Deep Learning Lab2: Butterfly & Moth Classification Report

ID: 412551030 Name: 黃羿寧

1. Introduction

Deep learning has been widely used in several fields such as image recognition, medical diagnosis and natural language processing. In field of image recognition, the particular focus has been placed on convolutional neural network (CNN)-based model for feature extraction, such as VGG[1] and ResNet[2] the runner up and the winner of ImageNet Large Scale Recognition Challenge (ILSVRC). To deeper understand the architecture, training process and comparison between two aforementioned models, this lab requests to build the model and train them from scratch to successfully classify the 100 species of butterflies and moths from open dataset. In this report, I will present the implementation details of two models, methods for data preprocessing and the results gained from the practice. In this practice, VGG19 and ResNet50 showed 67.6% and 90.8% prediction accuracy on testing dataset.
Dataset: https://www.kaggle.com/datasets/gpiosenka/butterfly-images40-species/data

2. Implementation Details

Run **main.py** you will get 150 iterations of training on two models and the test accuracy printed
The two models were trained on NVIDIA GeForce RTX 4090

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

In this lab, two models including VGG19 and ResNet50 are required to classify 100 species from images. The basic VGG19 architecture was shown in **Figure 1**. The VGG19 contains 19 convolutional layers with the kernel size of 3, and I applied batch-normalization and ReLU activation function right before each convolutional layer. The max polling was applied to down sample the image feature 5 times, so the pixels in dimension of width and height were decreased from (224, 224) to (7, 7) during the model. Finally, the fully connected layer exports an output with size of (4096, 100). The code is stored in VGG19 in the attachment.

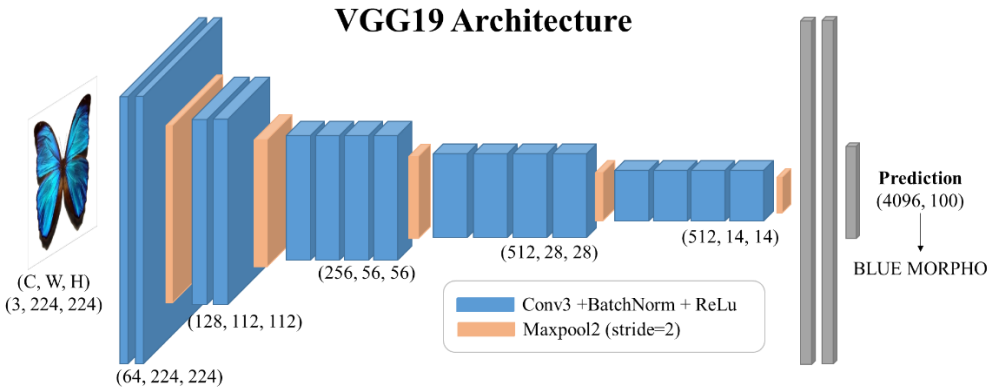

Figure 1. The architecture of VGG19

On the other hand, the ResNet50 architecture was illustrated in **Figure 2**. The ResNet50 model contains 50 layers of convolution, which is relatively deeper than VGG19. Most importantly, ResNet50 applied "**the bottleneck block**" to deal with the common problem revealed in the deep neural network: gradient vanish[3]. In traditional deep learning methods, the performance of models typically improves with an increase in the number of layers, but beyond a certain critical point, issues such as gradient vanish[3]or degradation problem occur. The issue of gradient vanish can be addressed through Batch Normalization, but the degradation problem requires the use of identity mapping in ResNet. This is achieved by storing the input before the convolution layer and adding it to the result after convolution[2], making the output of the additional layers that have not learned new features equal to the input. This ensures that if a layer does not learn new features, it does not lead to model degradation. The bottleneck block in ResNet50 was formed in the order of Conv1(ch), Conv3(ch), Conv1(ch×4), and the output will be concatenated to the input which is the shortcut of the model training. The ch derives the channel number, which will be 64, 128, 256 and 512 in the model. Additionally, the bottleneck block is repeated 3, 4, 6, and 3 time in ResNet50 model. Finally, the output will be in size of (2048, 100) to classify species. For the model training, the parameters used include a batch size of 16, and the Adam optimizer with a learning rate of 0.001 and a weight decay of 0.0001. The loss is calculated using cross-entropy.

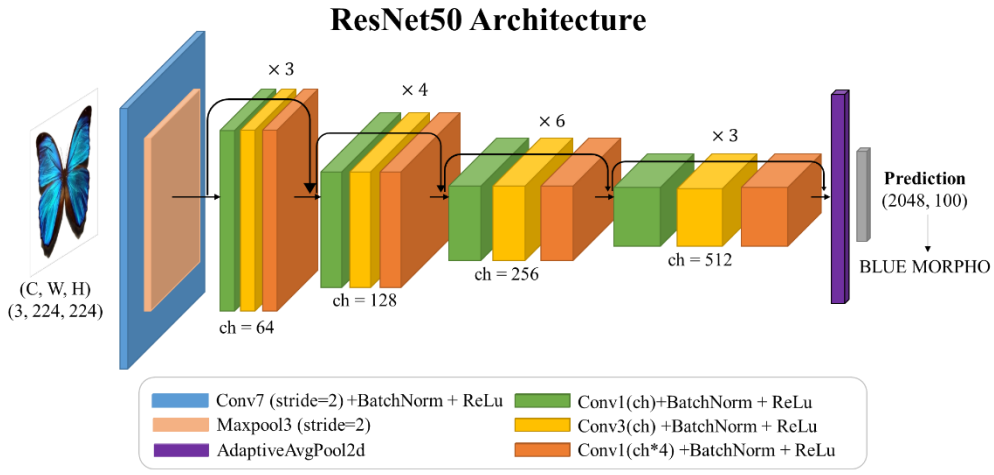

Figure 2. The architecture of ResNet50

The partial code of ResNet50

The `class conv_lay(nn.Module)` was the bottleneck block in the code. Specifically, "`Output += identity # Apply shortcut`" is the shortcut.

```
class conv_lay(nn.Module):
    """Left block + shortcut template"""
    def __init__(self, Input_Channel, Output_Channel, stride = 1, downsample=None):
        super(conv_lay, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(Input_Channel, Output_Channel, (1,1), 1, bias=False),
            nn.BatchNorm2d(Output_Channel),
            nn.ReLU(inplace = True))
        self.conv2 = nn.Sequential(
            nn.Conv2d(Output_Channel, Output_Channel, (3,3), stride = stride, bias=False, padding=1),
            nn.BatchNorm2d(Output_Channel),
            nn.ReLU(inplace = True))
```

```
        self.conv3 = nn.Sequential(
            nn.Conv2d(Output_Channel, Output_Channel*4, (1,1), 1, bias=False),
            nn.BatchNorm2d(Output_Channel*4))
        self.ReLu = nn.ReLU(inplace = True)
        self.downsample = downsample

    def forward(self, x):
        identity = x
        if self.downsample is not None:
            identity = self.downsample(x)
        Output = self.conv1(x)
        Output = self.conv2(Output)
        Output = self.conv3(Output)
        Output += identity # Apply shortcut
        Output = self.ReLu(Output)
        return Output
```

B. The details of your Data loader

The partial code of the dataloader.py was shown in Figure 3. The data list was first loaded through getData function which reads the csv file of train, test or valid trials. And then I checked if the file's existence to make sure there is no mismatch problem. Additionaly, I applied some data pre-processing in the _getitem_ function in the dataloader.py, which was shown in Figure 4. The self.transform was assigned in Figure 3 (left), while the methods used for dataset for training and validation are the same. The pre-processing details will be listed in the section 3.

```
class BufferflyMothLoader(data.Dataset):
    def __init__(self, root, mode):
        self.root = root
        self.img_name, self.label = getData(mode)
        self.mode = mode
        if self.mode == 'test':
            self.transform = transforms.Compose([
                transforms.ToTensor(),
            ])
        else:
            self.transform = transforms.Compose([
                transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5),
                transforms.ToTensor(),
                # Convert to tensor (this also normalizes pixel values to [0, 1])
            ])

        print("> Found %d images..." % (len(self.img_name)))
```

```
def getData(mode):
    if mode == 'train':
        df = pd.read_csv('dataset/train.csv')
    elif mode == 'test':
        df = pd.read_csv('dataset/test.csv')
    else:
        df = pd.read_csv('dataset/valid.csv')

    path = df['filepaths'].tolist()
    label = df['label_id'].tolist()

    # clean data check the list
    valid_indices = [i for i, p in enumerate(path) if os.path.exists(os.path.join('dataset', p))]
    path = [path[i] for i in valid_indices]
    label = [label[i] for i in valid_indices]
    return path, label
```

Figure 3. The partial code of dataloader.py

```
def __getitem__(self, index):
    """step1. Get the image path from 'self.img_name' and load it."""
    path = os.path.join(self.root, 'dataset', self.img_name[index])
    img = Image.open(path).convert('RGB')

    """step2. Get the ground truth label from self.label"""
    label = self.label[index]

    """step3. Transform the .jpg rgb images during the training phase,
              rotation, cropping, normalization etc. But at the beginni

              In the testing phase, if you have a normalization proces

              hints : Convert the pixel value to [0, 1]
                      Transpose the image shape from [H, W, C] to [C, H
    img = self.transform(img)

    # print(img.shape)
    """step4. Return processed image and label"""
    return img, label
```

Figure 4. The load data and data preprocess code in dataloader.py

3. Data Preprocessing

A. Steps of data preprocessing

The data preprocessing is essential for image data which are used as input of the DL model. The aim of data preprocessing is not limited to raise the quality of image and lead to better classification accuracy, but also increase the generalizability of the model which could also perform well in difficult data. The basic methods for image data preprocessing[4] include resize, rotation, color correction, contrast enhancement, noise reduction and so on. In this lab, since the size of the data was fitted at (224, 224, 3) and the kernel size of the selected model was not modified, the resize and rotation methods were excluded. However, to increase the classification capability of the model, I applied **color jitter** to the train and the validation dataset in the model training. The transformation of color randomly applied jitter to **brightness, contrast and saturation** to the original figure. The example below showed the 4 figures applied function.

Figure 5. The example of color jittered figures

B. What make this method special?

The color of the butterflies and the moths was very close, but the patterns on their wings are very different. I used this method aiming to reduce the model's reliability on their color and focus on other pattern to compensate the uncertainty of the color itself.

4. Experimental results

A. The highest testing accuracy

The testing accuracy was calculated on model which gained the maximum validation accuracy. During the 150 epochs of training, the VGG19 achieved the highest validation accuracy of 69% at the last epoch and gained the 67.6% prediction accuracy on testing dataset based on this model weight. On the other hand, the ResNet50 achieved the 89.4% accuracy on the 70th epoch and the testing accuracy is 90.8% by using this model weights.

```
----Test Data load----
> Found 500 images...
ResNet50 (argmax valid ep = 69): Train acc =0.9882, Valid acc =0.8940, Test acc=0.9080
VGG19 (argmax valid ep = 149): Train acc =0.9929, Valid acc =0.6900, Test acc=0.6760
```

B. Comparison figures

The accuracy and loss of training and validation during the training process was shown in **Figure 6.** The ResNet50 shows both the steeper learning slope in the classification and the higher validation accuracy in the comparison. The results demonstrate the high performance of ResNet50 in this classification practice.

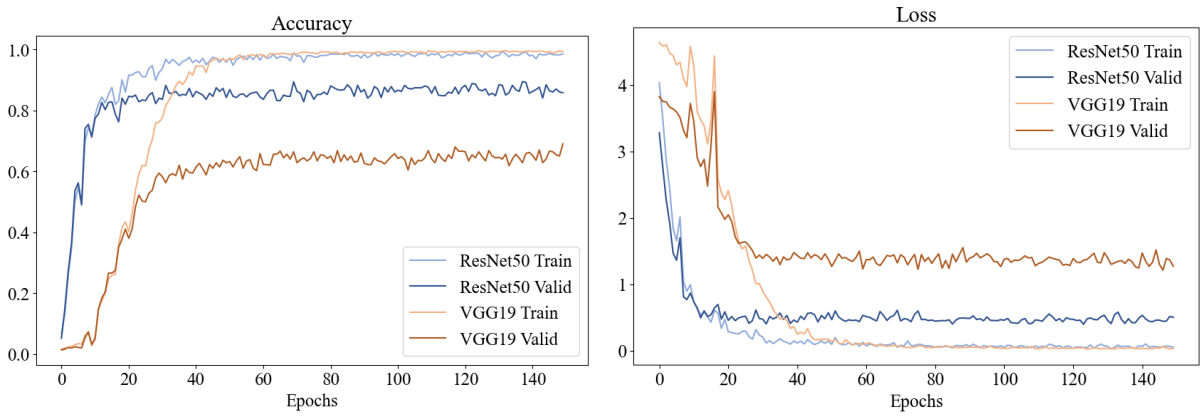

Figure 6. The comparison figures

5. Discussion

A. Overfitting issue

The result shown in **Figure 6** highlights the issue of overfitting, characterized by the training accuracy (loss) reaching its peak while the validation accuracy stops to improve. To address this problem in the future, I will consider implementing early stopping as a measure to reduce computational costs.

B. Model Performance

Regarding the outcomes of model training, previous studies suggested that the batch size and learning rate significantly affect the overall quality of training, especially in terms of generalizability[5]. According to Kandel et al. (2020), the performance of VGG16 on histopathology dataset, the combination of batch size = 16 and the Adam optimizer with the learning rate of 0.0001 achieved the best performance in their comparative results. Due to time constraints in this practice, if more time were available, a comparison of learning rates and batch sizes would be conducted to enhance the testing accuracy of our model. Additionally, beyond conducting a grid search for the optimal batch size and learning rate combination, integrating adaptive learning rate strategies into the training process could offer further enhancements in model training efficiency and effectiveness. Lastly, learning from two incorrect predictive cases in the test dataset highlighted an important challenge: the species listed appear very similar to each other (**Figure 7**). To improve model performance, I plan to increase the training data volume through data augmentation techniques in the future. For example, implementing image flipping, rotation, larger color jitter could significantly improve the model's generalizability in recognizing specific patterns, such as wing patterns in biological imaging. More importantly, in the example shows in **Figure 7**, the second example shows the possibility of the mismatch ground truth

label in the datasets. From my opinion, this testing butterfly #2 looks closer to the label [68] Orange Tip than its assigned label [56] Large Marble.

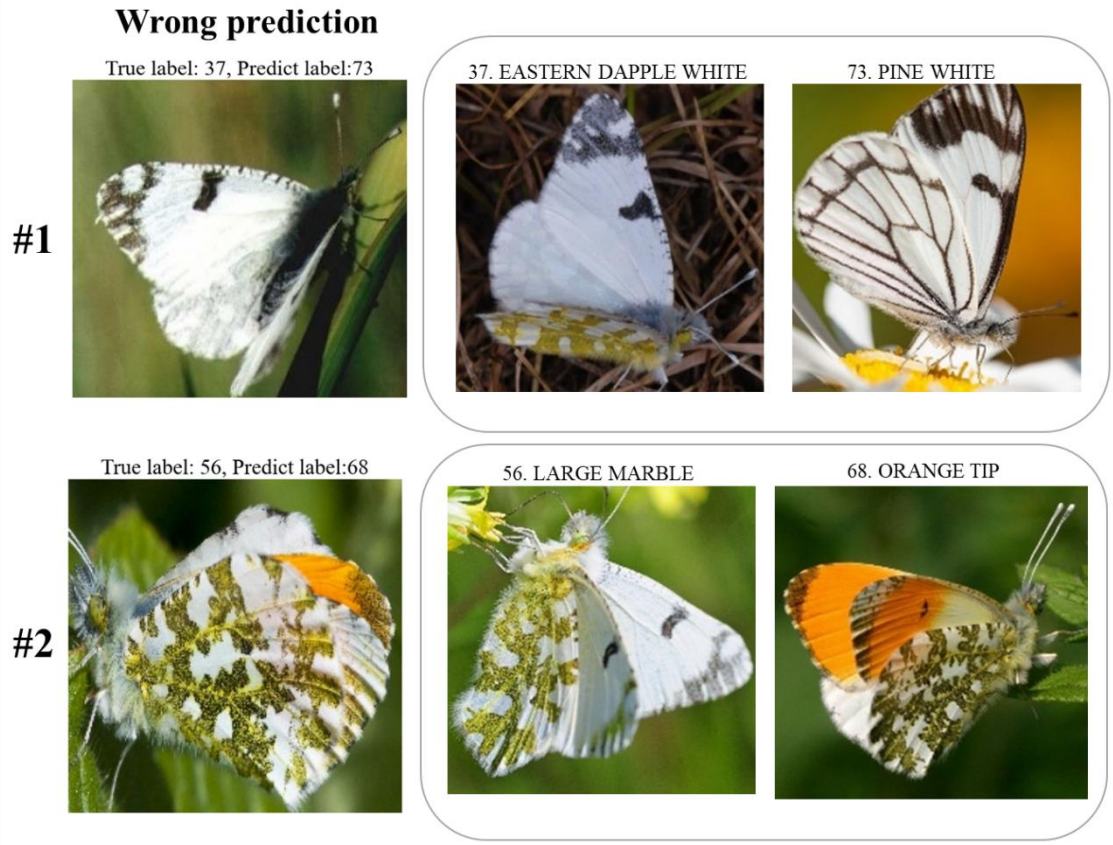

Figure 7. The incorrect prediction in the test dataset

6. References

1 K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556,* 2014.

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*, 2016, pp. 770-778.

3 Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks,* vol. 5, no. 2, pp. 157-166, 1994.

4 M. Sonka, V. Hlavac, R. Boyle, M. Sonka, V. Hlavac, and R. Boyle, "Image pre-processing," *Image processing, analysis and machine vision,* pp. 56-111, 1993.

5 I. Kandel and M. Castelli, "The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset," *ICT express,* vol. 6, no. 4, pp. 312-315, 2020.