Deep Learning Course

Assignment 2

Assignment Goals

- Design and implementation of CNNs.
- CNN visualization.
- Implementation of ResNet.

In this assignment, you will be asked to learn CNN models for an image dataset. Different experiments will help you achieve a better understanding of CNNs.

Dataset

The dataset consists of around 9K images (some grayscale and some RGB) belonging to 101 classes. The shape of each image is (64,64,3). Every image is labeled with one of the classes. The image file is contained in the folder named after the class name.

Requirements

1. **(40 points) Implement and improve a CNN model.**

   (a) We are aiming to learn a CNN on the given dataset. Download the dataset, and use PyTorch to implement LeNet5 to classify instances. Use a one-hot encoding for labels. Split the dataset into training (90 percent) and validation (10 percent) and report the model loss (cross-entropy) and accuracy on both training and validation sets. (20 points)

   The LeNet5 configuration is:

   - Convolutional layer (kernel size 5 x 5, 32 filters, stride 1 x 1 and followed by ReLU)
   - Max Pooling layer with size 4 and stride 4 x 4
   - Convolutional layer (kernel size 5 x 5, 64 filters, stride 1 x 1 and followed by ReLU)
   - Max Pooling layer with size 4 and stride 4 x 4
   - Fully Connected ReLU layer that has 1021 neurons
   - Fully Connected ReLU layer with 84 neurons
   - Fully Connected Softmax layer that has input 84 and output which is equal to the number of classes (one node for each of the classes).

   (b) Try to improve model accuracy on the validation dataset by tuning the model hyperparameters. You can use any improvement methods you prefer. You are expected to reach at least 65 percent accuracy on validation set. (20 points)

   Here are some improvement methods you can use, of course you can use others which are not mentioned here:

- Dropout
- L1, L2 regularization
- Try improved initialization (e.g., Xavier initialier)
- Batch Normalization

The grading of part (b) is based on the correctness of your implementation (5 points) and the performance of your improvement on the validation set. The validation accuracy and corresponding score is:

- 65% (5 points)
- 67% (8 points)
- 69% (12 points)
- 71% (15 points)

Structure of LENET-5

This following LENET-5 structure is for 10-class dataset. Therefore, the layer size is not exactly the same as ours.

hw2_lenet5.png

1. **(20 points) Visualize layer activation**

   There are several approaches to understand and visualize convolutional Networks, including visualizing the activations and layers weights. The most straight-forward visualization technique is to show the activations of the network during the forward pass. The second most common strategy is to visualize the weights. For more information we recommend the course notes on "Visualizing what ConvNets learn". More advanced techniques can be found in "Visualizing and Understanding Convolutional Networks" paper by Matthew D.Zeiler and Rob Fergus.

   Please visualize the layer activation of **the first conv layer** and **the second conv layer** of your above CNN model (after completing Q1), on the following 2 images:

   - accordion/image_0001
   - camera/image_0001

   Visualizing a CNN layer activation means to visualize the result of the activation layer as an image. Specifically, the activation of the first conv layer is the output of the first (conv + ReLU) layer during forward propagation. Since we have 32 filters in the first conv layer, you should draw 32 activation images for the first conv layer. Please display multiple images side by side in a row to make your output more readable (Hint: matplotlib.pyplot.subplot).

1. **(40 points) ResNet Implementation**

   Use PyTorch to implement ResNet 18 to classify the given dataset. Same as above, please use a one-hot encoding for labels, split the dataset into training (90 percent) and validation (10 percent) and report the model loss (cross-entropy) and accuracy on both training and validation sets. See the paper Deep Residual Learning for Image Recognition for detailed introduction of ResNet.

The grading of this part is mainly based on the implementation and performance on validation set. If you need more resources to complete the training, consider using Google Colab.

The ResNet 18 configuration is:

- conv_1 (kernel size 7 x 7, 64 filters, stride 2 x 2)
- conv_2 (max pooling layer with size 3 x 3, followed by 2 blocks.Each block contains two conv layers. Each conv layer has kernel size 3 x 3, 64 filters, stride 2 x 2)
- conv_3 (2 blocks, each contains 2 conv layers with kernel size 3*3, 128 filters)
- conv_4 (2 blocks, each contains 2 conv layers with kernel size 3*3, 256 filters)
- conv_5 (2 blocks, each contains 2 conv layers with kernel size 3*3, 512 filters)

A block has the structure:

hw2_resnet.png

Submission Notes

Please use Jupyter Notebook. The notebook should include the final code, results and your answers. You should submit your Notebook in (.pdf or .html) and .ipynb format. (penalty 10 points)

Your Implementation

In [1]:
```
# You can use the following helper functions

from typing import Any
from torch.utils.data import Dataset, DataLoader
import pandas as pd
import os
from torchvision.io import read_image
from torchvision import transforms
from sklearn.preprocessing import OneHotEncoder
import numpy as np
import torch
from torch import nn
from tqdm import tqdm
from matplotlib import pyplot as plt
import torch.optim as optim
from PIL import Image
import torchvision.transforms.functional as TF
```

In [2]:
```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

In [3]:
```
class ImageDataset( Dataset ):

    def __init__(self, is_val= False, transform = None) -> None:


        if is_val:
            self.df = pd.read_csv( 'validation.csv', index_col=0 )
```

```
        else:
            self.df = pd.read_csv( 'train.csv', index_col= 0 )

        self.cls_names = self.df['cls_name'].unique().tolist()
        self.df['label'] = self.df['cls_name'].apply( self.cls_names.index )

        self.transform = transform


    def get_num_classes(self):
        return len( self.cls_names )

    def __len__(self):
        return len( self.df )

    def __getitem__(self, index):
        path = self.df.iloc[index]['path']
        img = read_image( path ).type( torch.float32 )

        target = self.df.iloc[index]['label']

        if self.transform is not None:
            img = self.transform( img )

        target = torch.tensor( target )
        one_hot_target=torch.zeros(101,dtype=torch.float32)
        one_hot_target[target]=1.0

        return img/255 , one_hot_target

def collate_fn( batch ):
    imgs, targets = [], []

    for img, target in batch:
        imgs.append( img )
        targets.append( target )

    imgs = torch.stack( imgs, dim= 0 )
    targets = torch.stack( targets, dim= 0 )
    return imgs, targets
```

In [4]:
```
num_epochs = 50
batch_size = 64
learning_rate = 0.0001
weight_decay = 0.00001
number_of_class=101
```

In [5]:
```
transform = transforms.Compose([
    #transforms.Normalize( (0.485, 0.456, 0.406), (0.229, 0.224, 0.225) ),
    transforms.RandomVerticalFlip( .5 )
])

train_dataset = ImageDataset( is_val = False, transform = transform )
val_dataset = ImageDataset( is_val = True )

train_dataloader = DataLoader( train_dataset, batch_size = batch_size, shuffle= Tru
val_dataloader = DataLoader( val_dataset, batch_size = batch_size, shuffle= True, c
```

In [6]:
```
def init_weights( m ):
    if isinstance(m, nn.Linear):
        torch.nn.init.xavier_uniform_(m.weight)
        m.bias.data.fill_(0.01)
```

Implement and improve a CNN model

In [7]:
```
def accuracy(model,dataloader,device):
  correct=0
  total=0
  with torch.no_grad():
    for inputs,labels in dataloader:
      inputs,labels= inputs.to(device), labels.to(device)
      outputs=model(inputs)
      _, pred=torch.max(outputs.data, 1)
      _, labels_indices=torch.max(labels,1)
      total+=labels.size(0)
      correct+=(pred==labels_indices).sum().item()

  accuracy=correct/total*100
  return accuracy
```

In [8]:
```
# implement your Lenet5 here

class CNN_LeNet5(nn.Module):
  def __init__(self):
    super(CNN_LeNet5,self).__init__()
    self.conv1=nn.Conv2d(3,32,kernel_size=5, stride=1)
    self.pool1=nn.MaxPool2d(kernel_size=4, stride=4)
    self.conv2=nn.Conv2d(32,64,kernel_size=5,stride=1)
    self.pool2=nn.MaxPool2d(kernel_size=4, stride=4)
    self.linear=nn.Sequential(
      nn.Flatten(),
      nn.Linear(2*2*64, 1021),
      nn.ReLU(),

      nn.Linear(1021,84),
      nn.ReLU(),
      nn.Linear(84,number_of_class),
      nn.Softmax(dim=1)
    )
    self.ReLU=nn.ReLU()
  def forward(self,x):
    x=self.ReLU(self.conv1(x))
    x=self.pool1(x)
    x=self.ReLU(self.conv2(x))
    x=self.pool2(x)
    x=self.linear(x)
    return x

model=CNN_LeNet5().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
for epoch in range(num_epochs):
    model.train()  # Set the model to training mode
    running_loss = 0.0
    total_samples = 0

    for inputs, labels in train_dataloader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss+= loss.item() * inputs.size(0)
        total_samples+= inputs.size(0)
```

```
    epoch_loss = running_loss/total_samples
    train_accuracy = accuracy(model,train_dataloader, device)
    print(f"Epoch [{epoch+1}/{num_epochs}], trainLoss: {epoch_loss:.4f},trainAccura

    model.eval()
    running_loss = 0.0
    total_samples = 0
    for inputs, labels in val_dataloader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        running_loss+= loss.item() * inputs.size(0)
        total_samples+= inputs.size(0)

    epoch_loss = running_loss/total_samples
    val_accuracy = accuracy(model,val_dataloader, device)
    print(f"Epoch [{epoch+1}/{num_epochs}], ValidationLoss: {epoch_loss:.4f},Valida
```

```
Epoch [1/50], trainLoss: 4.5644,trainAccuracy: 9.2454%
Epoch [1/50], ValidationLoss: 4.5442,ValidationAccuracy: 8.7816%
Epoch [2/50], trainLoss: 4.5395,trainAccuracy: 9.2454%
Epoch [2/50], ValidationLoss: 4.5437,ValidationAccuracy: 8.7816%
Epoch [3/50], trainLoss: 4.5320,trainAccuracy: 11.3443%
Epoch [3/50], ValidationLoss: 4.5222,ValidationAccuracy: 10.8672%
Epoch [4/50], trainLoss: 4.5041,trainAccuracy: 13.3402%
Epoch [4/50], ValidationLoss: 4.5042,ValidationAccuracy: 12.9528%
Epoch [5/50], trainLoss: 4.4982,trainAccuracy: 13.5205%
Epoch [5/50], ValidationLoss: 4.5015,ValidationAccuracy: 12.9528%
Epoch [6/50], trainLoss: 4.4948,trainAccuracy: 15.8769%
Epoch [6/50], ValidationLoss: 4.4821,ValidationAccuracy: 15.6970%
Epoch [7/50], trainLoss: 4.4591,trainAccuracy: 19.3793%
Epoch [7/50], ValidationLoss: 4.4515,ValidationAccuracy: 18.4413%
Epoch [8/50], trainLoss: 4.4355,trainAccuracy: 20.2807%
Epoch [8/50], ValidationLoss: 4.4385,ValidationAccuracy: 19.7585%
Epoch [9/50], trainLoss: 4.4264,trainAccuracy: 20.6928%
Epoch [9/50], ValidationLoss: 4.4314,ValidationAccuracy: 20.0878%
Epoch [10/50], trainLoss: 4.4249,trainAccuracy: 21.1692%
Epoch [10/50], ValidationLoss: 4.4283,ValidationAccuracy: 20.4171%
Epoch [11/50], trainLoss: 4.4231,trainAccuracy: 21.1434%
Epoch [11/50], ValidationLoss: 4.4275,ValidationAccuracy: 20.3074%
Epoch [12/50], trainLoss: 4.4213,trainAccuracy: 21.2851%
Epoch [12/50], ValidationLoss: 4.4264,ValidationAccuracy: 20.4171%
Epoch [13/50], trainLoss: 4.4206,trainAccuracy: 21.1821%
Epoch [13/50], ValidationLoss: 4.4252,ValidationAccuracy: 20.3074%
Epoch [14/50], trainLoss: 4.4185,trainAccuracy: 21.8517%
Epoch [14/50], ValidationLoss: 4.4239,ValidationAccuracy: 21.1855%
Epoch [15/50], trainLoss: 4.4194,trainAccuracy: 20.9503%
Epoch [15/50], ValidationLoss: 4.4274,ValidationAccuracy: 20.4171%
Epoch [16/50], trainLoss: 4.4147,trainAccuracy: 21.8259%
Epoch [16/50], ValidationLoss: 4.4210,ValidationAccuracy: 20.9660%
Epoch [17/50], trainLoss: 4.4140,trainAccuracy: 22.1736%
Epoch [17/50], ValidationLoss: 4.4187,ValidationAccuracy: 21.4050%
Epoch [18/50], trainLoss: 4.4087,trainAccuracy: 22.6114%
Epoch [18/50], ValidationLoss: 4.4201,ValidationAccuracy: 21.5148%
Epoch [19/50], trainLoss: 4.4043,trainAccuracy: 23.3325%
Epoch [19/50], ValidationLoss: 4.4104,ValidationAccuracy: 22.5027%
Epoch [20/50], trainLoss: 4.3985,trainAccuracy: 23.3840%
Epoch [20/50], ValidationLoss: 4.4075,ValidationAccuracy: 22.7223%
Epoch [21/50], trainLoss: 4.3948,trainAccuracy: 23.9634%
Epoch [21/50], ValidationLoss: 4.4049,ValidationAccuracy: 22.7223%
Epoch [22/50], trainLoss: 4.3913,trainAccuracy: 24.3111%
Epoch [22/50], ValidationLoss: 4.4058,ValidationAccuracy: 22.7223%
Epoch [23/50], trainLoss: 4.3903,trainAccuracy: 24.6330%
Epoch [23/50], ValidationLoss: 4.4002,ValidationAccuracy: 23.4907%
Epoch [24/50], trainLoss: 4.3873,trainAccuracy: 24.5429%
Epoch [24/50], ValidationLoss: 4.4021,ValidationAccuracy: 23.2711%
Epoch [25/50], trainLoss: 4.3868,trainAccuracy: 24.8519%
Epoch [25/50], ValidationLoss: 4.3966,ValidationAccuracy: 23.8200%
Epoch [26/50], trainLoss: 4.3854,trainAccuracy: 25.0579%
Epoch [26/50], ValidationLoss: 4.3965,ValidationAccuracy: 23.7102%
Epoch [27/50], trainLoss: 4.3832,trainAccuracy: 25.1481%
Epoch [27/50], ValidationLoss: 4.3980,ValidationAccuracy: 23.6004%
Epoch [28/50], trainLoss: 4.3833,trainAccuracy: 25.3670%
Epoch [28/50], ValidationLoss: 4.3954,ValidationAccuracy: 23.8200%
Epoch [29/50], trainLoss: 4.3829,trainAccuracy: 24.9421%
Epoch [29/50], ValidationLoss: 4.3997,ValidationAccuracy: 23.2711%
Epoch [30/50], trainLoss: 4.3829,trainAccuracy: 25.3670%
Epoch [30/50], ValidationLoss: 4.3945,ValidationAccuracy: 23.8200%
Epoch [31/50], trainLoss: 4.3796,trainAccuracy: 25.3412%
Epoch [31/50], ValidationLoss: 4.4004,ValidationAccuracy: 23.3809%
Epoch [32/50], trainLoss: 4.3800,trainAccuracy: 25.4829%
Epoch [32/50], ValidationLoss: 4.4008,ValidationAccuracy: 23.1614%
```

```
Epoch [33/50], trainLoss: 4.3801,trainAccuracy: 25.5344%
Epoch [33/50], ValidationLoss: 4.3949,ValidationAccuracy: 23.9297%
Epoch [34/50], trainLoss: 4.3810,trainAccuracy: 25.6760%
Epoch [34/50], ValidationLoss: 4.3942,ValidationAccuracy: 23.8200%
Epoch [35/50], trainLoss: 4.3785,trainAccuracy: 25.3155%
Epoch [35/50], ValidationLoss: 4.3980,ValidationAccuracy: 23.2711%
Epoch [36/50], trainLoss: 4.3780,trainAccuracy: 25.5988%
Epoch [36/50], ValidationLoss: 4.3935,ValidationAccuracy: 23.8200%
Epoch [37/50], trainLoss: 4.3773,trainAccuracy: 25.3670%
Epoch [37/50], ValidationLoss: 4.3928,ValidationAccuracy: 23.8200%
Epoch [38/50], trainLoss: 4.3795,trainAccuracy: 25.5730%
Epoch [38/50], ValidationLoss: 4.3943,ValidationAccuracy: 23.8200%
Epoch [39/50], trainLoss: 4.3770,trainAccuracy: 25.6631%
Epoch [39/50], ValidationLoss: 4.3925,ValidationAccuracy: 24.1493%
Epoch [40/50], trainLoss: 4.3761,trainAccuracy: 25.6245%
Epoch [40/50], ValidationLoss: 4.3932,ValidationAccuracy: 23.8200%
Epoch [41/50], trainLoss: 4.3770,trainAccuracy: 25.2511%
Epoch [41/50], ValidationLoss: 4.3972,ValidationAccuracy: 23.3809%
Epoch [42/50], trainLoss: 4.3760,trainAccuracy: 25.7790%
Epoch [42/50], ValidationLoss: 4.3945,ValidationAccuracy: 23.7102%
Epoch [43/50], trainLoss: 4.3740,trainAccuracy: 25.5086%
Epoch [43/50], ValidationLoss: 4.3974,ValidationAccuracy: 23.6004%
Epoch [44/50], trainLoss: 4.3752,trainAccuracy: 25.9722%
Epoch [44/50], ValidationLoss: 4.3899,ValidationAccuracy: 24.1493%
Epoch [45/50], trainLoss: 4.3748,trainAccuracy: 25.7404%
Epoch [45/50], ValidationLoss: 4.3935,ValidationAccuracy: 23.9297%
Epoch [46/50], trainLoss: 4.3740,trainAccuracy: 25.8048%
Epoch [46/50], ValidationLoss: 4.3907,ValidationAccuracy: 24.3688%
Epoch [47/50], trainLoss: 4.3736,trainAccuracy: 26.0752%
Epoch [47/50], ValidationLoss: 4.3898,ValidationAccuracy: 24.4786%
Epoch [48/50], trainLoss: 4.3728,trainAccuracy: 25.8563%
Epoch [48/50], ValidationLoss: 4.3912,ValidationAccuracy: 23.9297%
Epoch [49/50], trainLoss: 4.3734,trainAccuracy: 26.4615%
Epoch [49/50], ValidationLoss: 4.3875,ValidationAccuracy: 24.5884%
Epoch [50/50], trainLoss: 4.3711,trainAccuracy: 26.3327%
Epoch [50/50], ValidationLoss: 4.3872,ValidationAccuracy: 24.6981%
```

In [9]:

```
num_epochs = 35
batch_size = 64
learning_rate = 0.001
weight_decay = 0.00001
number_of_class=101
# implement the improved of LeNet5:
class improve_of(nn.Module):
  def __init__(self):
    super(improve_of,self).__init__()
    self.layer1=nn.Sequential(
        nn.Conv2d(3,32,kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(32),
        nn.ReLU(),
        # nn.MaxPool2d(kernel_size=2)
    )
    self.layer2=nn.Sequential(
        nn.Conv2d(32,64,kernel_size=3,stride=1,padding=1),
        nn.BatchNorm2d(64),
        nn.ReLU(),
        # nn.MaxPool2d(kernel_size=4)
    )
    self.layer3=nn.Sequential(
        nn.Conv2d(64,128,kernel_size=3,stride=1,padding=1),
        nn.BatchNorm2d(128),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=4)
    )
```

```
        self.layer4=nn.Sequential(
            nn.Conv2d(128,256,kernel_size=3,stride=1,padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=4)

        )

        self.layer5=nn.Sequential(
            nn.Conv2d(256,128,kernel_size=3,stride=1,padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)

        )

        # )
#         self.layer7=nn.Sequential(
#             nn.Conv2d(128,128,kernel_size=3,stride=1,padding=1),
#             nn.BatchNorm2d(128),
#             nn.ReLU(),
#             nn.MaxPool2d(kernel_size=2)

#         )
#         self.layer8=nn.Sequential(
#             nn.Conv2d(128,64,kernel_size=3,stride=1,padding=1),
#             nn.BatchNorm2d(64),
#             nn.ReLU(),
#             nn.MaxPool2d(kernel_size=2)

#         )
        self.linear=nn.Sequential(
          nn.Flatten(),
          nn.Linear(2*2*128, 128),
          nn.ReLU(),

          # nn.Linear(256,512),
          # nn.ReLU(),

          # nn.Linear(512,256),
          # nn.ReLU(),

          # nn.Linear(256,128),
          # nn.ReLU(),

          nn.Linear(128,number_of_class)
        )
    def forward(self,x):
        x=self.layer1(x)
        x=self.layer2(x)
        x=self.layer3(x)
        x=self.layer4(x)
        x=self.layer5(x)
        # x=self.layer6(x)
        # x=self.layer7(x)
        # x=self.layer8(x)
        x=self.linear(x)
        return x

model=improve_of().to(device)
model.apply(init_weights)
criterion=nn.CrossEntropyLoss()
optimizer=optim.Adam(model.parameters(), lr=learning_rate,weight_decay=weight_decay
```

```
for epoch in range(num_epochs):
    model.train()  # Set the model to training mode
    running_loss = 0.0
    total_samples = 0

    for inputs, labels in train_dataloader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss+= loss.item() * inputs.size(0)
        total_samples+= inputs.size(0)

    epoch_loss = running_loss/total_samples
    train_accuracy = accuracy(model,train_dataloader, device)
    print(f"Epoch [{epoch+1}/{num_epochs}], trainLoss: {epoch_loss:.4f},trainAccura

    model.eval()
    running_loss = 0.0
    total_samples = 0
    for inputs, labels in val_dataloader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        running_loss+= loss.item() * inputs.size(0)
        total_samples+= inputs.size(0)

    epoch_loss = running_loss/total_samples
    val_accuracy = accuracy(model,val_dataloader, device)
    print(f"Epoch [{epoch+1}/{num_epochs}], ValidationLoss: {epoch_loss:.4f},Valida
```

```
Epoch [1/35], trainLoss: 3.3303,trainAccuracy: 41.3598%
Epoch [1/35], ValidationLoss: 2.9887,ValidationAccuracy: 36.9923%
Epoch [2/35], trainLoss: 2.4479,trainAccuracy: 52.4723%
Epoch [2/35], ValidationLoss: 2.2976,ValidationAccuracy: 47.3106%
Epoch [3/35], trainLoss: 1.8892,trainAccuracy: 60.2369%
Epoch [3/35], ValidationLoss: 1.9906,ValidationAccuracy: 52.2503%
Epoch [4/35], trainLoss: 1.5199,trainAccuracy: 67.4994%
Epoch [4/35], ValidationLoss: 1.7810,ValidationAccuracy: 57.5192%
Epoch [5/35], trainLoss: 1.2597,trainAccuracy: 73.1908%
Epoch [5/35], ValidationLoss: 1.5417,ValidationAccuracy: 61.8002%
Epoch [6/35], trainLoss: 1.0815,trainAccuracy: 76.9379%
Epoch [6/35], ValidationLoss: 1.5147,ValidationAccuracy: 62.5686%
Epoch [7/35], trainLoss: 0.9143,trainAccuracy: 80.6979%
Epoch [7/35], ValidationLoss: 1.4667,ValidationAccuracy: 64.9835%
Epoch [8/35], trainLoss: 0.7742,trainAccuracy: 84.3291%
Epoch [8/35], ValidationLoss: 1.3574,ValidationAccuracy: 65.8617%
Epoch [9/35], trainLoss: 0.6500,trainAccuracy: 87.2006%
Epoch [9/35], ValidationLoss: 1.3422,ValidationAccuracy: 65.9715%
Epoch [10/35], trainLoss: 0.5599,trainAccuracy: 89.4798%
Epoch [10/35], ValidationLoss: 1.3397,ValidationAccuracy: 67.6180%
Epoch [11/35], trainLoss: 0.4622,trainAccuracy: 90.9477%
Epoch [11/35], ValidationLoss: 1.3578,ValidationAccuracy: 68.1668%
Epoch [12/35], trainLoss: 0.3727,trainAccuracy: 93.1496%
Epoch [12/35], ValidationLoss: 1.3455,ValidationAccuracy: 67.5082%
Epoch [13/35], trainLoss: 0.2908,trainAccuracy: 94.7463%
Epoch [13/35], ValidationLoss: 1.3100,ValidationAccuracy: 69.5939%
Epoch [14/35], trainLoss: 0.2338,trainAccuracy: 96.5362%
Epoch [14/35], ValidationLoss: 1.3140,ValidationAccuracy: 70.9111%
Epoch [15/35], trainLoss: 0.2040,trainAccuracy: 96.6392%
Epoch [15/35], ValidationLoss: 1.3642,ValidationAccuracy: 68.8255%
Epoch [16/35], trainLoss: 0.1613,trainAccuracy: 97.3989%
Epoch [16/35], ValidationLoss: 1.3759,ValidationAccuracy: 70.2525%
Epoch [17/35], trainLoss: 0.1107,trainAccuracy: 98.6480%
Epoch [17/35], ValidationLoss: 1.3987,ValidationAccuracy: 70.0329%
Epoch [18/35], trainLoss: 0.0898,trainAccuracy: 98.4677%
Epoch [18/35], ValidationLoss: 1.4618,ValidationAccuracy: 68.4962%
Epoch [19/35], trainLoss: 0.0817,trainAccuracy: 98.8411%
Epoch [19/35], ValidationLoss: 1.3670,ValidationAccuracy: 71.5697%
Epoch [20/35], trainLoss: 0.0591,trainAccuracy: 98.7896%
Epoch [20/35], ValidationLoss: 1.4863,ValidationAccuracy: 70.0329%
Epoch [21/35], trainLoss: 0.0793,trainAccuracy: 99.4077%
Epoch [21/35], ValidationLoss: 1.4194,ValidationAccuracy: 71.4599%
Epoch [22/35], trainLoss: 0.0605,trainAccuracy: 99.4077%
Epoch [22/35], ValidationLoss: 1.4164,ValidationAccuracy: 71.0209%
Epoch [23/35], trainLoss: 0.0501,trainAccuracy: 99.2016%
Epoch [23/35], ValidationLoss: 1.4910,ValidationAccuracy: 70.8013%
Epoch [24/35], trainLoss: 0.0320,trainAccuracy: 99.7038%
Epoch [24/35], ValidationLoss: 1.4843,ValidationAccuracy: 71.2404%
Epoch [25/35], trainLoss: 0.0292,trainAccuracy: 99.4978%
Epoch [25/35], ValidationLoss: 1.4893,ValidationAccuracy: 71.4599%
Epoch [26/35], trainLoss: 0.0521,trainAccuracy: 98.6995%
Epoch [26/35], ValidationLoss: 1.5691,ValidationAccuracy: 69.4841%
Epoch [27/35], trainLoss: 0.0616,trainAccuracy: 97.1800%
Epoch [27/35], ValidationLoss: 1.8249,ValidationAccuracy: 66.3008%
Epoch [28/35], trainLoss: 0.1702,trainAccuracy: 94.3987%
Epoch [28/35], ValidationLoss: 1.7576,ValidationAccuracy: 67.7278%
Epoch [29/35], trainLoss: 0.1703,trainAccuracy: 96.7551%
Epoch [29/35], ValidationLoss: 1.7241,ValidationAccuracy: 68.9352%
Epoch [30/35], trainLoss: 0.1172,trainAccuracy: 97.8625%
Epoch [30/35], ValidationLoss: 1.5987,ValidationAccuracy: 68.8255%
Epoch [31/35], trainLoss: 0.0512,trainAccuracy: 99.6395%
Epoch [31/35], ValidationLoss: 1.5187,ValidationAccuracy: 70.9111%
Epoch [32/35], trainLoss: 0.0342,trainAccuracy: 99.5236%
Epoch [32/35], ValidationLoss: 1.5887,ValidationAccuracy: 71.3502%
```

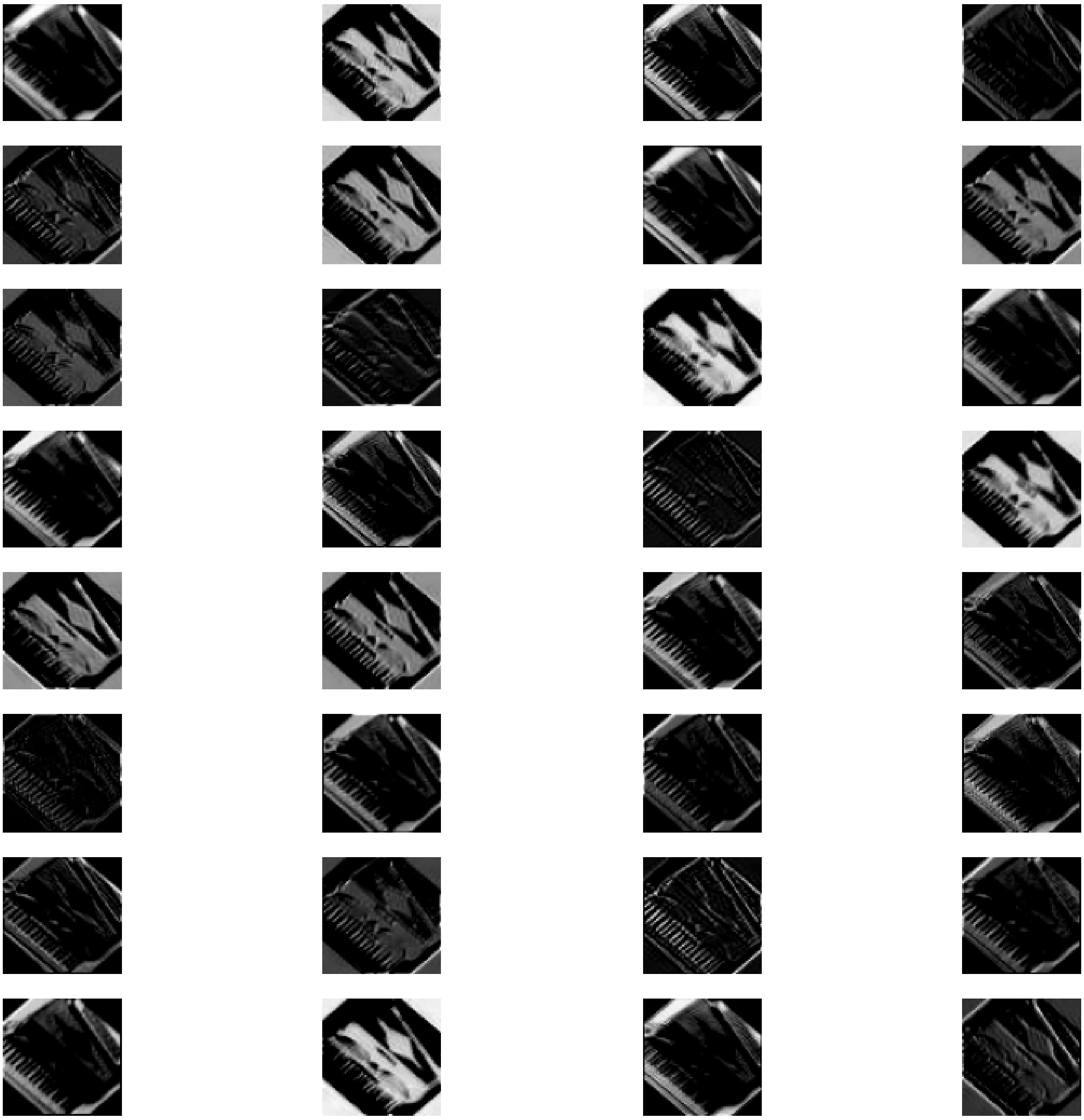
```
Epoch [33/35], trainLoss: 0.0197,trainAccuracy: 99.7296%
Epoch [33/35], ValidationLoss: 1.4902,ValidationAccuracy: 71.4599%
Epoch [34/35], trainLoss: 0.0113,trainAccuracy: 99.9227%
Epoch [34/35], ValidationLoss: 1.5301,ValidationAccuracy: 70.8013%
Epoch [35/35], trainLoss: 0.0118,trainAccuracy: 99.9485%
Epoch [35/35], ValidationLoss: 1.5078,ValidationAccuracy: 72.6674%
```

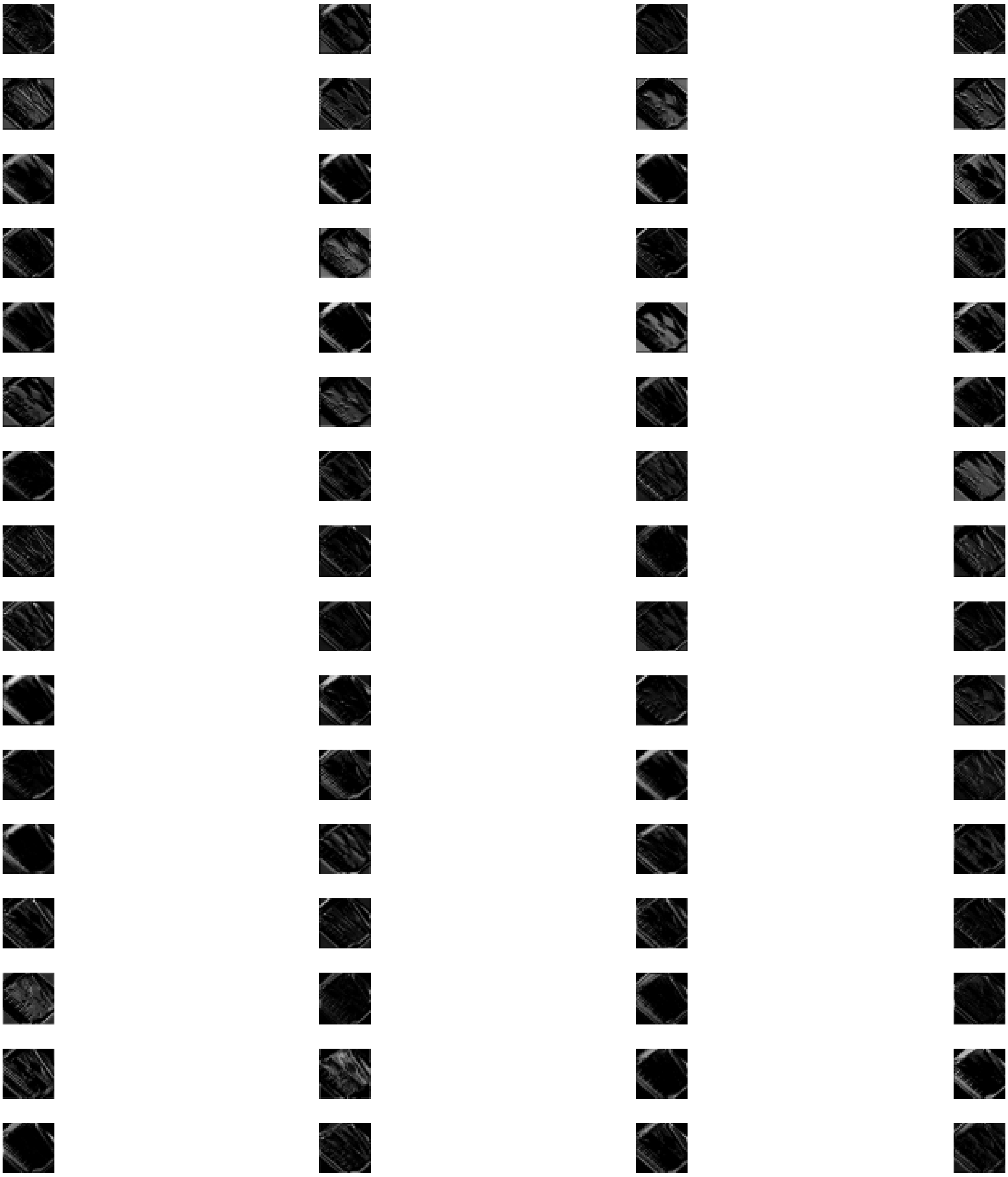
Visualize layer activation

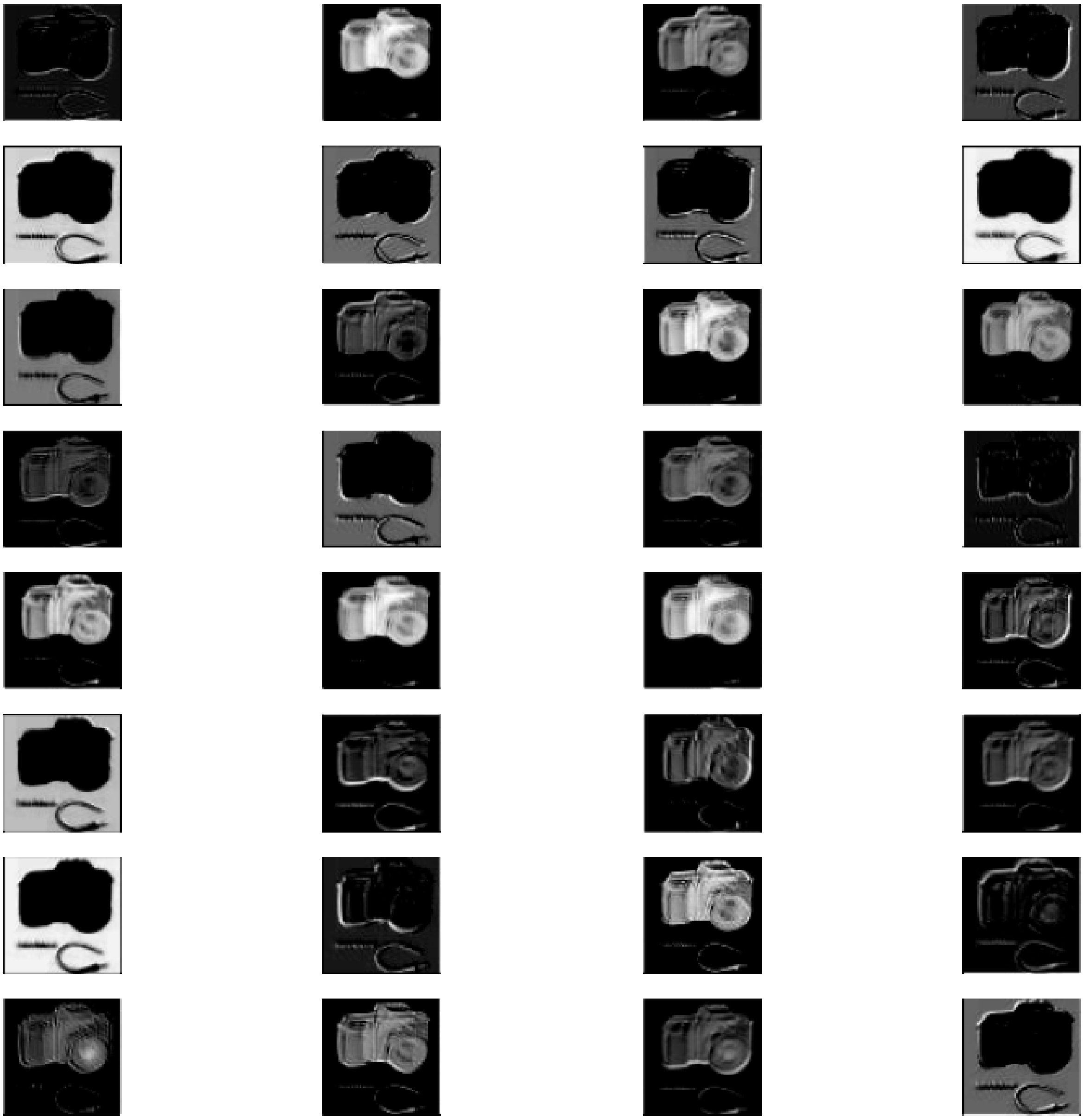
In [11]:

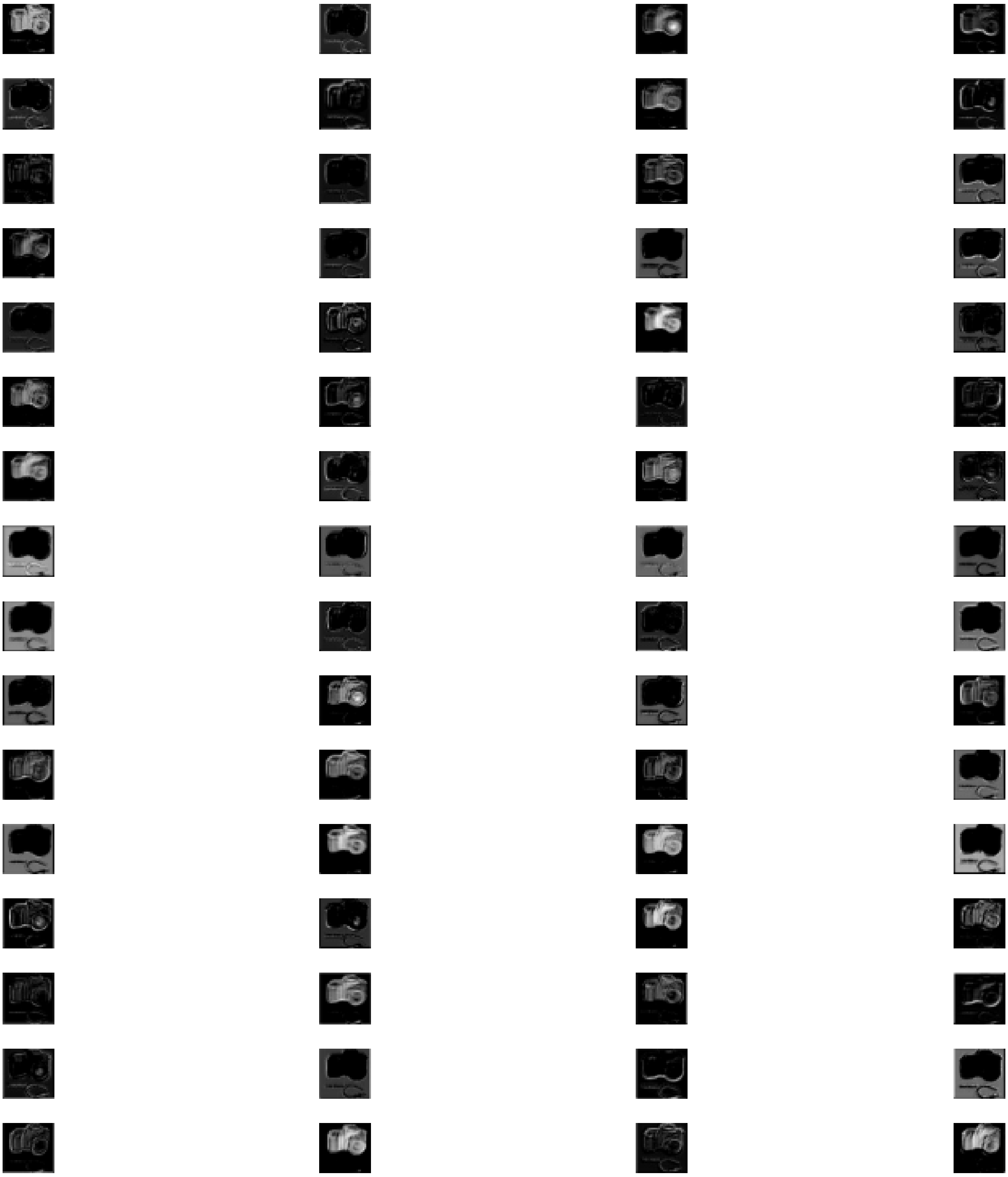
```
# implement your visualization here
def get_activation(model,input_image):
  activation=[]
  def hook(model,input,output):
    activation.append(output)
  hook_handle1 = model.layer1.register_forward_hook(hook)
  hook_handle2 = model.layer2.register_forward_hook(hook)
  model(input_image)
  for act in activation:
    num_channels = act.size(1)
    num_rows = num_channels//4+1
    plt.figure(figsize=(8,8))
    for j in range(num_channels):
        plt.subplot(num_rows, 4, j + 1)
        plt.imshow(act[0, j].detach().numpy(), cmap='gray')
        plt.axis('off')
    plt.tight_layout()
    plt.show
  hook_handle1.remove()
  hook_handle2.remove()

image_p1='accordion/image_0001.jpg'
image_p2='camera/image_0001.jpg'
image1=Image.open(f'Dataset/train/{image_p1}')
image2=Image.open(f'Dataset/train/{image_p2}')
x1=TF.to_tensor(image1)
x1.unsqueeze_(0)
x2=TF.to_tensor(image2)
x2.unsqueeze_(0)
get_activation(improve_of().cpu(),x1)
get_activation(improve_of().cpu(),x2)
```









In [12]:
```
num_epochs = 30
batch_size = 32
learning_rate = 0.001

weight_decay = 0.00001
number_of_class=101

# transform = transforms.Compose([
#     transforms.Resize(224),
# ])
# val_transform = transforms.Compose([
#     transforms.Resize(224),
# ])
t_d = ImageDataset( is_val = False)
v_d = ImageDataset( is_val = True)

train_loader = DataLoader( t_d, batch_size = batch_size, shuffle= True, collate_fn=
val_loader = DataLoader( v_d, batch_size = batch_size, shuffle= True, collate_fn=c
```

ResNet Implementation

In [ ]:
```
# implement a ResNet model here
class ResidualBlock(nn.Module):
    def __init__(self, input, output, stride=1, downsample=None):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(input,output,kernel_size=3,stride=stride,padding=1),
            nn.BatchNorm2d(output),
            nn.ReLU()
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(output,output,kernel_size=3,stride=1,padding=1),
            nn.BatchNorm2d(output)
        )
        self.downsample = downsample
        self.relu = nn.ReLU()

    def forward(self,x):
        residual = x
        result =self.conv1(x)
        result =self.conv2(result)
        if self.downsample is not None:
            residual=self.downsample(x)
        result+=residual
        result=self.relu(result)
        return result

class ResNet(nn.Module):
  def __init__(self, block, layers, number_of_class=101):
    super(ResNet, self).__init__()
    self.in_channels=64
    self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3)
    self.Maxpool = nn.MaxPool2d(kernel_size=3)
    self.layer1 = self._make_layer(block, 64, layers[0],stride=2)
    self.layer2 = self._make_layer(block, 128, layers[1])
    self.layer3 = self._make_layer(block, 256, layers[2])
    self.layer4 = self._make_layer(block, 512, layers[3])
    self.avgpool = nn.AvgPool2d(3, stride=1)
    self.fc = nn.Linear(4608, number_of_class)

  def _make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
        if stride!=1 or self.in_channels != out_channels:
            downsample = nn.Sequential(
                nn.Conv2d(self.in_channels, out_channels, kernel_size=1, stride=str
                nn.BatchNorm2d(out_channels),
            )

        layers = []
        layers.append(block(self.in_channels, out_channels, stride, downsample))
        self.in_channels = out_channels
        for _ in range(1, blocks):
            layers.append(block(self.in_channels, out_channels))

        return nn.Sequential(*layers)
  def forward(self, x):
    x = self.conv1(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 = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
model = ResNet(ResidualBlock, [2, 2, 2, 2], number_of_class).to(device)
model.apply(init_weights)
criterion=nn.CrossEntropyLoss()
optimizer=optim.Adam(model.parameters(), lr=learning_rate,weight_decay=weight_decay
for epoch in range(num_epochs):
    model.train()  # Set the model to training mode
    running_loss = 0.0
    total_samples = 0

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss+= loss.item() * inputs.size(0)
        total_samples+= inputs.size(0)

    epoch_loss = running_loss/total_samples
    train_accuracy = accuracy(model,train_loader, device)
    print(f"Epoch [{epoch+1}/{num_epochs}], trainLoss: {epoch_loss:.4f},trainAccura

    model.eval()
    running_loss = 0.0
    total_samples = 0
    for inputs, labels in val_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        running_loss+= loss.item() * inputs.size(0)
        total_samples+= inputs.size(0)

    epoch_loss = running_loss/total_samples
    val_accuracy = accuracy(model,val_loader, device)
    print(f"Epoch [{epoch+1}/{num_epochs}], ValidationLoss: {epoch_loss:.4f},Valida
```

```
Epoch [1/30], trainLoss: 3.9189,trainAccuracy: 36.1834%
Epoch [1/30], ValidationLoss: 3.0943,ValidationAccuracy: 34.0285%
Epoch [2/30], trainLoss: 2.7107,trainAccuracy: 48.2745%
Epoch [2/30], ValidationLoss: 2.4403,ValidationAccuracy: 44.2371%
Epoch [3/30], trainLoss: 2.1464,trainAccuracy: 56.1937%
Epoch [3/30], ValidationLoss: 2.2412,ValidationAccuracy: 47.6400%
Epoch [4/30], trainLoss: 1.7414,trainAccuracy: 64.4605%
Epoch [4/30], ValidationLoss: 1.9035,ValidationAccuracy: 53.2382%
Epoch [5/30], trainLoss: 1.4424,trainAccuracy: 70.8859%
Epoch [5/30], ValidationLoss: 1.7627,ValidationAccuracy: 58.2876%
Epoch [6/30], trainLoss: 1.1568,trainAccuracy: 76.4229%
Epoch [6/30], ValidationLoss: 1.7019,ValidationAccuracy: 59.4951%
Epoch [7/30], trainLoss: 0.9332,trainAccuracy: 82.4363%
Epoch [7/30], ValidationLoss: 1.6855,ValidationAccuracy: 61.2514%
Epoch [8/30], trainLoss: 0.7451,trainAccuracy: 85.7971%
Epoch [8/30], ValidationLoss: 1.6472,ValidationAccuracy: 60.8123%
Epoch [9/30], trainLoss: 0.5433,trainAccuracy: 91.1151%
Epoch [9/30], ValidationLoss: 1.7219,ValidationAccuracy: 62.7881%
Epoch [10/30], trainLoss: 0.4337,trainAccuracy: 91.7461%
Epoch [10/30], ValidationLoss: 1.7911,ValidationAccuracy: 61.9100%
Epoch [11/30], trainLoss: 0.3292,trainAccuracy: 91.2696%
Epoch [11/30], ValidationLoss: 1.8779,ValidationAccuracy: 63.2272%
Epoch [12/30], trainLoss: 0.2625,trainAccuracy: 94.3214%
Epoch [12/30], ValidationLoss: 1.9566,ValidationAccuracy: 63.8858%
Epoch [13/30], trainLoss: 0.2000,trainAccuracy: 96.7680%
Epoch [13/30], ValidationLoss: 1.9101,ValidationAccuracy: 64.9835%
Epoch [14/30], trainLoss: 0.1793,trainAccuracy: 95.8280%
Epoch [14/30], ValidationLoss: 1.9457,ValidationAccuracy: 64.1054%
Epoch [15/30], trainLoss: 0.1630,trainAccuracy: 95.5189%
Epoch [15/30], ValidationLoss: 2.1339,ValidationAccuracy: 62.7881%
Epoch [16/30], trainLoss: 0.1689,trainAccuracy: 97.5663%
Epoch [16/30], ValidationLoss: 1.9893,ValidationAccuracy: 66.5203%
Epoch [17/30], trainLoss: 0.1094,trainAccuracy: 97.6951%
Epoch [17/30], ValidationLoss: 2.3051,ValidationAccuracy: 65.2031%
Epoch [18/30], trainLoss: 0.0750,trainAccuracy: 98.4033%
Epoch [18/30], ValidationLoss: 2.1598,ValidationAccuracy: 65.0933%
Epoch [19/30], trainLoss: 0.0601,trainAccuracy: 98.1071%
Epoch [19/30], ValidationLoss: 2.2580,ValidationAccuracy: 66.7398%
Epoch [20/30], trainLoss: 0.1289,trainAccuracy: 96.5104%
Epoch [20/30], ValidationLoss: 2.1432,ValidationAccuracy: 65.4226%
Epoch [21/30], trainLoss: 0.0927,trainAccuracy: 98.6608%
Epoch [21/30], ValidationLoss: 2.0734,ValidationAccuracy: 65.9715%
Epoch [22/30], trainLoss: 0.0607,trainAccuracy: 97.9140%
Epoch [22/30], ValidationLoss: 2.0753,ValidationAccuracy: 63.0077%
Epoch [23/30], trainLoss: 0.1159,trainAccuracy: 97.1671%
Epoch [23/30], ValidationLoss: 2.1881,ValidationAccuracy: 63.9956%
Epoch [24/30], trainLoss: 0.1019,trainAccuracy: 97.5534%
Epoch [24/30], ValidationLoss: 2.1143,ValidationAccuracy: 64.8738%
Epoch [25/30], trainLoss: 0.1027,trainAccuracy: 98.0943%
Epoch [25/30], ValidationLoss: 2.1210,ValidationAccuracy: 66.5203%
```