DLP lab2

Introduction

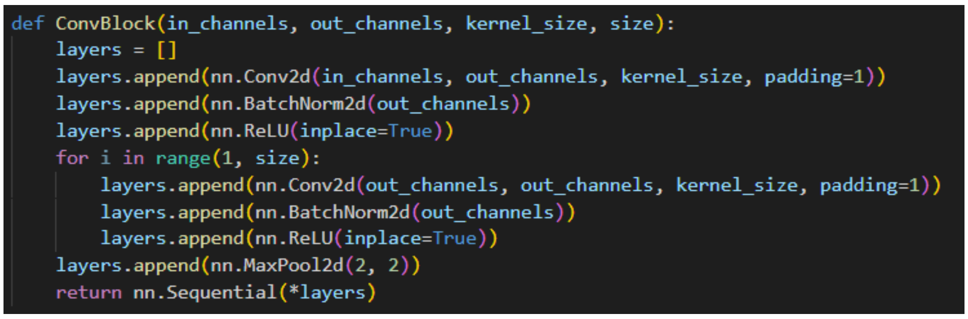
In this lab, we aim to classify a dataset comprising one hundred species of butterflies and moths using deep learning techniques. Our approach involves custom data loading and preprocessing using PyTorch, followed by classification using VGGNet and ResNet architectures. We'll monitor model performance by plotting accuracy curves for each epoch and identify the architecture achieving the higher accuracy.

Implementation Details

A. The details of your model

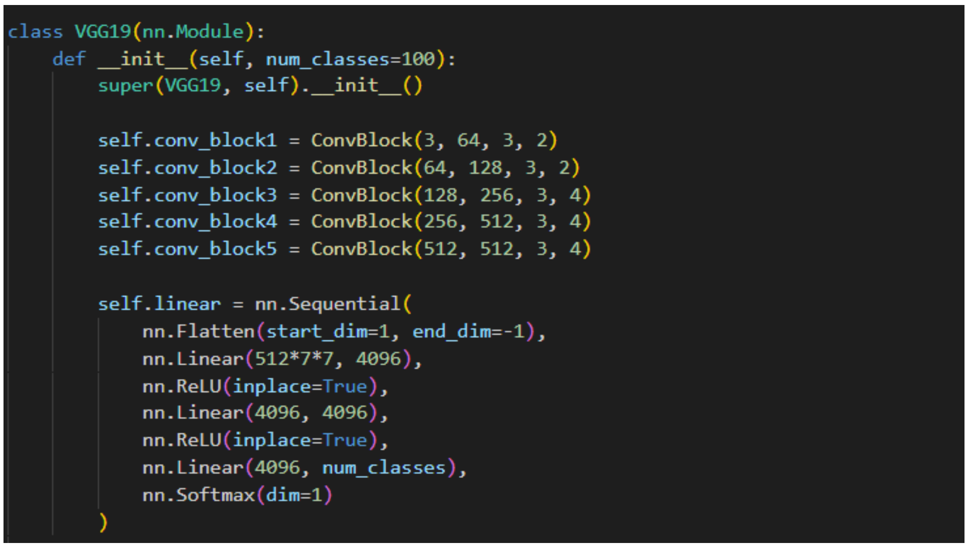
VGG

Convolution blocks

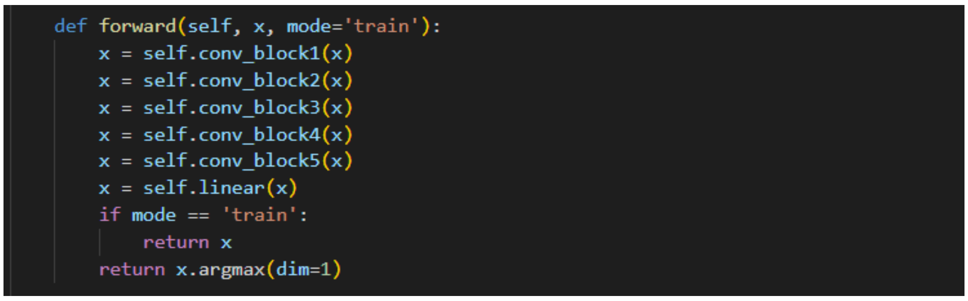


```
def ConvBlock(in_channels, out_channels, kernel_size, size):
    layers = []
    layers.append(nn.Conv2d(in_channels, out_channels, kernel_size, padding=1))
    layers.append(nn.BatchNorm2d(out_channels))
    layers.append(nn.ReLU(inplace=True))
    for i in range(1, size):
        layers.append(nn.Conv2d(out_channels, out_channels, kernel_size, padding=1))
        layers.append(nn.BatchNorm2d(out_channels))
        layers.append(nn.ReLU(inplace=True))
    layers.append(nn.MaxPool2d(2, 2))
    return nn.Sequential(*layers)
```

Stack the blocks



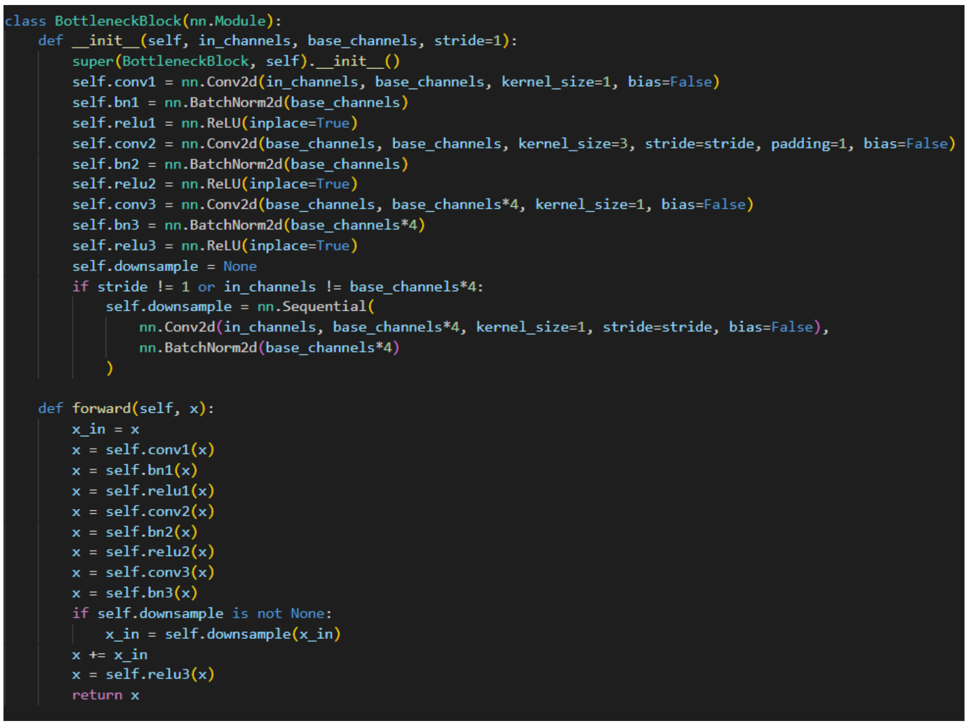
```
class VGG19(nn.Module):
    def __init__(self, num_classes=100):
        super(VGG19, self).__init__()

        self.conv_block1 = ConvBlock(3, 64, 3, 2)
        self.conv_block2 = ConvBlock(64, 128, 3, 2)
        self.conv_block3 = ConvBlock(128, 256, 3, 4)
        self.conv_block4 = ConvBlock(256, 512, 3, 4)
        self.conv_block5 = ConvBlock(512, 512, 3, 4)

        self.linear = nn.Sequential(
            nn.Flatten(start_dim=1, end_dim=-1),
            nn.Linear(512*7*7, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
            nn.Softmax(dim=1)
        )
```



```
    def forward(self, x, mode='train'):
        x = self.conv_block1(x)
        x = self.conv_block2(x)
        x = self.conv_block3(x)
        x = self.conv_block4(x)
        x = self.conv_block5(x)
        x = self.linear(x)
        if mode == 'train':
            return x
        return x.argmax(dim=1)
```

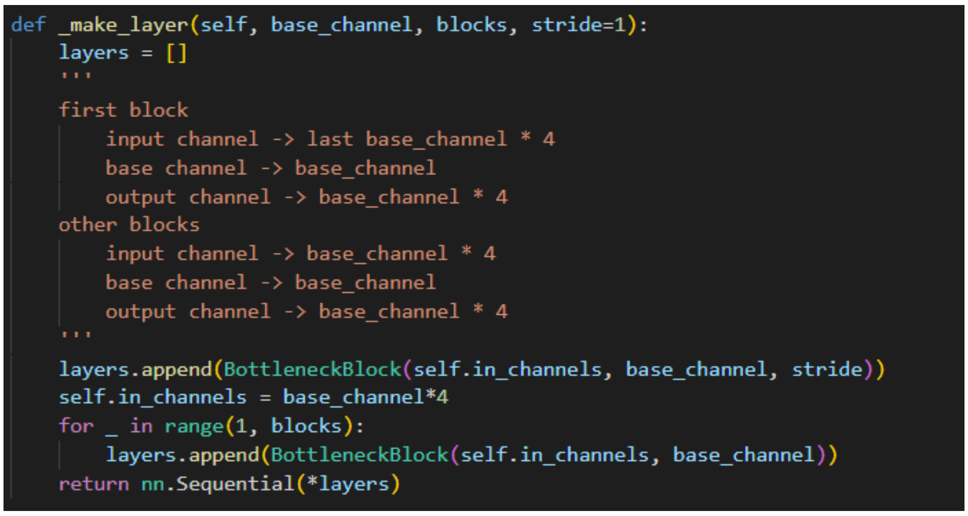
ResNet50

Bottleneck block



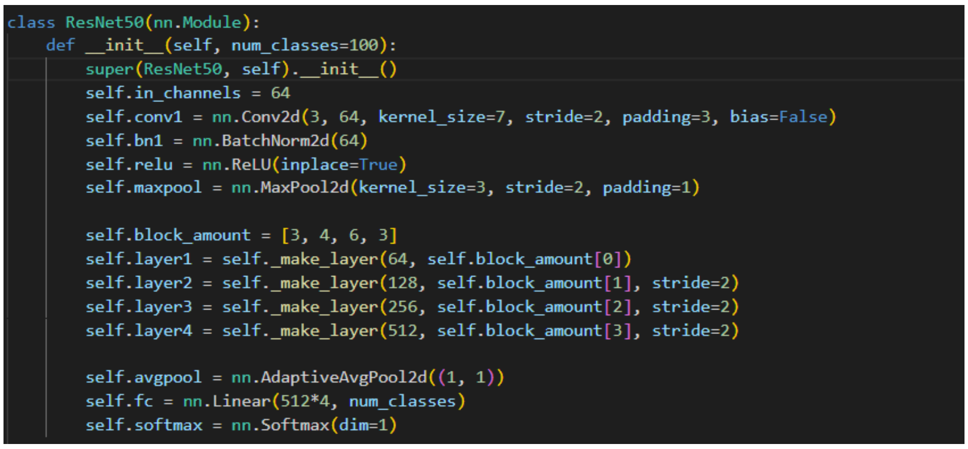
```
class BottleneckBlock(nn.Module):
    def __init__(self, in_channels, base_channels, stride=1):
        super(BottleneckBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, base_channels, kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(base_channels)
        self.relu1 = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(base_channels, base_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(base_channels)
        self.relu2 = nn.ReLU(inplace=True)
        self.conv3 = nn.Conv2d(base_channels, base_channels*4, kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(base_channels*4)
        self.relu3 = nn.ReLU(inplace=True)
        self.downsample = None
        if stride != 1 or in_channels != base_channels*4:
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channels, base_channels*4, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(base_channels*4)
            )

    def forward(self, x):
        x_in = x
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu1(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = self.relu2(x)
        x = self.conv3(x)
        x = self.bn3(x)
        if self.downsample is not None:
            x_in = self.downsample(x_in)
        x += x_in
        x = self.relu3(x)
        return x
```

Stack bottleneck blocks

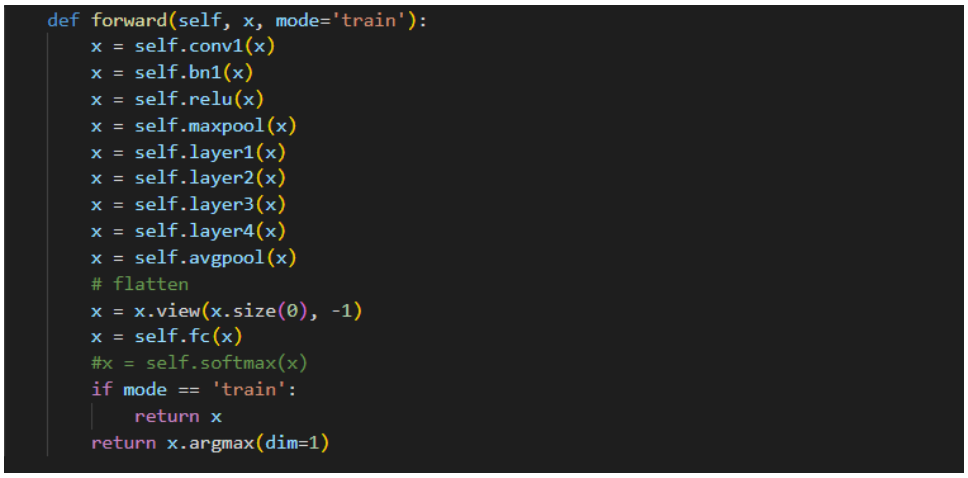


```
def _make_layer(self, base_channel, blocks, stride=1):
    layers = []
    '''
    first block
        input channel -> last base_channel * 4
        base channel -> base_channel
        output channel -> base_channel * 4
    other blocks
        input channel -> base_channel * 4
        base channel -> base_channel
        output channel -> base_channel * 4
    '''
    layers.append(BottleneckBlock(self.in_channels, base_channel, stride))
    self.in_channels = base_channel*4
    for _ in range(1, blocks):
        layers.append(BottleneckBlock(self.in_channels, base_channel))
    return nn.Sequential(*layers)
```

Stack the layers

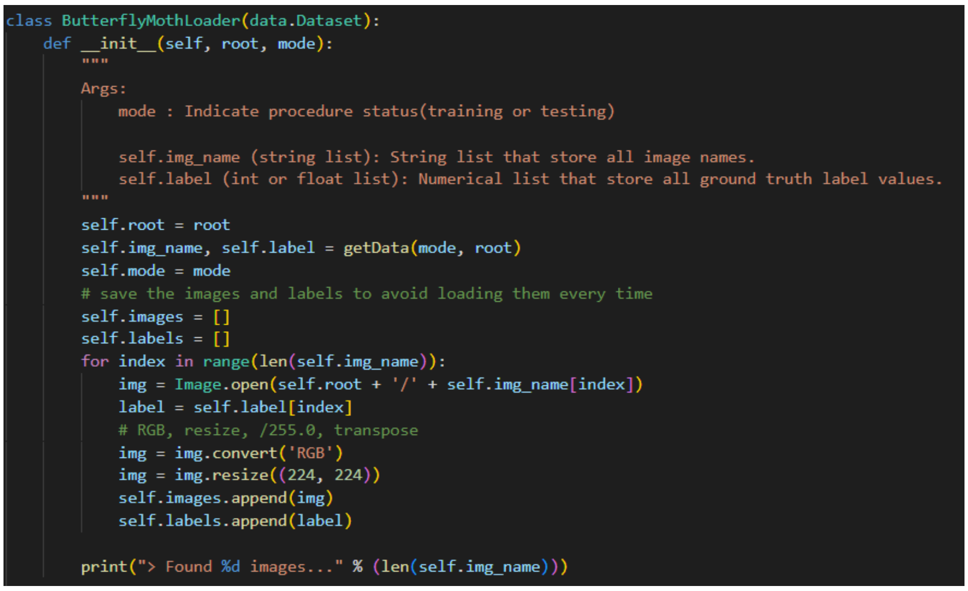


```
class ResNet50(nn.Module):
    def __init__(self, num_classes=100):
        super(ResNet50, self).__init__()
        self.in_channels = 64
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.block_amount = [3, 4, 6, 3]
        self.layer1 = self._make_layer(64, self.block_amount[0])
        self.layer2 = self._make_layer(128, self.block_amount[1], stride=2)
        self.layer3 = self._make_layer(256, self.block_amount[2], stride=2)
        self.layer4 = self._make_layer(512, self.block_amount[3], stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512*4, num_classes)
        self.softmax = nn.Softmax(dim=1)
```

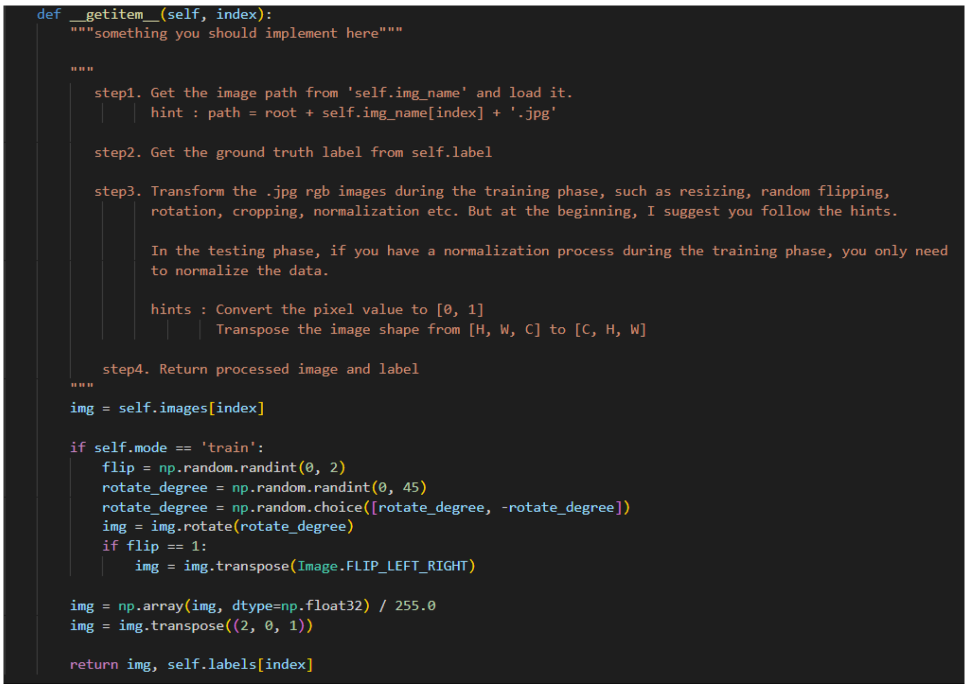


```
    def forward(self, x, mode='train'):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.avgpool(x)
        # flatten
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        #x = self.softmax(x)
        if mode == 'train':
            return x
        return x.argmax(dim=1)
```

B. The details of your Dataloader



```
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, root)
        self.mode = mode
        # save the images and labels to avoid loading them every time
        self.images = []
        self.labels = []
        for index in range(len(self.img_name)):
            img = Image.open(self.root + '/' + self.img_name[index])
            label = self.label[index]
            # RGB, resize, /255.0, transpose
            img = img.convert('RGB')
            img = img.resize((224, 224))
            self.images.append(img)
            self.labels.append(label)

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



```python
def __getitem__(self, index):
    """something 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 = self.images[index]

    if self.mode == 'train':
        flip = np.random.randint(0, 2)
        rotate_degree = np.random.randint(0, 45)
        rotate_degree = np.random.choice([rotate_degree, -rotate_degree])
        img = img.rotate(rotate_degree)
        if flip == 1:
            img = img.transpose(Image.FLIP_LEFT_RIGHT)

    img = np.array(img, dtype=np.float32) / 255.0
    img = img.transpose((2, 0, 1))

    return img, self.labels[index]
```

Data Preprocessing

A. How you preprocessed your data?

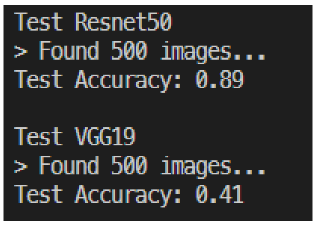
I convert the images into rgb, resize them to (224, 224), /255.0 to make the values between 0 to 1, and transpose it to [C, H, W]. For the training data, I randomly rotate and transpose the image.

B. What makes your method special?

The rotated and transposed images can be viewed as the image of same label. With data augmentation, the datasize increased significantly, making the robust to small rotations. Before I do these two things, the model overfits, having traning accuracy 0.95 and validation accuracy 0.7. After augmentation, the validation accuracy increased to 88%.

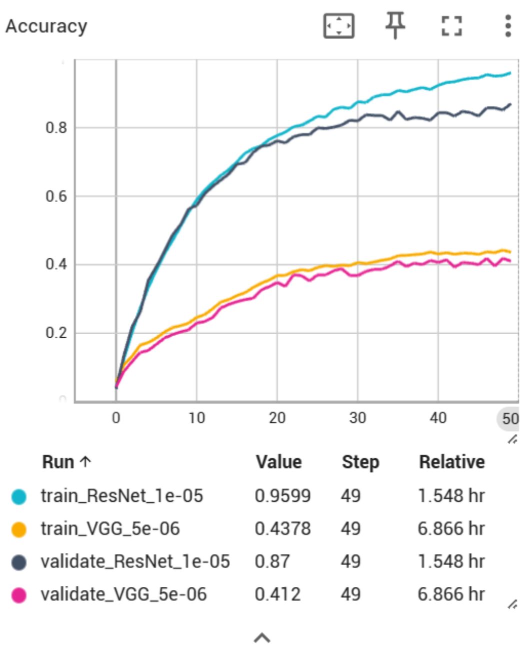
Experimental results

A. The highest testing accuracy



```
Test Resnet50
> Found 500 images...
Test Accuracy: 0.89

Test VGG19
> Found 500 images...
Test Accuracy: 0.41
```

B. Comparison figures



Run ↑	Value	Step	Relative
● train_ResNet_1e-05	0.9599	49	1.548 hr
● train_VGG_5e-06	0.4378	49	6.866 hr
● validate_ResNet_1e-05	0.87	49	1.548 hr
● validate_VGG_5e-06	0.412	49	6.866 hr

Discussion

A. Softmax Function

At first, I put a softmax function after the last layer. This will make the output logit interpretable, but I found out that it does not help when calculating the loss. According to CrossEntropyLoss documentation, the input logit does not have to be normalized, so I disable the softmax function for both models, and it did make the model converges faster.

B. Learning Rate

I thought higher learning rates will make the model converge faster, so I tried learning rate 0.001 for ResNet at first. The models achieved 95% accuracy on the training set at about 1000 epochs, and I thought the learning rate is too small so that the model converges too slow. I tuned the learning rate between 0.1 and 0.001, and they all converge slow. After a few trials, I tried an absolutely small values for learning rate – 1e-7, and the model actually converge in 50 epochs, which is quite surprising.