                  to normalize the data.

                  hints : Convert the pixel value to [0, 1]
                          Transpose the image shape from [H, W, C] to [C, H, W]

            step4. Return processed image and label
        """
        img_path = self.root + self.img_name[index]
        label = self.label[index]
        img = Image.open(img_path)
        transform1 = transforms.Compose([transforms.ToTensor(),
                                         transforms.RandomHorizontalFlip(p=0.7),
                                         transforms.RandomVerticalFlip(p=0.7),
                                         transforms.RandomRotation(30),
                                         transforms.Resize((224,224)),
                                         transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225])])
        transform2 = transforms.Compose([transforms.ToTensor(),
                                         transforms.Resize((224,224)),
                                         transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225])])
        if self.mode == 'train':
            img = transform1(img)
        else:
            img = transform2(img)

        return img, label
```

We defined a class called ButterflyMothLoader inherit from torch.utils.data.Data. This can let us customize our own dataset. For __init__ we initialize our constructor by giving it the dataset's root path and the mode for training, validating or testing. The __len__ function returns the size of the dataset. The __getitem__ function let us have access to the data in the dataset by returning the data and the label for it. After declaring the ButterflyMothLoader, we can use it as argument for DataLoader. DataLoader has three main parameters, the first one is the dataset, for this lab which is ButterflyMothLoader. The second parameter is the batch size and the third parameter is shuffle. After setting all of this, we can use it for model training.

3. Data Preprocessing

```
transform1 = transforms.Compose([transforms.ToTensor(),
                                 transforms.RandomHorizontalFlip(p=0.7),
                                 transforms.RandomVerticalFlip(p=0.7),
                                 transforms.RandomRotation(30),
                                 transforms.Resize((224,224)),
                                 transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225])])
transform2 = transforms.Compose([transforms.ToTensor(),
                                 transforms.Resize((224,224)),
                                 transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225])])
```

Data preprocessing and augmentation

A. How do you preprocessed your data?

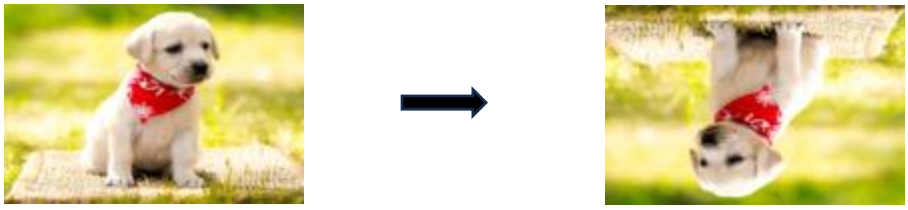
I use torchvision.transform to apply data preprocessing and data augmentation. For data preprocessing, I turned the image to tensor type data since the model's inputs need to be tensor by using "transforms.ToTensor()". By doing so, the image pixel turns from 0 ~ 255 to 0 ~ 1 and transpose the dimension of image from H * W * C into C * H * W, where C stands for channel, H stands for height and W stands for width. I also resize the image to (256*256) and normalize the image by using "transforms.Resize()" and "transforms.Normalize()" respectively. Resizing makes sure all the inputs have the same size and normalizing let the training be more easier and converge early. These three preprocessing parts all apply for the training dataset and the testing dataset. For data augmentation, I only apply on the training dataset. I did random vertical flip, horizontal flip and rotation. By doing so, the model can learn better compared to the original dataset.

B. What makes your method special.

I did not apply the cropping technique because I think if doing so, data might lose the original meaning. For example, the following figure shows a butterfly. If I accidentally crop the data into the next figure, which is the head of the butterfly, how can the model learn from this

and recognize it as a butterfly? I only do flip and rotation, I think these two techniques will not have big influence for model and will increase the diversity of the dataset. By doing so, we can increase the performance of the model.

Original data *Cropped data*



Example: Vertical Flip

4. Experimental results

A. The highest testing accuracy

```
-------------------------VGG19-------------------------------
VGG19       |   Train accuracy: 99.27%|  Test accuracy: 95.00%
-------------------------ResNet50----------------------------
ResNet50    |   Train accuracy: 87.99%|  Test accuracy: 87.20%
```

I trained 120 epoch for each model and VGG19 had converged already. While ResNet50 was not converged. The 120th epoch of VGG19, it had 99.32% of training accuracy and 94.20% of validating accuracy. For

ResNet50, it had 87.61% of training accuracy and 84.20% of validating accuracy.

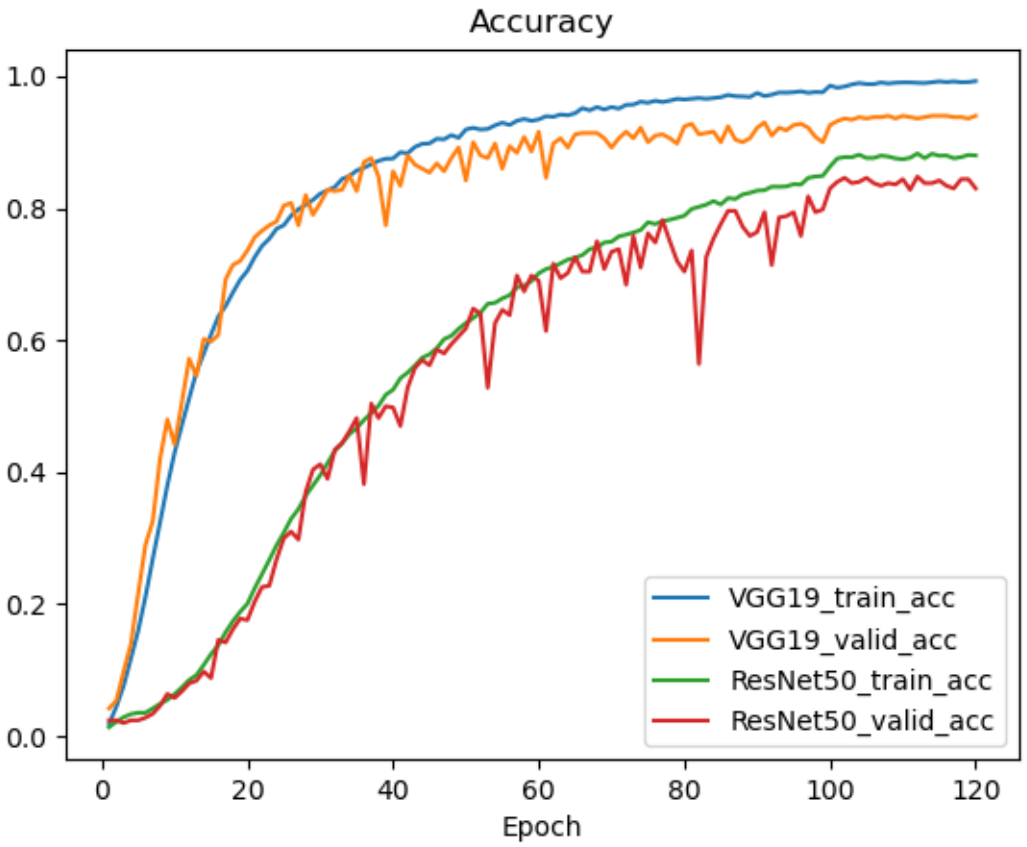
```
Epoch 120:
Train: 100%| | 525/525 [04:20<00:00,  2.01it/s]
Test: 100%| | 500/500 [00:04<00:00, 103.99it/s]
Training loss:   0.030179, Accuracy: 99.32%, Validation accuracy: 94.20%
Best accuracy: 94.20%. Model saved...
```

VGG19 training history

```
Epoch 120:
Train: 100%| | 263/263 [02:52<00:00,  1.52it/s]
Test: 100%| | 500/500 [00:07<00:00, 67.36it/s]
Training loss:   0.459353, Accuracy: 87.61%, Validation accuracy: 84.20%
```

ResNet50 training history

B. Comparison figures

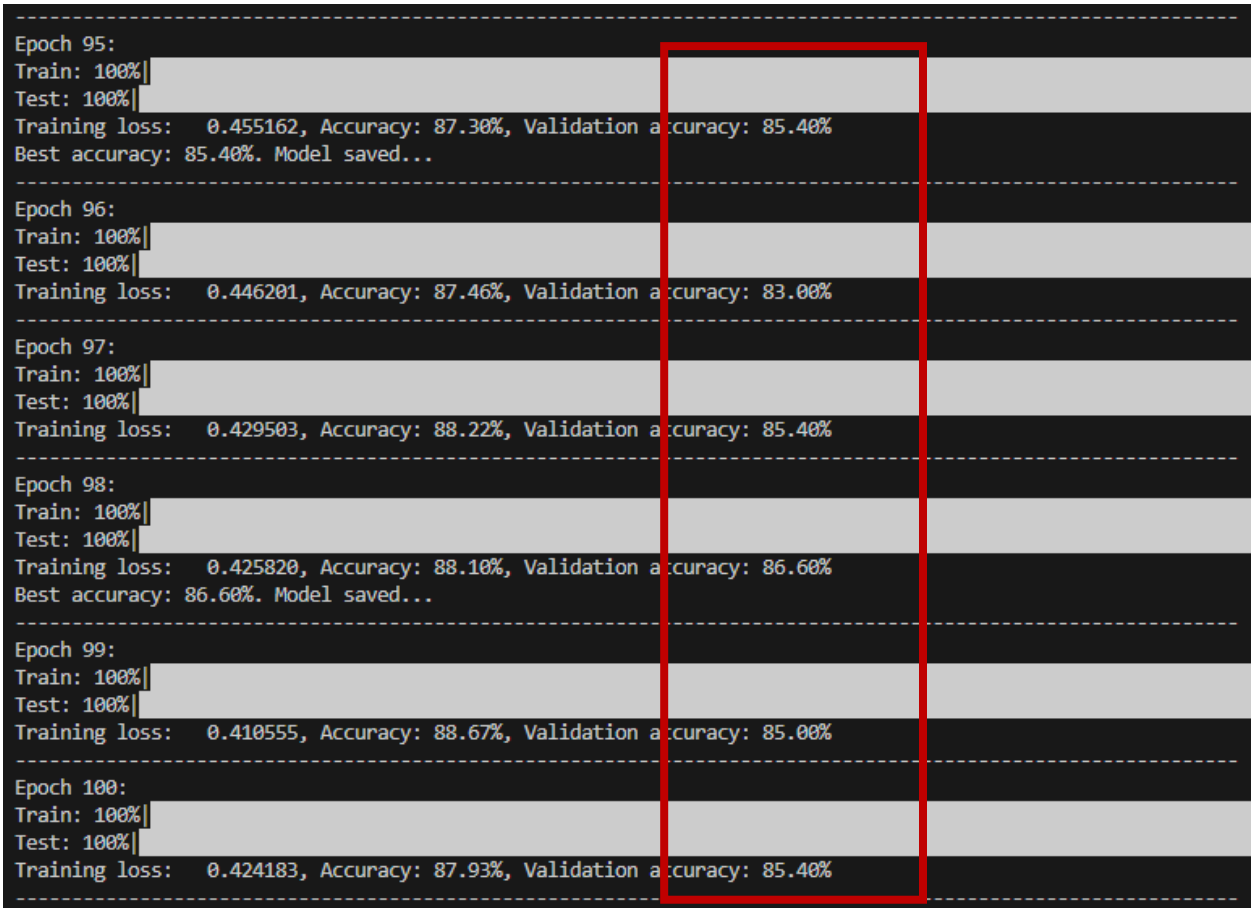


From the figure above, we can see that VGG19 has better accuracy than ResNet50 since the beginning of the training. The accuracy of VGG19 increase very fast at the beginning and start to increase slowly at about 40 epoch. For ResNet50, the accuracy just increase normally with the same gradient and started to go slowly at 85 epoch.

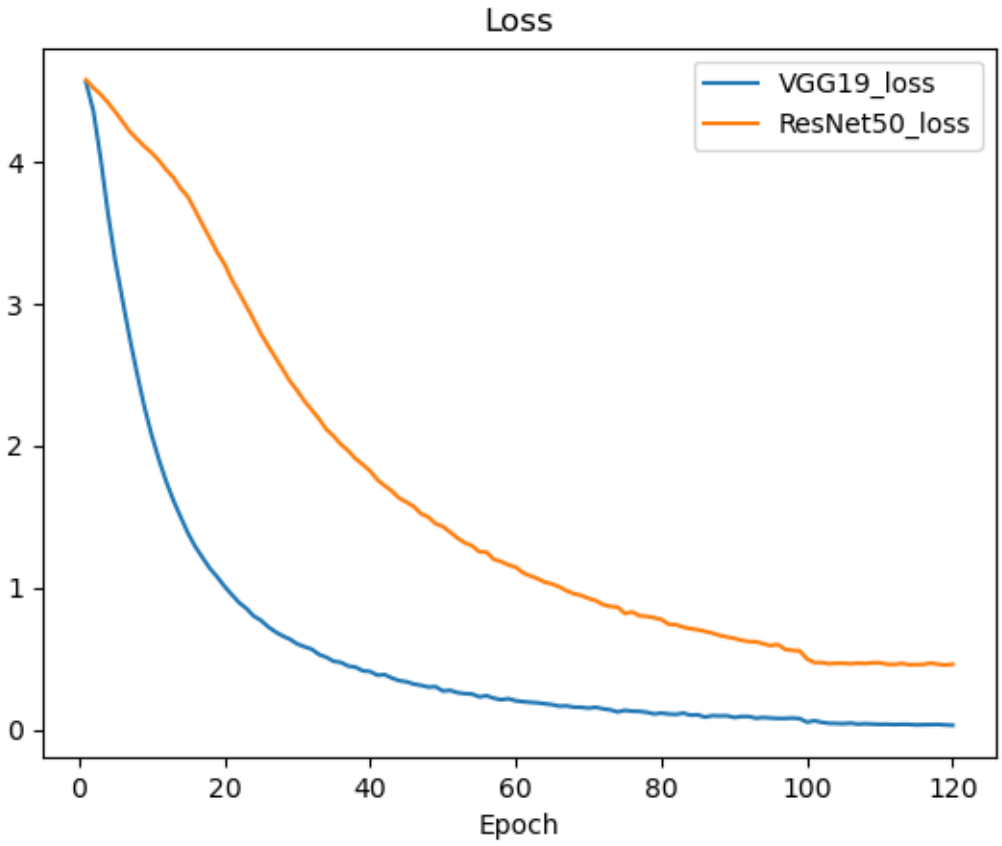
5. Discussion

Problem 1:

Through this lab, I understand the architecture of the VGG and ResNet. For my perspective, VGG is just a lot of convolutional layer and for ResNet, it is a different architecture. Theoretically, ResNet50 should have better performance than VGG19 since its architecture is to deal with the vanishing and exploding gradients problems. However, in my experiment, my ResNet50 did not go well as expected. It stated to oscillate around 85% of accuracy.



Now we look at the loss figure, it looks like VGG19 has converged completely after 70 epoch. For ResNet50, the loss kept dropping until the epoch is about 100 epoch and began to go flat. At first, I think it might be the lack of training for ResNet50 to converge, but now I had no idea for that problem. Finally, I still have a decent accuracy of VGG19 which is 95%.



Problem 2:

At first, both my model architectures do not have batch normalization layer after each convolutional layer. I thought that might not be a problem. However, when I started to train, the loss will never went down until I add the BN layer after the convolutional layer. I think it

was because the network is very deep that the loss cannot converge. After applying the normalizing technique, the loss began to go down.