Problem 1: -10 (missing problem 1.4 and VGG feature visualizations)

Problem 2: -14 (-1 labels 2.1, -3 no evidence that you widened the network 2.3, -10 missing 2.5 and 2.6)

Problem 3: -12, (-2) No label mapping done. (-10) No training done.

Assignment 3

Chih-Hao(Andy) Tsai

October 2024

1 Problem 1

1.1 Original Image versus Translated Image

I download an image (1920*2880) from Pinterest and translate it to the same size, but 1 pixel to the left. Then I calculate l_1, Mean Absolute Error (MAB), and l_2, Mean Square Error (MSE). And here's the result:

```
Loss 1 (MAE): 109.51044071903935
Loss 2 (MSE): 37.839024944540895
```

Figure 1: The result of l_1 is about 109.74, and the result of l_2 is approximately 37.79.

Figure 2: Original image (resolution: 1920×2880)

Figure 3: Translated image, which is the original image 1 pixel to the left.

1.2 VGG-16

VGG-16 consists of 13 2-D convolutional layers (CNN) and 3 layers of Fully Connected layers (FC). In the structure of VGG-16, there are 5 layers of Max Polling interspersed within the CNN as well as several layers of linear and nonlinear transformations.

VGG16 Model Summary:

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 244, 244]	1,792
ReLU-2	[-1, 64, 244, 244]	0
Conv2d-3	[-1, 64, 244, 244]	36,928
ReLU-4	[-1, 64, 244, 244]	0
MaxPool2d-5	[-1, 64, 122, 122]	0
Conv2d-6	[-1, 128, 122, 122]	73,856
ReLU-7	[-1, 128, 122, 122]	0
Conv2d-8	[-1, 128, 122, 122]	147,584
ReLU-9	[-1, 128, 122, 122]	0
MaxPool2d-10	[-1, 128, 61, 61]	0
Conv2d-11	[-1, 256, 61, 61]	295,168
ReLU-12	[-1, 256, 61, 61]	0
Conv2d-13	[-1, 256, 61, 61]	590,080
ReLU-14	[-1, 256, 61, 61]	0
Conv2d-15	[-1, 256, 61, 61]	590,080
ReLU-16	[-1, 256, 61, 61]	0
MaxPool2d-17	[-1, 256, 30, 30]	0
Conv2d-18	[-1, 512, 30, 30]	1,180,160
ReLU-19	[-1, 512, 30, 30]	0
Conv2d-20	[-1, 512, 30, 30]	2,359,808
ReLU-21	[-1, 512, 30, 30]	0
Conv2d-22	[-1, 512, 30, 30]	2,359,808
ReLU-23	[-1, 512, 30, 30]	0
MaxPool2d-24	[-1, 512, 15, 15]	0
Conv2d-25	[-1, 512, 15, 15]	2,359,808
ReLU-26	[-1, 512, 15, 15]	0
Conv2d-27	[-1, 512, 15, 15]	2,359,808
ReLU-28	[-1, 512, 15, 15]	0
Conv2d-29	[-1, 512, 15, 15]	2,359,808
ReLU-30	[-1, 512, 15, 15]	0
MaxPool2d-31	[-1, 512, 7, 7]	0
AdaptiveAvgPool2d-32	[-1, 512, 7, 7]	0
Linear-33	[-1, 4096]	102,764,544
ReLU-34	[-1, 4096]	0
Dropout-35	[-1, 4096]	0
Linear-36	[-1, 4096]	16,781,312
ReLU-37	[-1, 4096]	0
Dropout-38	[-1, 4096]	0
Linear-39	[-1, 1000]	4,097,000

Figure 4: Architecture of VGG-16, 13 layers of CNN and 3 layers of fully connected layers.

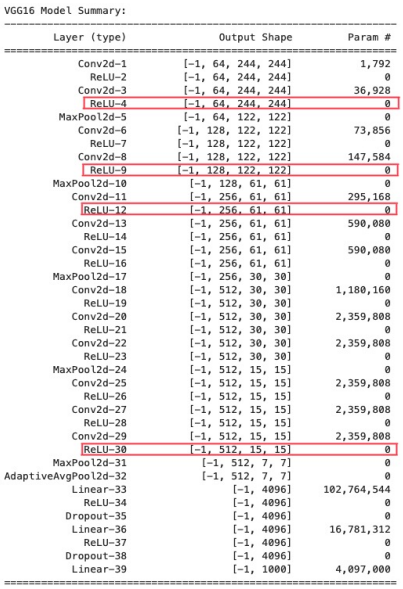

Figure 5: 4 different layers to extract features.

- Pick different places to extract feature.
  To better observe the change in feature space, I picked layer 4, 9, 12, 30 of VGG-16. Most of the calculated loss are way higher than l_1 and l_2 except for layer 30.

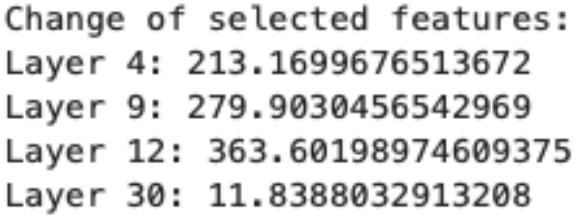

Figure 6: Except for layer 30, loss in other layers are at least 2 times higher than l_1, and at least 5 times higher than l_2.

- Reasons for my pick.
  - Layer 4: I choose this layer to represent earlier stage of the network, which is theoretically capturing features that people can conceive, such as edges.
  - Layer 9: This layer is meant to observe more abstract features as well as to observe feature changes before Max pooling.
  - Layer 12: By combining the result of Layer 9 and this layer, we can observe the impact of Max Pooling on feature changes.
  - Layer 30: This layer is the last CNN before fully connected layer. Thus, I choose this layer to observe the feature changes over multiple layers of CNN and Max Pooling.

- Why choosing *Layer 30* to implement perceptual loss?
  Theoretically, Layer 30 should capture more abstract and global features than other layers because image goes through more CNN and Max Pooling layers. Therefore, Layer 30 is less sensitive to small changes in the image, which means Layer 30 is less sensitive to noises.

1.3 Test the robustness of Layer 30

In this section, I add Gaussian Noise into both test image and translated image.

- Qualitative analysis:

Figure 7: The difference between original image (left) and noise image after adding into Gaussian Noise (right).

Figure 8: The difference between original image (left) and noise image after adding into Gaussian Noise (right).

- Quantitative analysis
  First, I calculate l_1 and l_2 of both original image and translated image versus their own noise image. Also, I calculate their perceptual loss, the result is surprisingly high compare to their l_2 loss.

```
l_1 of original image and noise image: 175.98707658179012
l_1 of translated image and noise image: 175.99215187355324

l_2 of original image and noise image: 102.51400836709105
l_2 of translated image and noise image: 102.52369978539738

Perceptual loss of original image: 134.18960571289062
Perceptual loss of translated image: 133.22518920898438
```

Figure 9: Perceptual loss between original image and its noise image is about 30 higher than their l_2. The same pattern can be seen between translated image and its noise image.

2 Problem 2

2.1 Summarize *Oxford Flowers 102 dataset*

In this section, I summarized Oxford Flowers 102 dataset using different features, including total number of images, total number of classes, total number of different resolutions. Furthermore, I plot some of the resolutions correspondent to their number as shown in figure 11. Also, I randomly select 5 different images from different categories along with their labels demonstrated in figure 12.

	Images	Classes	Different Resolutions
Total Number	1,020	102	363

Figure 10: Summarized details of Oxford Flowers 102 dataset.

Resolution	Number of Images
(667, 500)	163
(752, 500)	84
(666, 500)	51
(500, 667)	33
(750, 500)	30
(500, 500)	12
(625, 500)	12
(626, 500)	11
(667, 501)	11
(500, 752)	10
(668, 500)	9
(500, 666)	8
(500, 750)	8
(500, 625)	6
(751, 500)	6
(600, 500)	5
(624, 500)	5
(731, 500)	5
(500, 505)	4
(500, 607)	4
(701, 500)	4
(662, 500)	4
(627, 500)	4
(542, 500)	4
(528, 500)	4
(500, 527)	4
(535, 500)	4
(679, 500)	4
(500, 751)	4
(672, 500)	4
(749, 500)	4
(754, 500)	4
(755, 500)	4
(500, 538)	3
(659, 500)	3
(592, 500)	3
(532, 500)	3
(647, 500)	3
(692, 500)	3
(718, 500)	3
(537, 500)	3

Figure 11: Brief glance of chart of different resolution correspondent to its number of image.

Figure 12: 5 different classes in Oxford Flowers 102 dataset.

2.2 Train *EfficientNetV2-S* with training set of *Oxford Flowers 102 dataset*

In this sub-problem, I implement training session on the training data in 'train' set of Oxford Flowers 102. Also, I summarize the structure of EfficientNetV2-S model and took a snapshot as shown in figure 13. From the output, we can see that the training accuracy and test accuracy are really low.

```
Epoch [1/2], Loss: 5.2506, Accuracy: 1.18%
Epoch [2/2], Loss: 4.2428, Accuracy: 2.94%
Training time: 12.06 seconds
Test Accuracy: 1.46%
Total parameters: 22103832


Last 7 lines of torchsummary:
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 318.69
Params size (MB): 84.32
Estimated Total Size (MB): 403.59
----------------------------------------------------------------
```

Figure 13: The output of training and last 7 lines of summary.

2.3 Training on modified EfficientNetV2-S model.

A modified version of the EfficientNetV2-S model is implemented in the training session. I multiply 1.1 times to each layers as can be seen in figure 14. As in subproblem 2-2, the last 7 lines of summary of modified structure as well as the number of its parameters. As can be seen in figure 15, the training accuracy and the test accuracy still remain under 3%.

```
class ModifiedEfficientNetV2S(nn.Module):
    def __init__(self):
        super(ModifiedEfficientNetV2S, self).__init__()
        # Get the original architecture
        self.model = efficientnet_v2_s(pretrained=False)

        # Widen the neural network by multiplying 1.1 to the number output channel of each layers
        self.model.features[2][0].out_channels = int(self.model.features[2][0].out_channels * 1.1)
        self.model.features[3][0].out_channels = int(self.model.features[3][0].out_channels * 1.1)
        self.model.features[4][0].out_channels = int(self.model.features[4][0].out_channels * 1.1)
        self.model.features[5][0].out_channels = int(self.model.features[5][0].out_channels * 1.1)


    def forward(self, x):
        return self.model(x)
```

Figure 14: The number of nodes in each layers is being multiplied by 1.1.

```
Epoch [1/2], Loss: 5.6629, Accuracy: 1.86%
Epoch [2/2], Loss: 4.5136, Accuracy: 2.25%
Training time: 20.73 seconds
Test Accuracy: 2.21%
Total parameters: 21458488


Last 7 lines of torchsummary:
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 322.34
Params size (MB): 81.86
Estimated Total Size (MB): 404.78
----------------------------------------------------------------
```

Figure 15: The output of modified structure.

2.4 Training on 2 A100 GPUs

In this section, I modified the code to run training session on 2 NVIDIA A100 GPUs. Surprisingly, the output percentage of both training accuracy and test accuracy improved a little. Also, the training time decreased to under 15 seconds.

```
Epoch [1/2], Loss: 5.5195, Accuracy: 1.27%
Epoch [2/2], Loss: 4.3819, Accuracy: 3.24%
Training time: 14.52 seconds
Test Accuracy: 7.32%
Total parameters: 21458488


Last 7 lines of torchsummary:
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 322.35
Params size (MB): 81.86
Estimated Total Size (MB): 404.78
----------------------------------------------------------------
```

Figure 16: The output of training on 2 NVIDIA A100 GPUs.

2.5 My own images augmentation

I created a method that randomly covers the image with a noise block, where the size and location of the block are also randomly selected 17. To my surprise, this image augmentation method does not help increase the resulting accuracy (figure 18).

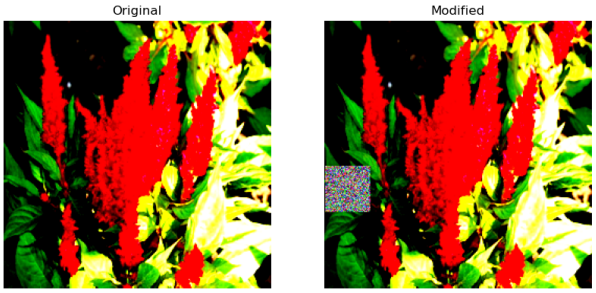

Figure 17: An example image of my image augmentation method.

```
Epoch [1/2], Loss: 5.6647, Accuracy: 0.78%
Epoch [2/2], Loss: 4.4445, Accuracy: 2.25%
Training time: 3355.90 seconds
Test Accuracy: 2.31%
```

Figure 18: The testing accuracy does not increase even implementing image augmentation.

2.6 Implementing Grad-CAM

In this section, I implemented Grad-CAM in my trained model. As my expectation, the result of predicting labels of selected images is not really good. I tried over 100 images, and nearly all of them are incorrect. I believe the main reason is because the testing accuracy of my model is pretty low (only 2.21%). The result can be seen in figure 19.

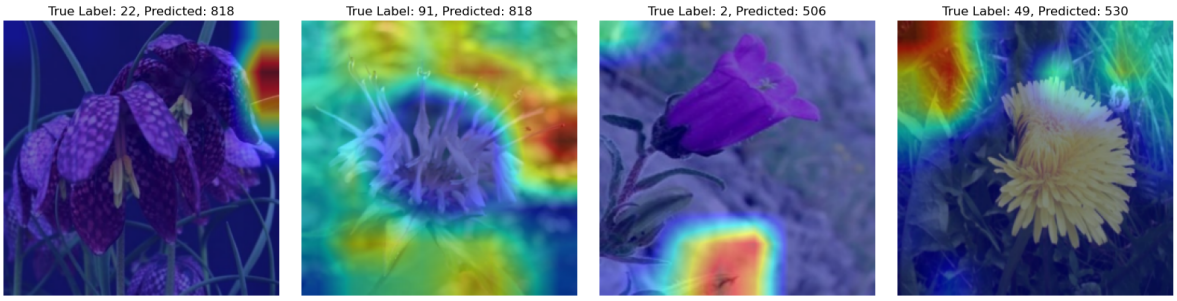

Figure 19: Nearly all of the images are predicted wrong with my model.

3 Problem 3

3.1 Create Prototype Dataset

	Total Number
Number of Classes	500
Images per Class	600
Total Images	187200
Training Set Size	131039
Validation Set Size	28080
Test Set Size	28081

Figure 20: A catalog of my prototype dataset.

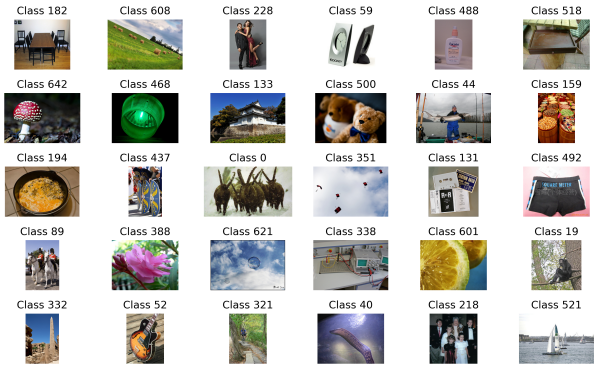

Figure 21: Few samples with their labels in my prototype dataset.

In this section, I randomly select 500 classes from ImageNet, and randomly pick 600 images for each class. The catalog of my prototype dataset is shown in figure 20.

3.2 Design 36-layer Resnet model

In this sub-problem, I modify the architecture of popular Resnet-34. First, I add one more Convolutional layer before move into 4 modified residual layers. As to the structure of residual blocks, I add in one more layer of Convolutional layer and one more layer of batch normalization to each residual block, instead of only 2 convolutional layers and 2 batch normalization layers for residual bloks in Resnet-34. In the structure, I use 4 different residual layers, the first 3 layers have 3 residual blocks in each layers, and the last residual layer have only 2 residual blocks. Before output layer, the same as Resnet-34, I also use a fully connected layer.

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 220, 220]	9,408
BatchNorm2d-2	[-1, 64, 220, 220]	128
Conv2d-3	[-1, 128, 108, 108]	401,408
BatchNorm2d-4	[-1, 128, 108, 108]	256
MaxPool2d-5	[-1, 128, 54, 54]	0

Figure 22: 2 convolutional layers and 2 batch normalization layers before enetering residual layers.

Layer 1 Summary:

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 128, 54, 54]	147,456
BatchNorm2d-2	[-1, 128, 54, 54]	256
Conv2d-3	[-1, 128, 54, 54]	147,456
BatchNorm2d-4	[-1, 128, 54, 54]	256
Conv2d-5	[-1, 128, 54, 54]	147,456
BatchNorm2d-6	[-1, 128, 54, 54]	256
ResidualBlock-7	[-1, 128, 54, 54]	0
Conv2d-8	[-1, 128, 54, 54]	147,456
BatchNorm2d-9	[-1, 128, 54, 54]	256
Conv2d-10	[-1, 128, 54, 54]	147,456
BatchNorm2d-11	[-1, 128, 54, 54]	256
Conv2d-12	[-1, 128, 54, 54]	147,456
BatchNorm2d-13	[-1, 128, 54, 54]	256
ResidualBlock-14	[-1, 128, 54, 54]	0
Conv2d-15	[-1, 128, 54, 54]	147,456
BatchNorm2d-16	[-1, 128, 54, 54]	256
Conv2d-17	[-1, 128, 54, 54]	147,456
BatchNorm2d-18	[-1, 128, 54, 54]	256
Conv2d-19	[-1, 128, 54, 54]	147,456
BatchNorm2d-20	[-1, 128, 54, 54]	256
ResidualBlock-21	[-1, 128, 54, 54]	0

Figure 23: First residual layer of my Resnet-36, which has 3 residual blocks.

Layer 4 Summary:

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 1024, 7, 7]	4,718,592
BatchNorm2d-2	[-1, 1024, 7, 7]	2,048
Conv2d-3	[-1, 1024, 7, 7]	9,437,184
BatchNorm2d-4	[-1, 1024, 7, 7]	2,048
Conv2d-5	[-1, 1024, 7, 7]	9,437,184
BatchNorm2d-6	[-1, 1024, 7, 7]	2,048
Conv2d-7	[-1, 1024, 7, 7]	524,288
BatchNorm2d-8	[-1, 1024, 7, 7]	2,048
ResidualBlock-9	[-1, 1024, 7, 7]	0
Conv2d-10	[-1, 1024, 7, 7]	9,437,184
BatchNorm2d-11	[-1, 1024, 7, 7]	2,048
Conv2d-12	[-1, 1024, 7, 7]	9,437,184
BatchNorm2d-13	[-1, 1024, 7, 7]	2,048
Conv2d-14	[-1, 1024, 7, 7]	9,437,184
BatchNorm2d-15	[-1, 1024, 7, 7]	2,048
ResidualBlock-16	[-1, 1024, 7, 7]	0

Figure 24: The last residual layer of my Resnet-36.

3.3 Customize activation function

In this sub-problem, I create my own activation function:

$$f(x) = \begin{cases} x^2 & \text{if } x > 0 \\ 0.2x & \text{if } x < 0 \\ 0 & \text{if } x = 0 \end{cases}$$

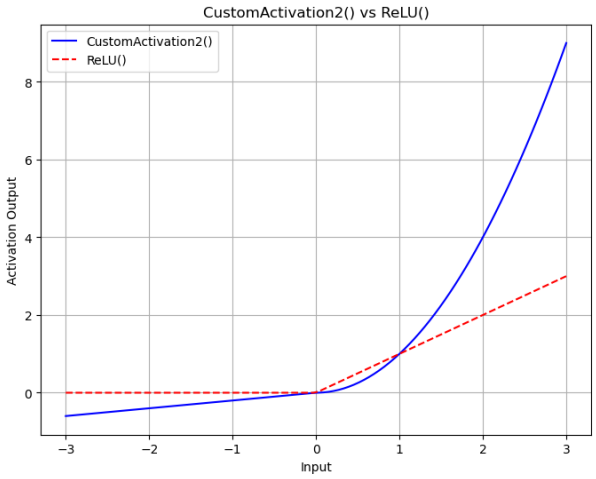

Figure 25: My activation function vs ReLU

3.4 Training result of training ImageNET

```
Batch 14700/18213 - Loss: nan, Accuracy: 0.15%
Batch 14800/18213 - Loss: nan, Accuracy: 0.15%
Batch 14900/18213 - Loss: nan, Accuracy: 0.15%
Batch 15000/18213 - Loss: nan, Accuracy: 0.15%
Batch 15100/18213 - Loss: nan, Accuracy: 0.15%
Batch 15200/18213 - Loss: nan, Accuracy: 0.15%
Batch 15300/18213 - Loss: nan, Accuracy: 0.15%
Batch 15400/18213 - Loss: nan, Accuracy: 0.15%
Batch 15500/18213 - Loss: nan, Accuracy: 0.15%
Batch 15600/18213 - Loss: nan, Accuracy: 0.15%
Batch 15700/18213 - Loss: nan, Accuracy: 0.15%
Batch 15800/18213 - Loss: nan, Accuracy: 0.15%
Batch 15900/18213 - Loss: nan, Accuracy: 0.15%
Batch 16000/18213 - Loss: nan, Accuracy: 0.15%
Batch 16100/18213 - Loss: nan, Accuracy: 0.15%
Batch 16200/18213 - Loss: nan, Accuracy: 0.15%
Batch 16300/18213 - Loss: nan, Accuracy: 0.15%
Batch 16400/18213 - Loss: nan, Accuracy: 0.15%
Batch 16500/18213 - Loss: nan, Accuracy: 0.15%
Batch 16600/18213 - Loss: nan, Accuracy: 0.15%
Batch 16700/18213 - Loss: nan, Accuracy: 0.15%
Batch 16800/18213 - Loss: nan, Accuracy: 0.15%
Batch 16900/18213 - Loss: nan, Accuracy: 0.15%
Batch 17000/18213 - Loss: nan, Accuracy: 0.15%
Batch 17100/18213 - Loss: nan, Accuracy: 0.15%
Batch 17200/18213 - Loss: nan, Accuracy: 0.15%
Batch 17300/18213 - Loss: nan, Accuracy: 0.15%
Batch 17400/18213 - Loss: nan, Accuracy: 0.15%
Batch 17500/18213 - Loss: nan, Accuracy: 0.15%
Batch 17600/18213 - Loss: nan, Accuracy: 0.15%
Batch 17700/18213 - Loss: nan, Accuracy: 0.15%
Batch 17800/18213 - Loss: nan, Accuracy: 0.15%
Batch 17900/18213 - Loss: nan, Accuracy: 0.15%
Batch 18000/18213 - Loss: nan, Accuracy: 0.15%
Batch 18100/18213 - Loss: nan, Accuracy: 0.15%
Batch 18200/18213 - Loss: nan, Accuracy: 0.15%
Train Loss: nan, Train Accuracy: 0.15%
Test Loss: nan, Test Accuracy: 0.16%
Validation Loss: nan, Validation Accuracy: 0.17%
Epoch 10/10
```

Figure 26: The training result of training ResNet34.

Figure 27: The training result of training my own model (ResNet36).