Lab 2 Butterfly vs. Moth Classification

109550121 温柏萱

Introduction

This homework aims to implement the classical Visual Geometry Group network and Residual Networks to create a butterfly and moth classifier and compare the performance of the two networks.

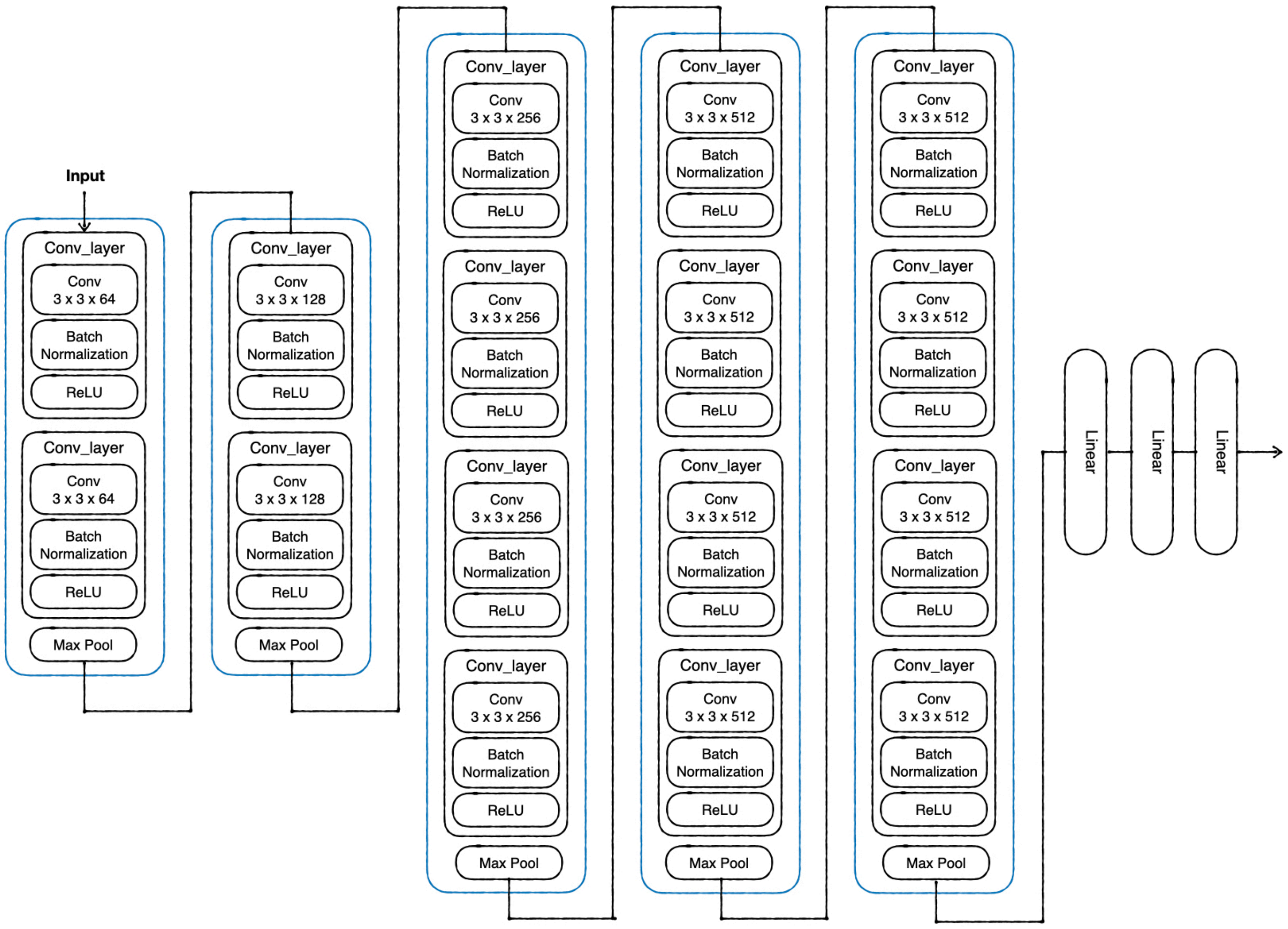
The family of VGG network comes from the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition", using 3×3 filters to increase the depth, and reduces volume size with max pooling. It won the first runner-up of the ILSVRC (ImageNet Large Scale Visual Recognition Competition) 2014 in the classification task. Its simplicity and and performance is widely recognized, but has the disadvantages of being computationally heavy and is likely to overfit when the dataset is too small.

ResNet is introduced in "Deep Residual Learning for Image Recognition", aiming to address the gradient vanishing problem that happens when training deep neural networks. It utilized the residual connection for the back propagation to be done smoothly. Another advantage of the ResNet family is the reduction in parameters compared to traditional CNNs, the depth is increased without propotional increase in parameters.

Implementation Details

VGG19

VGG19 consisits of 19 layers, the first 16 layers are constructed by a sequence of 3×3 convolutional layers of varied depth, visualized as the figure below.



To simplify the implementation, I created a `conv_layer` class for the basic building block of the VGG19 network. The `conv_layer` consists of a convolutional layer, batch normalization and an activation. The code of `conv_layer` is as follows:

```
class conv_layer(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=3, stride=1,
padding=None):
        super(conv_layer, self).__init__()
        if padding is None:
            padding = kernel_size // 2
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size,
stride, padding, bias=True)
        self.bn = nn.BatchNorm2d(out_channels)
        self.act = nn.ReLU(inplace=True)

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        x = self.act(x)
        return x
```

With the `conv_layer` class, we can construct the `VGG19` class:

```
class VGG19(nn.Module):
    def __init__(self, num_classes=1000):
        super(VGG19, self).__init__()
        self.stage1 = self.layer_init(3, 64, kernel_size=3, stride=1,
padding=1, num_blocks=2)
        self.stage2 = self.layer_init(64, 128, kernel_size=3, stride=1,
padding=1, num_blocks=2)
        self.stage3 = self.layer_init(128, 256, kernel_size=3, stride=1,
padding=1, num_blocks=4)
        self.stage4 = self.layer_init(256, 512, kernel_size=3, stride=1,
padding=1, num_blocks=4)
        self.stage5 = self.layer_init(512, 512, kernel_size=3, stride=1,
padding=1, num_blocks=4)

        self.tail = nn.Sequential(*[
            nn.Flatten(),
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes)
        ])
    def layer_init(self, in_channels, out_channels, kernel_size=3, stride=1, padding=1, num_blocks=1):
        layers = [conv_layer(in_channels, out_channels, kernel_size=kernel_size, stride=stride, padding=padding)]
        for _ in range(1, num_blocks):
            layers.append(conv_layer(out_channels, out_channels, kernel_size=kernel_size, stride=stride, padding=padding))
        layers.append(nn.MaxPool2d(kernel_size=2, stride=2, padding=0))
        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.stage1(x)
        x = self.stage2(x)
        x = self.stage3(x)
        x = self.stage4(x)
        x = self.stage5(x)
        x = self.tail(x)
        return x
```

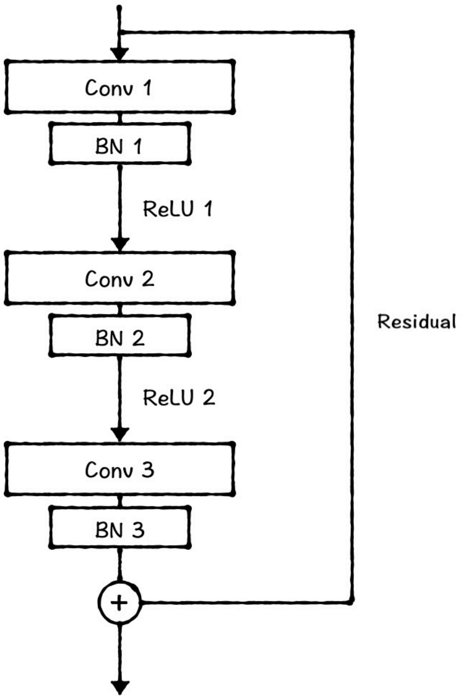
The first 5 `conv_layers` are constructed by the `layer_init` function, which ensembles a number of `conv_layers` specified by the parameter `num_blocks`. Following the 5 `conv_layers`, I used `nn.Sequential` to pack the final linears together to simplify the implementation when forwarding.

ResNet50

Bottleneck Block

ResNet50 replaced the basic building block with the bottleneck block structured as the following figure. The bottleneck block enables a deeper network structure as the shortcut connection bypasses three convolutional layers, adding a batch normalization layer between the convolution layer and activation function stabilizes the learning process, speeds up the training and lowers the sensivity to initialization.

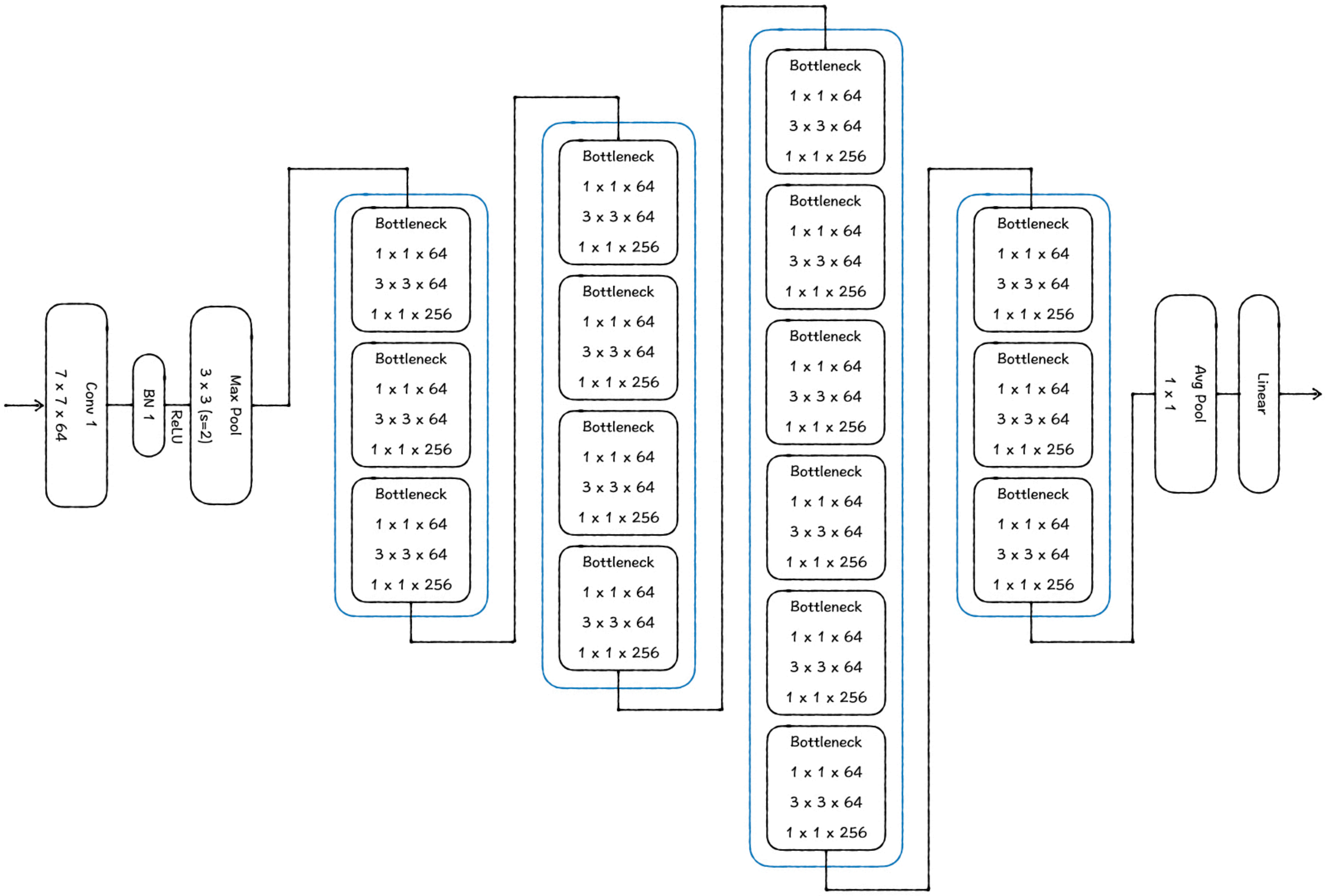
The bottleneck block consists of three convolutional layers of size $1 \times 1, 3 \times 3, 1 \times 1$, a batch normalization layer in between the activation function, then activate the output of the batch normalization with `ReLU`. The residual shortcut path passes the input directly to the output, preventing the problem of gradient vanishing from happening.



```
class BottleneckBlock(nn.Module):
    expansion = 4
    def __init__(self, in_channels, out_channels, stride=1):
        super(BottleneckBlock, self).__init__()

        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=1,
bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu1 = nn.ReLU(inplace=True)

        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
stride=stride, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.relu2 = nn.ReLU(inplace=True)

        self.conv3 = nn.Conv2d(out_channels, out_channels *
self.expansion, kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(out_channels * self.expansion)
        self.relu3 = nn.ReLU(inplace=True)

        self.residual = None
        if stride != 1 or in_channels != out_channels *
BottleneckBlock.expansion:
```

```
            self.residual = nn.Sequential(
                nn.Conv2d(in_channels, out_channels *
BottleneckBlock.expansion, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels * BottleneckBlock.expansion),
            )

        self.stride = stride

    def forward(self, x):
        identity = x

        output = self.conv1(x)
        output = self.bn1(output)
        output = self.relu1(output)

        output = self.conv2(output)
        output = self.bn2(output)
        output = self.relu1(output)

        output = self.conv3(output)
        output = self.bn3(output)

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

        output += identity
        output = self.relu3(output)

        return output
```

ResNet50 uses a 7×7 convolutional layer to extract the basic features, shrink down the spatial dimension and increase the depth. A batch normalization is applied to help adjust and scale up the activations. The Global Average Pooling to reduce the dimension of the feature maps. Then apply a series of layered bottleneck blocks with $3, 4, 6, 3$ blocks each in its own layer, then global average pooling averages the spatial dimension to 1×1, producing a single vector for each feature map, focusing on the presence of features over their location. Finally, a fully connected layer is applied to map the learnt features to the target classes.



```python
class RN50(nn.Module):
    def __init__(self, num_classes=1000):
        super(RN50, self).__init__()
        self.in_channels = 64
        # Initial convolution
        self.conv1 = nn.Conv2d(3, self.in_channels, kernel_size=7,
stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(self.in_channels)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        # ResNet layers
        self.layer1 = self.layer_init(out_channels=64, blocks=3, stride=1)
        self.layer2 = self.layer_init(out_channels=128, blocks=4,
stride=2)
        self.layer3 = self.layer_init(out_channels=256, blocks=6,
stride=2)
        self.layer4 = self.layer_init(out_channels=512, blocks=3,
stride=2)
```

```
        # Final layers
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * BottleneckBlock.expansion, num_classes)

    def layer_init(self, out_channels, blocks, stride):
        layers = []
        layers.append(BottleneckBlock(self.in_channels, out_channels,
stride))
        self.in_channels = out_channels * BottleneckBlock.expansion
        for _ in range(1, blocks):
            layers.append(BottleneckBlock(self.in_channels, out_channels))

        return nn.Sequential(*layers)

    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 = torch.flatten(x, 1)
        x = self.fc(x)

        return x
```

In ResNet50, I also used a `layer_init` function to pack the bottleneck blocks of the same size in a layer. The number of bottleneck blocks is specified by the parameter `blocks`.

Data Loader

For training samples, the data loader first resizes the image to 224 × 224, then apply random horizonal and vertical flip with probability 0.5, then convert the image to tensor and normalize it.

For testing samples, the data loader resizes the image to 224 × 224 then apply normalization.

```
class BufferflyMothLoader(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)

    def __getitem__(self, index):

        if self.img_name[index].find('jpg'):
            path = self.root + self.img_name[index]
        else:
            path = self.root + self.img_name[index] + '.jpg'
        image = Image.open(path).convert('RGB')
        label = self.label[index]

        if self.mode == 'train':
            transform = transforms.Compose([
                transforms.Resize((224, 224)),  # Resize images to 224x224
                transforms.RandomHorizontalFlip(p=0.5),  # Horizontal Flip with probability 0.5
                transforms.RandomVerticalFlip(p=0.5), # Vertical Flip with probability 0.5
                transforms.ToTensor(),
            ])

        else:
            transform = transforms.Compose([
                transforms.Resize((224, 224)),
                transforms.ToTensor(),
            ])
        img = transform(image)

        return img, label
```

Data Preprocessing

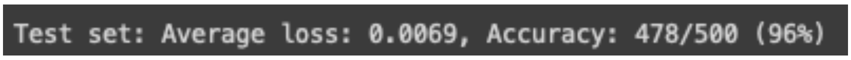
1. Resize
   Resizing the image to 224×224 enhances the efficiency of computation since Convolutional Neural Network assumes a fixed input size, and could therefore void the need to handle varied input size.
2. Random Flipping
   Apply random horizontal and vertical flipping to the image introduces variability to the training data, increasing the generalize ability of the model.
3. Transformation (done by `ToTensor()`)

   Map the pixel values from $[0, 255]$ to $[0, 1]$ and transpose the data from (h, w, c) to (c, h, w), where h, w, c represent height, width and number of channels of the input image. Re-scaling the values from $[0, 255]$ to $[0, 1]$ helps stable the training process, transposing the input makes it compatible to be fed into the neural network.

Experimental Results

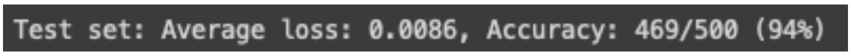
Test Accuracy

- VGG19



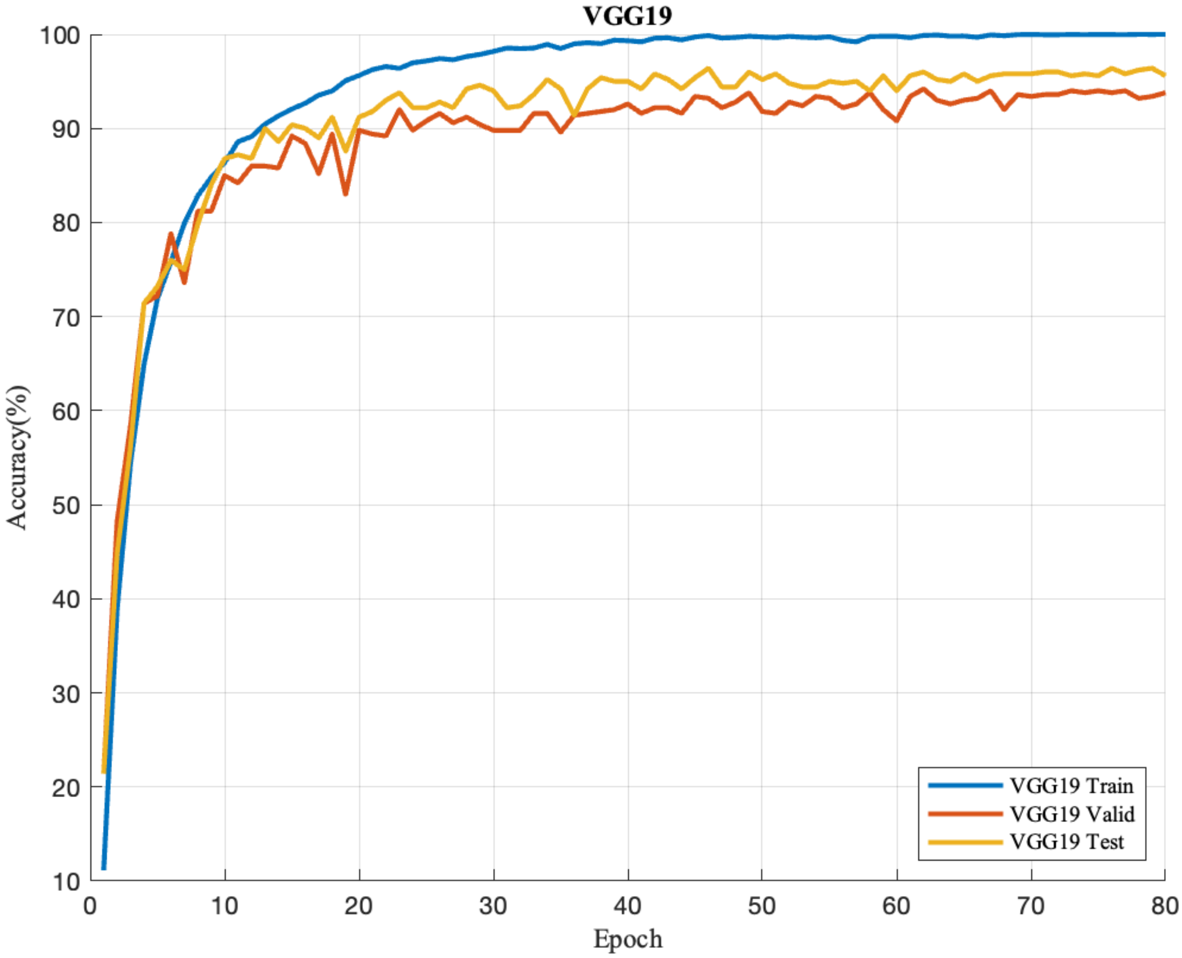
```
Test set: Average loss: 0.0069, Accuracy: 478/500 (96%)
```

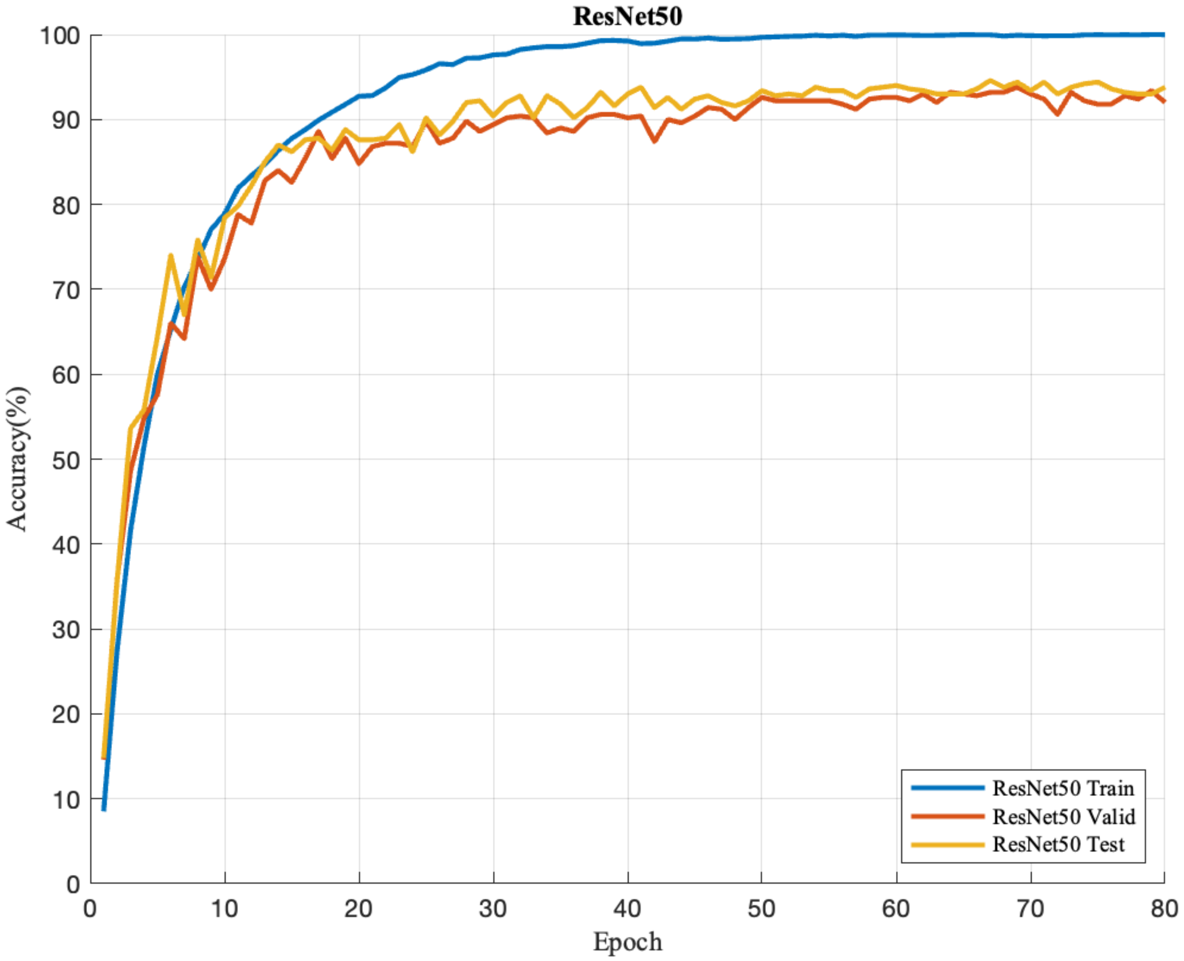
- ResNet50

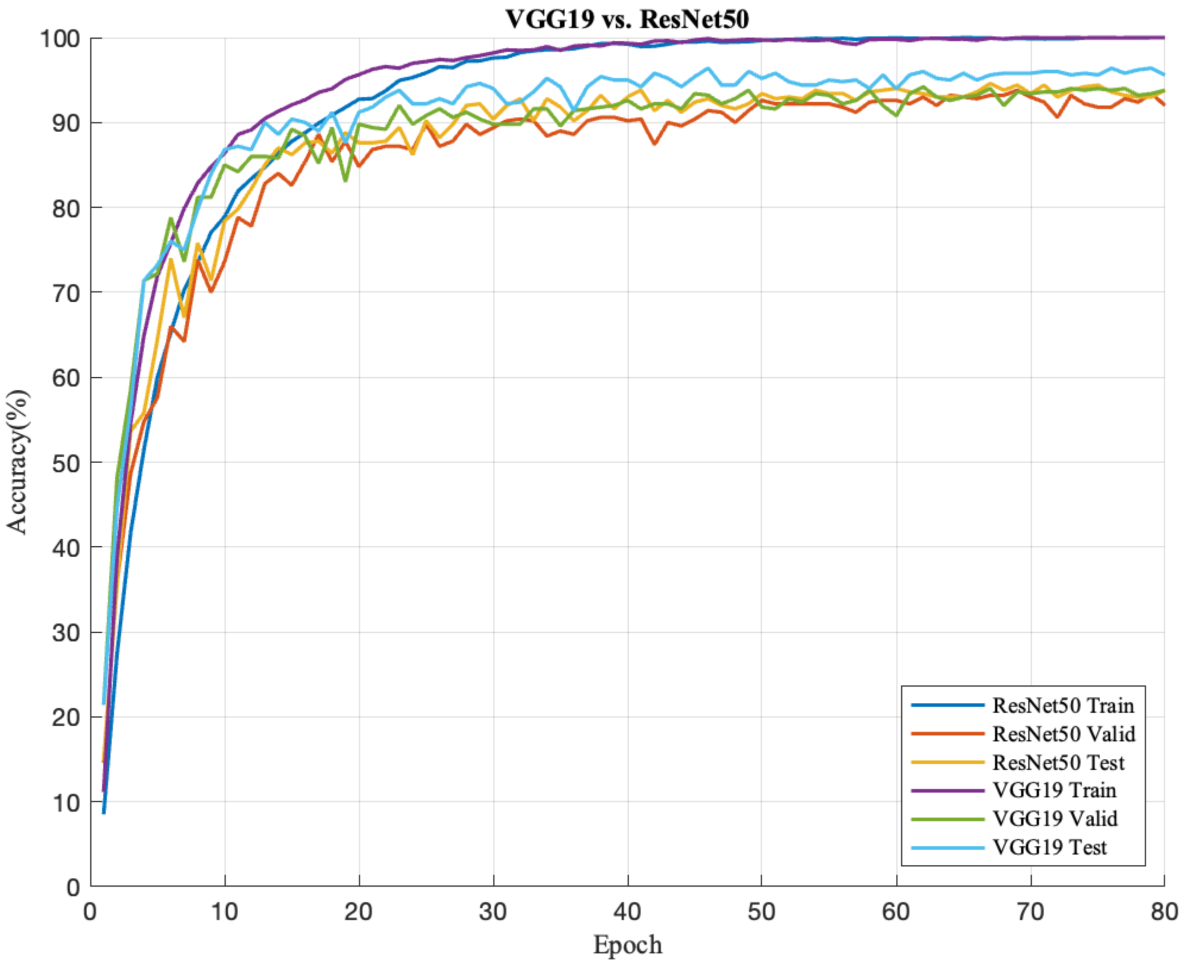


```
Test set: Average loss: 0.0086, Accuracy: 469/500 (94%)
```

Comparison Figures (Learning Curve)




ResNet50
Accuracy(%)
100
90
80
70
60
50
40
30
20
10
0
0
10
20
30
40
50
60
70
80
Epoch
ResNet50 Train
ResNet50 Valid
ResNet50 Test



Discussions

Model Comparision (VGG19 vs. ResNet50)

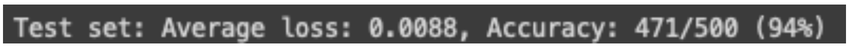
Both VGG19 and ResNet50 has high accuracy on the test set, implying the successfully learning on the dataset.

VGG19 has a higher accuracy (2% higher) on both test and validation set with multiple runs, indicating that it better captures the differences between butterflies and moths. VGG19 is shallower compared to ResNet50, but with good performance when the dataset isn't unpleasently large and complicated. Deeper network does not always lead to superior performances, when the problem does not benefit significantly from deeper networks, shallower network could outperform deeper networks.

ResNet18

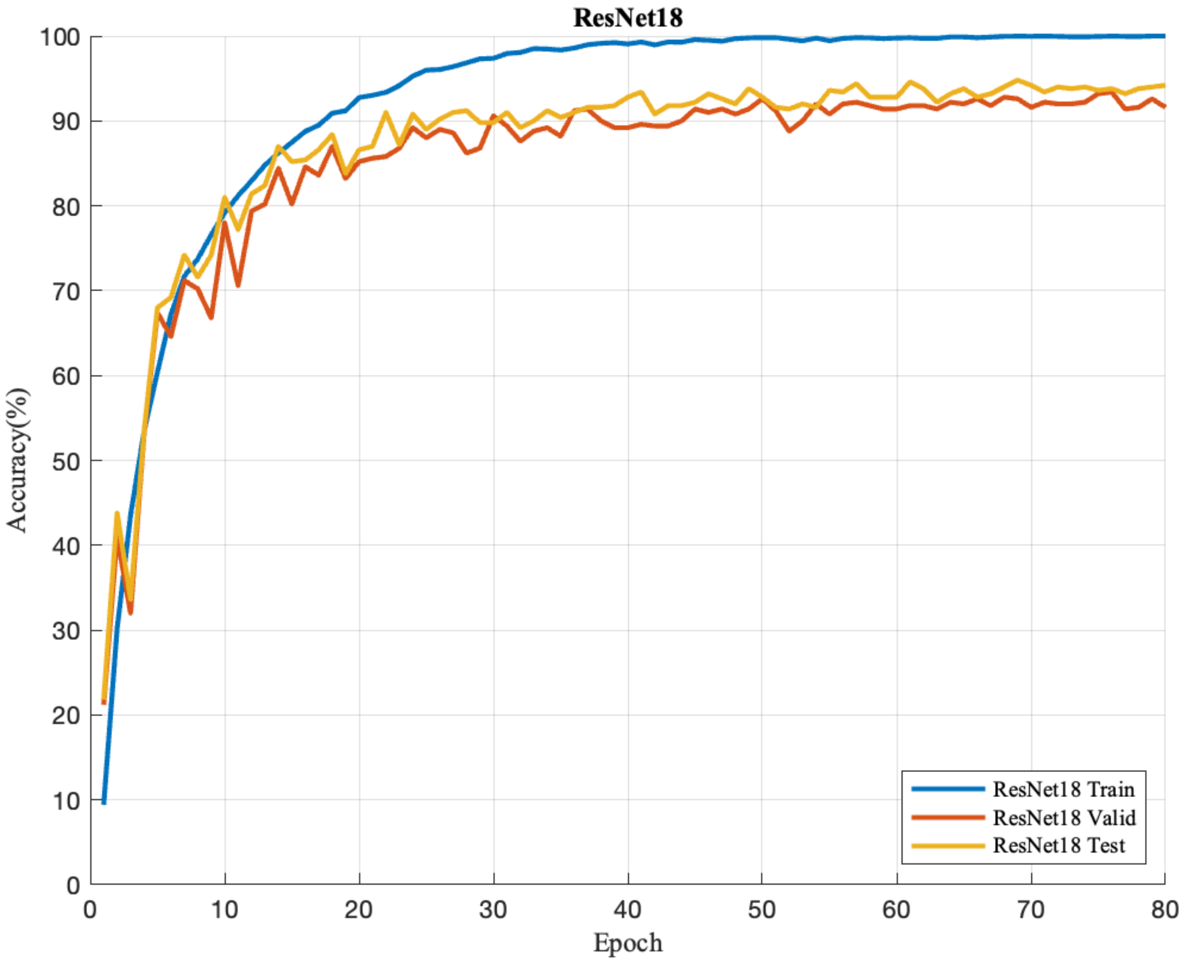
To verify if the structural of two networks matter, I also implemented ResNet18 to compare with VGG19 and ResNet50.

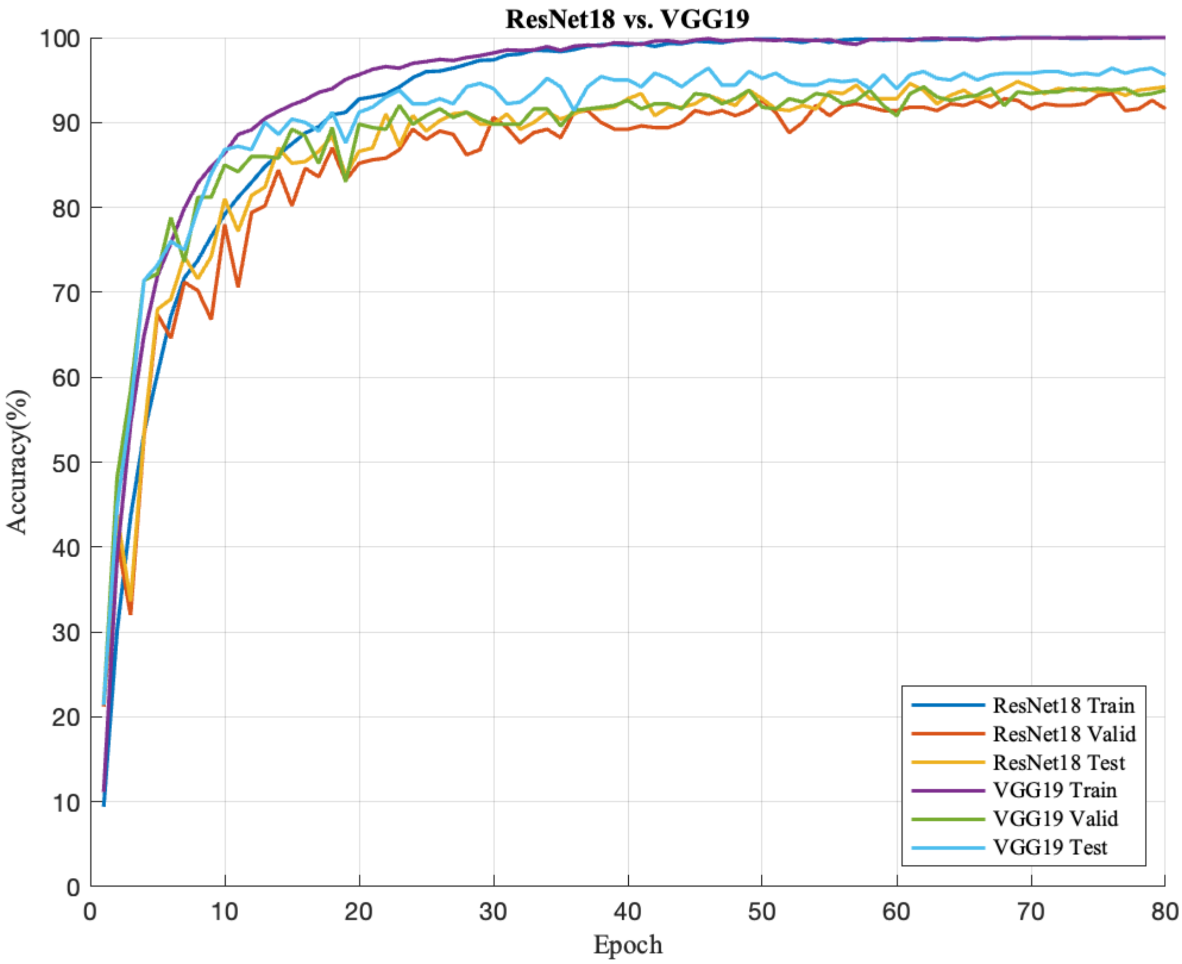
Test Accuracy

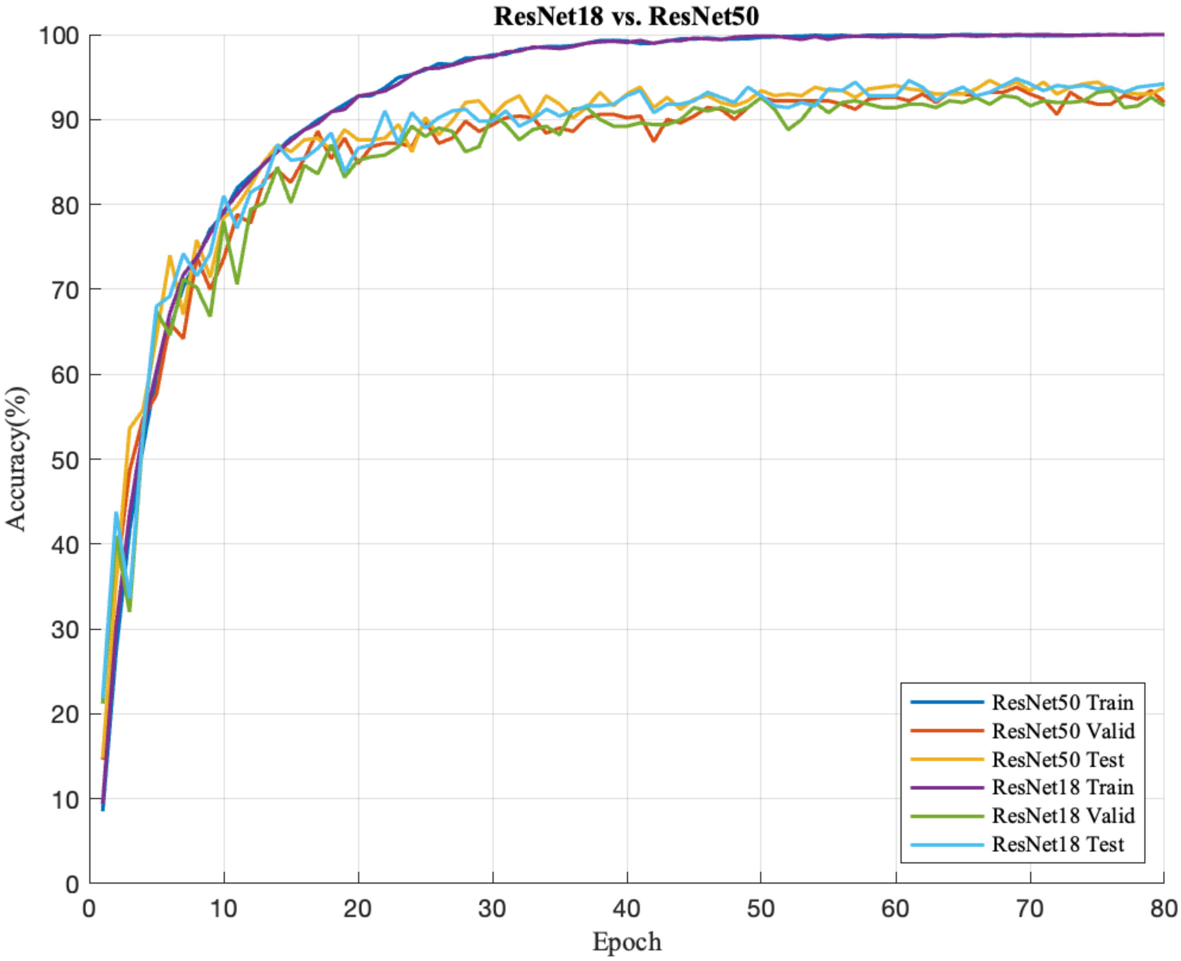


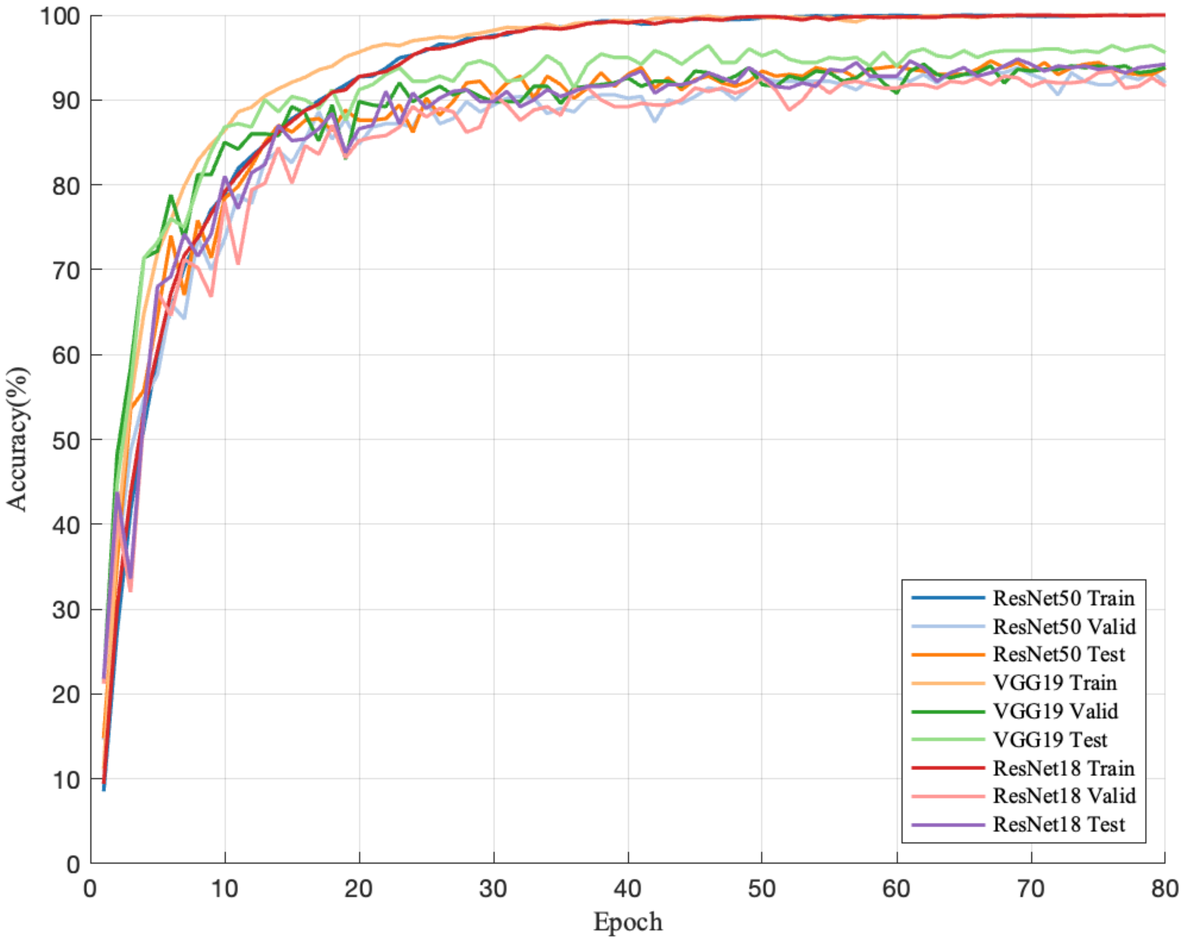
```
Test set: Average loss: 0.0088, Accuracy: 471/500 (94%)
```

Learning Curve




ResNet18 vs. VGG19
Accuracy(%)
0
10
20
30
40
50
60
70
80
90
100
Epoch
0
10
20
30
40
50
60
70
80
ResNet18 Train
ResNet18 Valid
ResNet18 Test
VGG19 Train
VGG19 Valid
VGG19 Test


ResNet18 vs. ResNet50
Accuracy(%)
0
10
20
30
40
50
60
70
80
90
100
Epoch
0
10
20
30
40
50
60
70
80
ResNet50 Train
ResNet50 Valid
ResNet50 Test
ResNet18 Train
ResNet18 Valid
ResNet18 Test



The learning curve of ResNet18 and ResNet50 almost overlap with eachother, and the test accuracy are identical with both being 94%, indicating the depth of the network did not lead to significant difference on this task, also explaining why a shallower network could out perform a deeper one. This leads to the difference in the structure between VGG19 and ResNet that caused the difference in the accuracy.

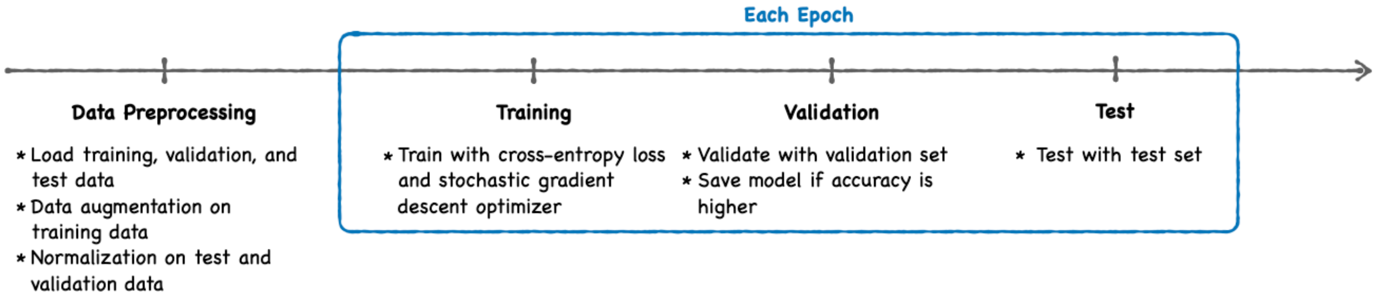
Data Preprocessing-Normalization

```
class BufferflyMothLoader(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)

    def __getitem__(self, index):

        if self.img_name[index].find('jpg'):
            path = self.root + self.img_name[index]
        else:
            path = self.root + self.img_name[index] + '.jpg'
        image = Image.open(path).convert('RGB')
        label = self.label[index]

        if self.mode == 'train':
            transform = transforms.Compose([
                transforms.Resize((224, 224)),  # Resize images to 224x224
                transforms.RandomHorizontalFlip(p=0.5),  # Horizontal Flip with probability 0.5
                transforms.RandomVerticalFlip(p=0.5), # Vertical Flip with probability 0.5
                transforms.ToTensor(),
                transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
            ])

        else:
            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])
            ])
        img = transform(image)
        return img, label
```

In data preprocessing, I also tried normalizing the images to some typical values of the ImageNet dataset, but didn't make a difference on the accuracy.

Training Process



Each Epoch
Data Preprocessing
* Load training, validation, and test data
* Data augmentation on training data
* Normalization on test and validation data
Training
* Train with cross-entropy loss and stochastic gradient descent optimizer
Validation
* Validate with validation set
* Save model if accuracy is higher
Test
* Test with test set

After loading the datasets, the iterative training process can be divided into three phases: first train on the train set and modify the weights and bias accordingly, then use validation dataset and test set to monitor the performance of the model. Also, we include a conditional update mechanism when saving the model, that is, update the saved model only when the validation accuracy is by far the highest, this mechanism can prevent overfitting to the training data.