Final_sugarcane

February 20, 2024

[10]:
```
!pip uninstall torch
!rm -rf /Users/peternoble/miniforge3/lib/python3.10/site-packages/torch*
!pip install torch==2.1.2

#!pip install torch==2.1.2
```

```
WARNING: No metadata found in
/Users/peternoble/miniforge3/lib/python3.10/site-packages
Found existing installation: torch 2.1.2
ERROR: Cannot uninstall torch 2.1.2, RECORD file not found. You might be
able to recover from this via: 'pip install --force-reinstall --no-deps
torch==2.1.2'.
Collecting torch==2.1.2
  Using cached torch-2.1.2-cp310-none-macosx_11_0_arm64.whl.metadata (25 kB)
Requirement already satisfied: filelock in
/Users/peternoble/miniforge3/lib/python3.10/site-packages (from torch==2.1.2)
(3.12.2)
Requirement already satisfied: typing-extensions in
/Users/peternoble/miniforge3/lib/python3.10/site-packages (from torch==2.1.2)
(4.9.0)
Requirement already satisfied: sympy in
/Users/peternoble/miniforge3/lib/python3.10/site-packages (from torch==2.1.2)
(1.11.1)
Requirement already satisfied: networkx in
/Users/peternoble/miniforge3/lib/python3.10/site-packages (from torch==2.1.2)
(3.1)
Requirement already satisfied: jinja2 in
/Users/peternoble/miniforge3/lib/python3.10/site-packages (from torch==2.1.2)
(3.1.2)
Requirement already satisfied: fsspec in
/Users/peternoble/miniforge3/lib/python3.10/site-packages (from torch==2.1.2)
(2023.12.2)
Requirement already satisfied: MarkupSafe>=2.0 in
/Users/peternoble/miniforge3/lib/python3.10/site-packages (from
jinja2->torch==2.1.2) (2.1.3)
Requirement already satisfied: mpmath>=0.19 in
```

```
/Users/peternoble/miniforge3/lib/python3.10/site-packages (from
sympy->torch==2.1.2) (1.3.0)
Using cached torch-2.1.2-cp310-none-macosx_11_0_arm64.whl (59.6 MB)
Installing collected packages: torch
Successfully installed torch-2.1.2
```

[ ]:

[2]:
```
import torch                                          # imports the core PyTorch␣
 ↪library
import torch.nn as nn                                 # torch.nn library is a␣
 ↪high-level interface for building and training neural networks
import torch.optim as optim                           # Needed for optimizer
from torch.utils.data import DataLoader               # Needed for loading data␣
 ↪from MNIST png files
from torchvision import datasets, transforms          # Needed for making sure␣
 ↪data from MNIST png files is properly formatted
import torch.nn.functional as F                       # Needed for functional␣
 ↪equation in forward loop
import numpy as np                                    # Needed for outputting␣
 ↪weights and biases
from torchvision.datasets import VisionDataset        # VisionDataset is is␣
 ↪designed to be a base class for datasets in computer vision tasks
from PIL import Image                                 # Needed for manipulating␣
 ↪MNIST png image files
import os                                             # Needed for operating␣
 ↪system for dealing with dir of training and test png files
from torchvision import transforms
from skimage import exposure
import matplotlib.pyplot as plt
from torch.utils.data import Subset
from sklearn.model_selection import train_test_split
from torchvision.datasets.vision import VisionDataset
from torch.utils.data import Subset
from sklearn.model_selection import train_test_split
from PIL import Image
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
import random


# Set the device
device = "mps" if torch.backends.mps.is_available() else "cpu"
# Check PyTorch has access to MPS (Metal Performance Shader, Apple's GPU␣
 ↪architecture)
print(f"Is MPS (Metal Performance Shader) built? {torch.backends.mps.
 ↪is_built()}")
```

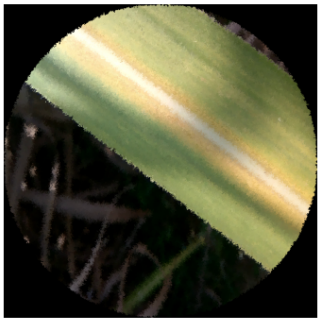
```
Is MPS (Metal Performance Shader) built? True
```

[3]:
```
# List of directories containing images
directories = [
    '/Users/peternoble/Downloads/Sugarcane_true/training/0',
    '/Users/peternoble/Downloads/Sugarcane_true/training/1',
    '/Users/peternoble/Downloads/Sugarcane_true/training/2',
    '/Users/peternoble/Downloads/Sugarcane_true/training/3',
    '/Users/peternoble/Downloads/Sugarcane_true/training/4',
]

# Custom labels
custom_labels = ['Yellow', 'Healthy', 'Rust', 'RedRot', 'Mosaic']

# Create subplots for each image pair
fig, axs = plt.subplots(len(directories), 3, figsize=(15, 5 * len(directories)))

for i, image_dir in enumerate(directories):
    # List all files in the directory
    image_files = [f for f in os.listdir(image_dir) if os.path.isfile(os.path.
 ↪join(image_dir, f))]

    # Choose a random image file
    random_image_file = random.choice(image_files)

    # Load the original image using PIL
    original_image = Image.open(os.path.join(image_dir, random_image_file))

    # Define the transformation for the original image
    original_transform = transforms.Compose([
        transforms.Resize((488, 488)),
        transforms.ToTensor(),
    ])

    # Apply the transformation to the original image for display purposes
    original_display_image = original_transform(original_image)

    # Define the transformation for the transformed image without normalization
    transform_without_norm = transforms.Compose([
        transforms.Resize((488, 488)),
        transforms.CenterCrop((488, 488)),
        transforms.RandomApply([
            transforms.RandomRotation(degrees=(i * 10, (i + 1) * 10)) for i in␣
 ↪range(36)
        ], p=1.0),
        transforms.ToTensor(),
    ])
```

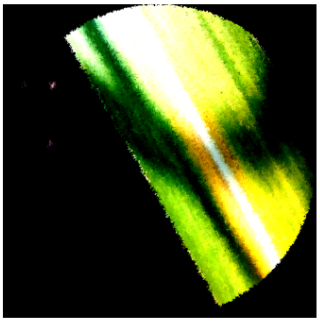
```
    # Apply the transformation to the original image without normalization
    transformed_image_without_norm = transform_without_norm(original_image)

    # Define the transformation for the transformed image with normalization
    transform_with_norm = transforms.Compose([
        transforms.Resize((488, 488)),
        transforms.CenterCrop((488, 488)),
      transforms.RandomApply([
            transforms.RandomRotation(degrees=(i * 10, (i + 1) * 10)) for i in␣
↪range(36)
        ], p=1.0),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.
↪225]),
    ])

    # Apply the transformation to the original image with normalization
    transformed_image_with_norm = transform_with_norm(original_image)

    # Clip pixel values to the valid range
    original_display_image = torch.clamp(original_display_image, 0.0, 1.0)
    transformed_image_without_norm = torch.
↪clamp(transformed_image_without_norm, 0.0, 1.0)
    transformed_image_with_norm = torch.clamp(transformed_image_with_norm, 0.0,␣
↪1.0)

    # Display the original, transformed without normalization, and transformed␣
↪with normalization images
    axs[i, 0].imshow(original_display_image.permute(1, 2, 0))
    axs[i, 0].set_title(f"Class {i} - {custom_labels[i]} - Original")
    axs[i, 0].axis('off')

    axs[i, 1].imshow(transformed_image_without_norm.permute(1, 2, 0))
    axs[i, 1].set_title(f"Class {i} - {custom_labels[i]} - Transformed (No␣
↪Norm)")
    axs[i, 1].axis('off')

    axs[i, 2].imshow(transformed_image_with_norm.permute(1, 2, 0))
    axs[i, 2].set_title(f"Class {i} - {custom_labels[i]} - Transformed (With␣
↪Norm)")
    axs[i, 2].axis('off')

plt.show()
```

Class 0 - Yellow - Original

Class 0 - Yellow - Transformed (No Norm)



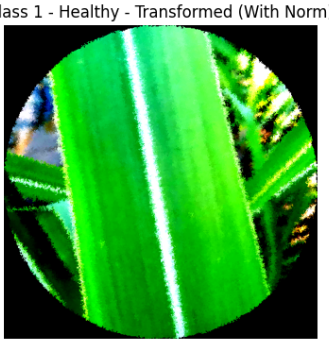
Class 0 - Yellow - Transformed (With Norm)



Class 1 - Healthy - Original

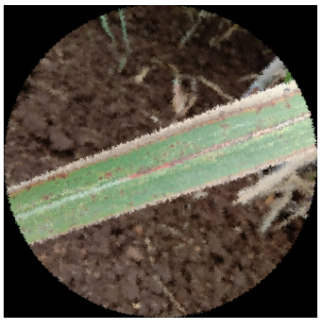
Class 1 - Healthy - Transformed (No Norm)

Class 1 - Healthy - Transformed (With Norm)



Class 2 - Rust - Original

Class 2 - Rust - Transformed (No Norm)



Class 2 - Rust - Transformed (With Norm)

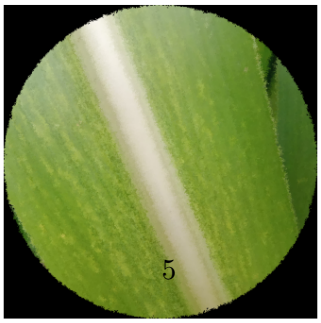
Class 3 - RedRot - Original

Class 3 - RedRot - Transformed (No Norm)

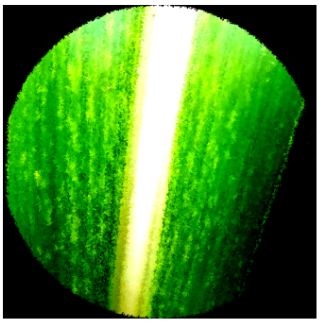
Class 3 - RedRot - Transformed (With Norm)

Class 4 - Mosaic - Original

Class 4 - Mosaic - Transformed (No Norm)



Class 4 - Mosaic - Transformed (With Norm)



[4]:
```
class Sugarcane(VisionDataset):
    def __init__(self, root, split='train', transform=None,␣
 ↪target_transform=None):
        super(Sugarcane, self).__init__(root, transform=transform,␣
 ↪target_transform=target_transform)

        self.split = split
        self.data_folder = "training" if split == 'train' else "testing"
        self.images_folder = os.path.join(self.root, self.data_folder)

        self.image_paths = self._get_image_paths()
        self.mean = np.array([0.4961, 0.5261, 0.3792])
        self.std = np.array([0.1772, 0.1765, 0.1841])

        # Add a check to ensure transform is not None
        if transform is None:
            self.transform = transforms.Compose([
                transforms.Resize((488, 488)),
                transforms.CenterCrop((488, 488)),
                transforms.RandomApply([
                    transforms.RandomRotation(degrees=(i * 10, (i + 1) * 10))␣
 ↪for i in range(36)
                ], p=1.0),
                transforms.ToTensor(),
                transforms.Normalize(mean=self.mean, std=self.std),
            ])
        else:
            self.transform = transform

    def _get_image_paths(self):
        image_paths = []
        for digit_folder in os.listdir(self.images_folder):
            digit_folder_path = os.path.join(self.images_folder, digit_folder)

            # Check if it's a directory before listing its contents
            if os.path.isdir(digit_folder_path):
                for image_name in os.listdir(digit_folder_path):
                    image_path = os.path.join(digit_folder_path, image_name)

                    # Skip files with the '.DS_Store' extension
                    if not image_path.endswith('.DS_Store'):
                        image_paths.append((image_path, int(digit_folder)))

        return image_paths
```

```
  def _split_dataset(self, dataset, train_size=0.7, val_size=0.20,␣
↪test_size=0.10, random_state=None):
      # Extract images and labels from the dataset
      images, labels = zip(*dataset)

      # First, split into train and temp sets
      train_images, temp_images, train_labels, temp_labels = train_test_split(
          images, labels, test_size=(val_size + test_size),␣
↪random_state=random_state
      )

      # Second, split temp set into validation and test sets
      val_images, test_images, val_labels, test_labels = train_test_split(
          temp_images, temp_labels, test_size=test_size / (val_size +␣
↪test_size), random_state=random_state
      )

      train_dataset = list(zip(train_images, train_labels))
      val_dataset = list(zip(val_images, val_labels))
      test_dataset = list(zip(test_images, test_labels))

      return train_dataset, val_dataset, test_dataset

  def get_datasets(self):
      if self.split not in ['train', 'val', 'test']:
          raise ValueError("Invalid split. Use 'train', 'val', or 'test'.")

      dataset = self._get_image_paths()
      train_dataset, val_dataset, test_dataset = self._split_dataset(dataset)

      return train_dataset, val_dataset, test_dataset

  def __getitem__(self, index):
      if index < 0 or index >= len(self.image_paths):
          raise IndexError("Index out of range")

      image_path, label = self.image_paths[index]

      # Skip files with the '.DS_Store' extension
      while image_path.endswith('.DS_Store'):
          index += 1
          if index >= len(self.image_paths):
              raise IndexError("Index out of range")
          image_path, label = self.image_paths[index]

      # Open and read the image
```

```
        with Image.open(image_path) as image:
            # Check if the image has an alpha channel (4 channels)
            if image.mode == 'RGBA':
                image = image.convert('RGB')

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

        return image, label

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

    def custom_collate(self, batch):
        images, labels = zip(*batch)

    # Load images and apply transformations
        image_tensors = [self.transform(Image.open(img)) for img in images]

    # Convert labels to PyTorch tensor
        label_tensor = torch.tensor(labels, dtype=torch.long)

        return torch.stack(image_tensors), label_tensor
```

[5]:
```
# Function to print class distribution
def print_class_distribution(loader, name):
    labels = [label for _, label in loader.dataset]
    print(f"{name} set size: {len(loader.dataset)}")
    for class_label in range(5):  # Assuming you have 5 classes
        count = labels.count(class_label)
        print(f"Class {class_label}: {count} examples in the {name} set")

# Create separate instances for training, validation, and test datasets
root_directory = '/Users/peternoble/Downloads/Sugarcane_true'
sugarcane_instance = Sugarcane(root_directory, split='train')
train_dataset, val_dataset, test_dataset = sugarcane_instance.get_datasets()

# Create data loaders
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32,␣
 ↪shuffle=True, collate_fn=sugarcane_instance.custom_collate)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=32,␣
 ↪shuffle=False, collate_fn=sugarcane_instance.custom_collate)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32,␣
 ↪shuffle=False, collate_fn=sugarcane_instance.custom_collate)

# Print dataset sizes
print_class_distribution(train_loader, "Train")
```

```
# Print the size of the validation set
print_class_distribution(val_loader, "Validation")

# Print the size of the test set
print_class_distribution(test_loader, "Test")
```

```
Train set size: 1764
Class 0: 333 examples in the Train set
Class 1: 386 examples in the Train set
Class 2: 358 examples in the Train set
Class 3: 366 examples in the Train set
Class 4: 321 examples in the Train set
Validation set size: 504
Class 0: 118 examples in the Validation set
Class 1: 87 examples in the Validation set
Class 2: 105 examples in the Validation set
Class 3: 102 examples in the Validation set
Class 4: 92 examples in the Validation set
Test set size: 253
Class 0: 54 examples in the Test set
Class 1: 49 examples in the Test set
Class 2: 51 examples in the Test set
Class 3: 50 examples in the Test set
Class 4: 49 examples in the Test set
```

[6]:
```
# Set the path where you want to save the images. Useful for visualizing the␣
 ↪actual images used for validation
output_path = '/Users/peternoble/Desktop/results_true_3'
# Ensure the output directory exists
os.makedirs(output_path, exist_ok=True)
# Variables to store all images and labels
all_images = []
all_labels = []

# Iterate through the validation loader and concatenate images and labels
for batch_idx, (images, labels) in enumerate(val_loader):
# Apply histogram equalization to each image in the batch
#     equalized_images = torch.stack([torch.tensor(exposure.equalize_hist(img.
 ↪numpy())) for img in images])

#     all_images.append(equalized_images)
    all_images.append(images)
    all_labels.append(labels)

# Concatenate the lists to get all images and labels
all_images = torch.cat(all_images, dim=0)
```

```
all_labels = torch.cat(all_labels, dim=0)

# Define a function to save images
def save_images(images, labels, output_path):
    for i in range(len(images)):
        image, label = images[i], labels[i]
        # Assuming the images are in the range [0, 1], and using torchvision to␣
 ↪convert to PIL
        image_pil = transforms.ToPILImage()(image)
        image_pil.save(os.path.join(output_path, f"image_{i}_label_{label}.
 ↪png"))

# Save all equalized images from the validation dataset
save_images(all_images, all_labels, output_path)
```

[7]:
```
class CustomCNN(nn.Module):
    def __init__(self, num_classes=5):
        super(CustomCNN, self).__init__()

        # Layer 1
        self.conv1 = nn.Conv2d(3, 32, kernel_size=2, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 2
        self.conv2 = nn.Conv2d(32, 16, kernel_size=2, padding=1)
        self.bn2 = nn.BatchNorm2d(16)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 3
        self.conv3 = nn.Conv2d(16, 4, kernel_size=2, padding=1)
        self.bn3 = nn.BatchNorm2d(4)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 4
        self.conv4 = nn.Conv2d(4, 16, kernel_size=2, padding=1)
        self.bn4 = nn.BatchNorm2d(16)
        self.relu4 = nn.ReLU()
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 5
        self.conv5 = nn.Conv2d(16, 32, kernel_size=2, padding=1)
        self.bn5 = nn.BatchNorm2d(32)
        self.relu5 = nn.ReLU()
```

```
        self.pool5 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 6
        self.conv6 = nn.Conv2d(32, 64, kernel_size=2, padding=1)
        self.bn6 = nn.BatchNorm2d(64)
        self.relu6 = nn.ReLU()
        self.pool6 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 7
        self.conv7 = nn.Conv2d(64, 128, kernel_size=2, padding=1)
        self.bn7 = nn.BatchNorm2d(128)
        self.relu7 = nn.ReLU()
        self.pool7 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Layer 8
        self.conv8 = nn.Conv2d(128, 488, kernel_size=2, padding=1)
        self.bn8 = nn.BatchNorm2d(488)
        self.relu8 = nn.ReLU()
        self.pool8 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Adjusted fully connected layer
        self.fc = nn.Linear(488 * 2 * 2, num_classes)  # Adjust the size based␣
↪on your model architecture

    def forward(self, x):
        x = self.pool1(self.relu1(self.bn1(self.conv1(x))))
        x = self.pool2(self.relu2(self.bn2(self.conv2(x))))
        x = self.pool3(self.relu3(self.bn3(self.conv3(x))))
        x = self.pool4(self.relu4(self.bn4(self.conv4(x))))
        x = self.pool5(self.relu5(self.bn5(self.conv5(x))))
        x = self.pool6(self.relu6(self.bn6(self.conv6(x))))
        x = self.pool7(self.relu7(self.bn7(self.conv7(x))))
        x = self.pool8(self.relu8(self.bn8(self.conv8(x))))

        # Flatten the output before passing it through the fully connected layer
        x = x.view(x.size(0), -1)

        # Fully connected layer
        x = self.fc(x)

        # Apply softmax activation
        x = nn.functional.softmax(x, dim=1)

        return x

# Create an instance of the CustomCNN model
model = CustomCNN()
```

```python
# Print the model architecture
print(model)
```

```
CustomCNN(
  (conv1): Conv2d(3, 32, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu1): ReLU()
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv2): Conv2d(32, 16, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu2): ReLU()
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv3): Conv2d(16, 4, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (bn3): BatchNorm2d(4, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu3): ReLU()
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv4): Conv2d(4, 16, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (bn4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu4): ReLU()
  (pool4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv5): Conv2d(16, 32, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (bn5): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu5): ReLU()
  (pool5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv6): Conv2d(32, 64, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (bn6): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu6): ReLU()
  (pool6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv7): Conv2d(64, 128, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (bn7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu7): ReLU()
  (pool7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv8): Conv2d(128, 488, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
```

```
  (bn8): BatchNorm2d(488, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu8): ReLU()
  (pool8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (fc): Linear(in_features=1952, out_features=5, bias=True)
)
```

[8]:
```python
# Assuming sugarcane_loader is your DataLoader
for inputs, labels in val_loader:
    print("Input shape:", inputs.shape)
    break
```

```
Input shape: torch.Size([32, 3, 488, 488])
```

[9]:
```python
#Instantiate your model, optimizer, and criterion
#model = CustomCNN()  # Create an instance of your CustomModel
#optimizer = optim.Adam(model.parameters(), lr=0.001)
# Assuming you have a validation DataLoader named val_loader
# Specify the weight decay (regularization strength)
weight_decay = 0.01
#complex_model_95_7.pth'
# Define your optimizer and pass the weight_decay parameter L2 regular
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=weight_decay)
#optimizer = optim.Adam(model.parameters(), lr=0.001)
# Define RMSprop optimizer
#optimizer = optim.RMSprop(model.parameters(), lr=0.001, alpha=0.9)

criterion = nn.CrossEntropyLoss()

num_epochs = 150

class EarlyStopping:
    def __init__(self, patience=5, delta=0, verbose=False):
        self.patience = patience
        self.delta = delta
        self.verbose = verbose
        self.counter = 0
        self.best_score = None
        self.early_stop = False

    def __call__(self, val_loss, model):
        score = -val_loss

        if self.best_score is None:
            self.best_score = score
        elif score < self.best_score + self.delta:
            self.counter += 1
```

```
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.counter = 0

        if self.verbose:
            print(f'EarlyStopping counter: {self.counter} out of {self.
 ↪patience}')

        return self.early_stop

# Create an instance of EarlyStopping before the training loop
early_stopping = EarlyStopping(patience=20, verbose=True)

train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []

# Initialize variables for accuracy calculation
correct_train = 0
total_train = 0
correct_val = 0
total_val = 0

for epoch in range(num_epochs):
    model.train()
    running_train_loss = 0.0

    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_train_loss += loss.item()

        _, predicted_train = outputs.max(1)
        total_train += labels.size(0)
        correct_train += predicted_train.eq(labels).sum().item()

    average_train_loss = running_train_loss / len(train_loader)
    train_losses.append(average_train_loss)
    train_accuracy = correct_train / total_train
    train_accuracies.append(train_accuracy)
```

```
    model.eval()
    running_val_loss = 0.0

    # Reset variables for accuracy calculation
    correct_val = 0
    total_val = 0

    with torch.no_grad():
        for val_inputs, val_labels in val_loader:
            val_outputs = model(val_inputs)
            val_loss = criterion(val_outputs, val_labels)
            running_val_loss += val_loss.item()

            _, predicted_val = val_outputs.max(1)
            total_val += val_labels.size(0)
            correct_val += predicted_val.eq(val_labels).sum().item()

    average_val_loss = running_val_loss / len(val_loader)
    val_losses.append(average_val_loss)
    val_accuracy = correct_val / total_val
    val_accuracies.append(val_accuracy)

    # Call early_stopping within the loop
    if early_stopping(average_val_loss, model):
        print("Early stopping")
        break

    print(f"Epoch {epoch + 1}/{num_epochs}, Training Loss: {average_train_loss:.
↪4f}, Training Accuracy: {train_accuracy:.4f}, Validation Loss:␣
↪{average_val_loss:.4f}, Validation Accuracy: {val_accuracy:.4f}")

# Plotting the accuracy curve
epochs = range(1, len(train_accuracies) + 1)
plt.plot(epochs, train_accuracies, label='Training Accuracy')
plt.plot(epochs, val_accuracies, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy Over Epochs')
plt.legend()
plt.show()

# Plotting the loss curve
epochs = range(1, len(train_losses) + 1)
plt.plot(epochs, train_losses, label='Training Loss')
plt.plot(epochs, val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
```

```
plt.title('Training and Validation Loss Over Epochs')
plt.legend()
plt.show()
```

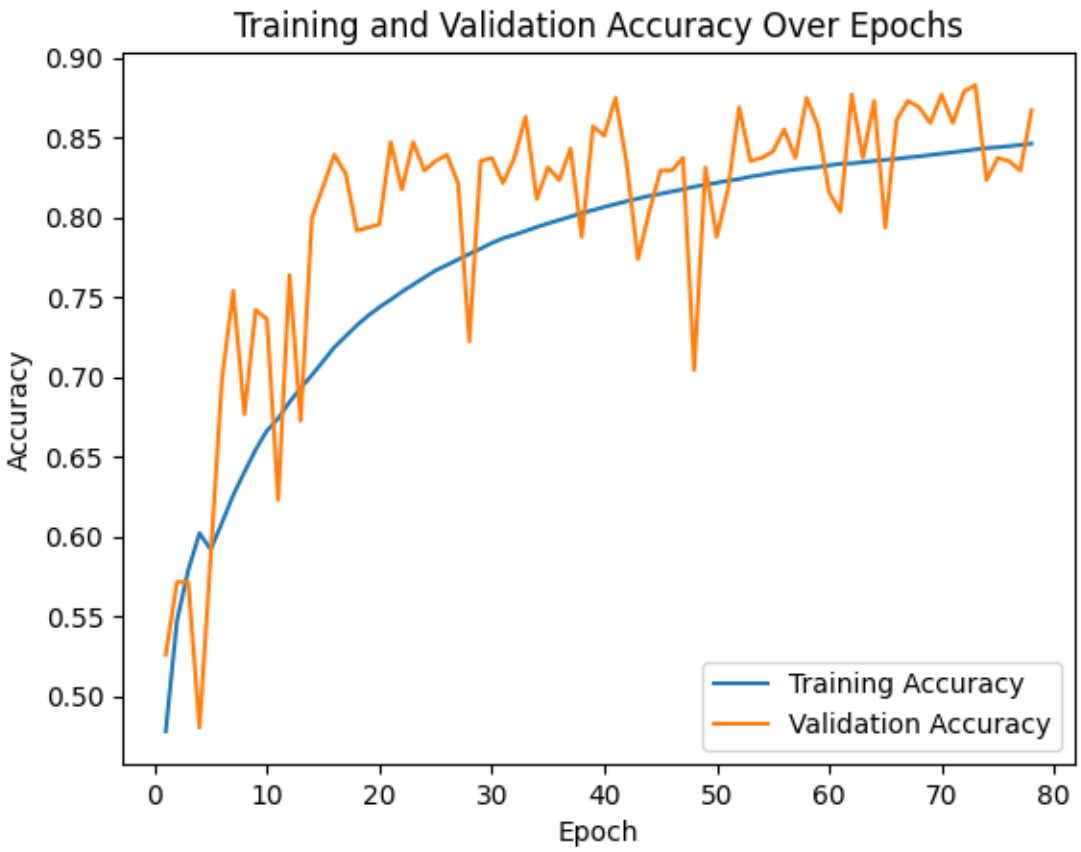
```
EarlyStopping counter: 0 out of 20
Epoch 1/150, Training Loss: 1.4091, Training Accuracy: 0.4779, Validation Loss:
1.3804, Validation Accuracy: 0.5258
EarlyStopping counter: 0 out of 20
Epoch 2/150, Training Loss: 1.2963, Training Accuracy: 0.5471, Validation Loss:
1.3235, Validation Accuracy: 0.5714
EarlyStopping counter: 1 out of 20
Epoch 3/150, Training Loss: 1.2665, Training Accuracy: 0.5788, Validation Loss:
1.3376, Validation Accuracy: 0.5714
EarlyStopping counter: 2 out of 20
Epoch 4/150, Training Loss: 1.2319, Training Accuracy: 0.6020, Validation Loss:
1.4177, Validation Accuracy: 0.4802
EarlyStopping counter: 0 out of 20
Epoch 5/150, Training Loss: 1.3512, Training Accuracy: 0.5917, Validation Loss:
1.3125, Validation Accuracy: 0.5873
EarlyStopping counter: 0 out of 20
Epoch 6/150, Training Loss: 1.2095, Training Accuracy: 0.6086, Validation Loss:
1.2028, Validation Accuracy: 0.6984
EarlyStopping counter: 0 out of 20
Epoch 7/150, Training Loss: 1.1777, Training Accuracy: 0.6259, Validation Loss:
1.1534, Validation Accuracy: 0.7540
EarlyStopping counter: 1 out of 20
Epoch 8/150, Training Loss: 1.1637, Training Accuracy: 0.6404, Validation Loss:
1.2153, Validation Accuracy: 0.6766
EarlyStopping counter: 2 out of 20
Epoch 9/150, Training Loss: 1.1476, Training Accuracy: 0.6544, Validation Loss:
1.1676, Validation Accuracy: 0.7421
EarlyStopping counter: 3 out of 20
Epoch 10/150, Training Loss: 1.1276, Training Accuracy: 0.6663, Validation Loss:
1.1711, Validation Accuracy: 0.7361
EarlyStopping counter: 4 out of 20
Epoch 11/150, Training Loss: 1.1502, Training Accuracy: 0.6739, Validation Loss:
1.2738, Validation Accuracy: 0.6230
EarlyStopping counter: 0 out of 20
Epoch 12/150, Training Loss: 1.1105, Training Accuracy: 0.6842, Validation Loss:
1.1339, Validation Accuracy: 0.7639
EarlyStopping counter: 1 out of 20
Epoch 13/150, Training Loss: 1.1137, Training Accuracy: 0.6930, Validation Loss:
1.2187, Validation Accuracy: 0.6726
EarlyStopping counter: 0 out of 20
Epoch 14/150, Training Loss: 1.1012, Training Accuracy: 0.7011, Validation Loss:
1.1085, Validation Accuracy: 0.7996
EarlyStopping counter: 0 out of 20
Epoch 15/150, Training Loss: 1.0870, Training Accuracy: 0.7099, Validation Loss:
```

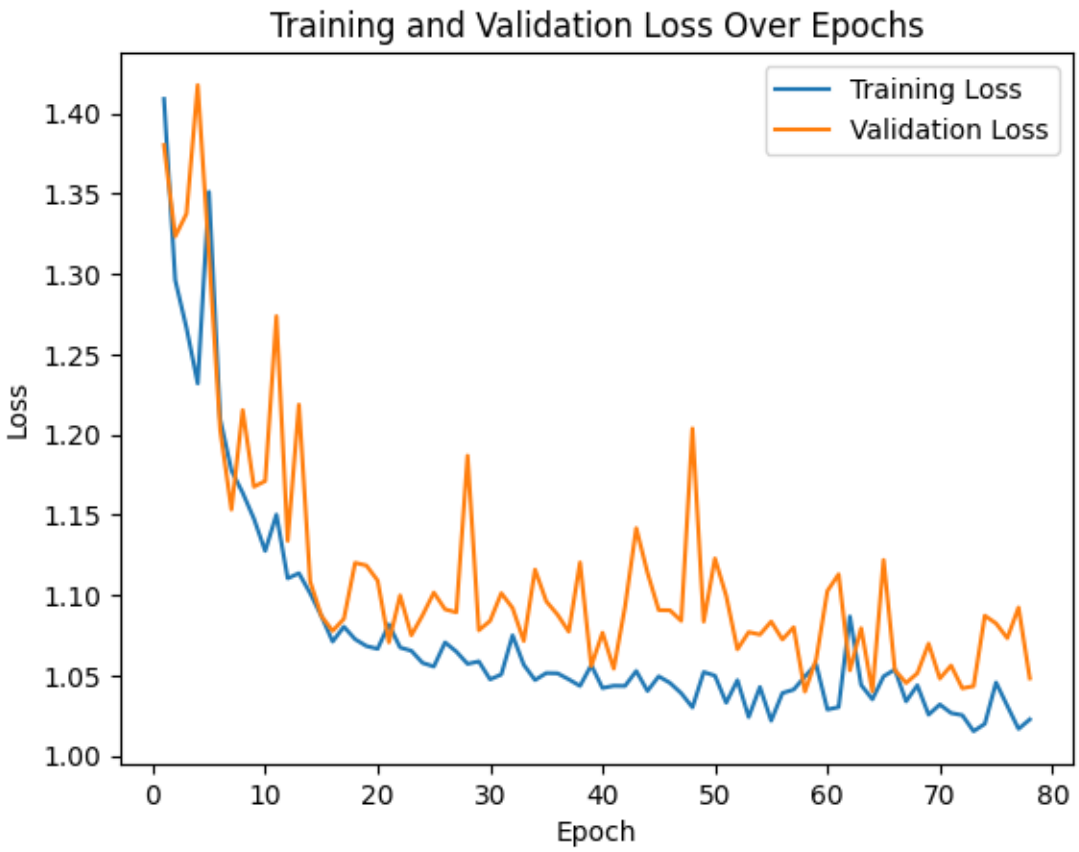
```
1.0870, Validation Accuracy: 0.8194
EarlyStopping counter: 0 out of 20
Epoch 16/150, Training Loss: 1.0711, Training Accuracy: 0.7187, Validation Loss:
1.0778, Validation Accuracy: 0.8393
EarlyStopping counter: 1 out of 20
Epoch 17/150, Training Loss: 1.0804, Training Accuracy: 0.7255, Validation Loss:
1.0853, Validation Accuracy: 0.8274
EarlyStopping counter: 2 out of 20
Epoch 18/150, Training Loss: 1.0726, Training Accuracy: 0.7324, Validation Loss:
1.1202, Validation Accuracy: 0.7917
EarlyStopping counter: 3 out of 20
Epoch 19/150, Training Loss: 1.0683, Training Accuracy: 0.7384, Validation Loss:
1.1185, Validation Accuracy: 0.7937
EarlyStopping counter: 4 out of 20
Epoch 20/150, Training Loss: 1.0666, Training Accuracy: 0.7437, Validation Loss:
1.1091, Validation Accuracy: 0.7956
EarlyStopping counter: 0 out of 20
Epoch 21/150, Training Loss: 1.0818, Training Accuracy: 0.7484, Validation Loss:
1.0706, Validation Accuracy: 0.8472
EarlyStopping counter: 1 out of 20
Epoch 22/150, Training Loss: 1.0674, Training Accuracy: 0.7533, Validation Loss:
1.0999, Validation Accuracy: 0.8175
EarlyStopping counter: 2 out of 20
Epoch 23/150, Training Loss: 1.0654, Training Accuracy: 0.7579, Validation Loss:
1.0750, Validation Accuracy: 0.8472
EarlyStopping counter: 3 out of 20
Epoch 24/150, Training Loss: 1.0578, Training Accuracy: 0.7624, Validation Loss:
1.0876, Validation Accuracy: 0.8294
EarlyStopping counter: 4 out of 20
Epoch 25/150, Training Loss: 1.0556, Training Accuracy: 0.7668, Validation Loss:
1.1016, Validation Accuracy: 0.8353
EarlyStopping counter: 5 out of 20
Epoch 26/150, Training Loss: 1.0706, Training Accuracy: 0.7701, Validation Loss:
1.0912, Validation Accuracy: 0.8393
EarlyStopping counter: 6 out of 20
Epoch 27/150, Training Loss: 1.0649, Training Accuracy: 0.7736, Validation Loss:
1.0893, Validation Accuracy: 0.8214
EarlyStopping counter: 7 out of 20
Epoch 28/150, Training Loss: 1.0572, Training Accuracy: 0.7771, Validation Loss:
1.1869, Validation Accuracy: 0.7222
EarlyStopping counter: 8 out of 20
Epoch 29/150, Training Loss: 1.0587, Training Accuracy: 0.7804, Validation Loss:
1.0782, Validation Accuracy: 0.8353
EarlyStopping counter: 9 out of 20
Epoch 30/150, Training Loss: 1.0476, Training Accuracy: 0.7839, Validation Loss:
1.0840, Validation Accuracy: 0.8373
EarlyStopping counter: 10 out of 20
Epoch 31/150, Training Loss: 1.0506, Training Accuracy: 0.7869, Validation Loss:
```

```
1.1012, Validation Accuracy: 0.8214
EarlyStopping counter: 11 out of 20
Epoch 32/150, Training Loss: 1.0750, Training Accuracy: 0.7891, Validation Loss:
1.0920, Validation Accuracy: 0.8373
EarlyStopping counter: 12 out of 20
Epoch 33/150, Training Loss: 1.0566, Training Accuracy: 0.7915, Validation Loss:
1.0715, Validation Accuracy: 0.8631
EarlyStopping counter: 13 out of 20
Epoch 34/150, Training Loss: 1.0472, Training Accuracy: 0.7940, Validation Loss:
1.1160, Validation Accuracy: 0.8115
EarlyStopping counter: 14 out of 20
Epoch 35/150, Training Loss: 1.0515, Training Accuracy: 0.7962, Validation Loss:
1.0962, Validation Accuracy: 0.8313
EarlyStopping counter: 15 out of 20
Epoch 36/150, Training Loss: 1.0513, Training Accuracy: 0.7983, Validation Loss:
1.0879, Validation Accuracy: 0.8234
EarlyStopping counter: 16 out of 20
Epoch 37/150, Training Loss: 1.0476, Training Accuracy: 0.8004, Validation Loss:
1.0773, Validation Accuracy: 0.8433
EarlyStopping counter: 17 out of 20
Epoch 38/150, Training Loss: 1.0435, Training Accuracy: 0.8028, Validation Loss:
1.1204, Validation Accuracy: 0.7877
EarlyStopping counter: 0 out of 20
Epoch 39/150, Training Loss: 1.0564, Training Accuracy: 0.8045, Validation Loss:
1.0558, Validation Accuracy: 0.8571
EarlyStopping counter: 1 out of 20
Epoch 40/150, Training Loss: 1.0422, Training Accuracy: 0.8065, Validation Loss:
1.0767, Validation Accuracy: 0.8512
EarlyStopping counter: 0 out of 20
Epoch 41/150, Training Loss: 1.0436, Training Accuracy: 0.8083, Validation Loss:
1.0542, Validation Accuracy: 0.8750
EarlyStopping counter: 1 out of 20
Epoch 42/150, Training Loss: 1.0435, Training Accuracy: 0.8101, Validation Loss:
1.0927, Validation Accuracy: 0.8333
EarlyStopping counter: 2 out of 20
Epoch 43/150, Training Loss: 1.0528, Training Accuracy: 0.8118, Validation Loss:
1.1418, Validation Accuracy: 0.7738
EarlyStopping counter: 3 out of 20
Epoch 44/150, Training Loss: 1.0404, Training Accuracy: 0.8135, Validation Loss:
1.1141, Validation Accuracy: 0.8036
EarlyStopping counter: 4 out of 20
Epoch 45/150, Training Loss: 1.0494, Training Accuracy: 0.8148, Validation Loss:
1.0907, Validation Accuracy: 0.8294
EarlyStopping counter: 5 out of 20
Epoch 46/150, Training Loss: 1.0454, Training Accuracy: 0.8162, Validation Loss:
1.0906, Validation Accuracy: 0.8294
EarlyStopping counter: 6 out of 20
Epoch 47/150, Training Loss: 1.0389, Training Accuracy: 0.8176, Validation Loss:
```

```
1.0840, Validation Accuracy: 0.8373
EarlyStopping counter: 7 out of 20
Epoch 48/150, Training Loss: 1.0301, Training Accuracy: 0.8193, Validation Loss:
1.2038, Validation Accuracy: 0.7044
EarlyStopping counter: 8 out of 20
Epoch 49/150, Training Loss: 1.0521, Training Accuracy: 0.8205, Validation Loss:
1.0836, Validation Accuracy: 0.8313
EarlyStopping counter: 9 out of 20
Epoch 50/150, Training Loss: 1.0499, Training Accuracy: 0.8217, Validation Loss:
1.1229, Validation Accuracy: 0.7877
EarlyStopping counter: 10 out of 20
Epoch 51/150, Training Loss: 1.0330, Training Accuracy: 0.8230, Validation Loss:
1.0996, Validation Accuracy: 0.8175
EarlyStopping counter: 11 out of 20
Epoch 52/150, Training Loss: 1.0471, Training Accuracy: 0.8240, Validation Loss:
1.0663, Validation Accuracy: 0.8690
EarlyStopping counter: 12 out of 20
Epoch 53/150, Training Loss: 1.0241, Training Accuracy: 0.8256, Validation Loss:
1.0770, Validation Accuracy: 0.8353
EarlyStopping counter: 13 out of 20
Epoch 54/150, Training Loss: 1.0428, Training Accuracy: 0.8266, Validation Loss:
1.0755, Validation Accuracy: 0.8373
EarlyStopping counter: 14 out of 20
Epoch 55/150, Training Loss: 1.0218, Training Accuracy: 0.8280, Validation Loss:
1.0836, Validation Accuracy: 0.8413
EarlyStopping counter: 15 out of 20
Epoch 56/150, Training Loss: 1.0391, Training Accuracy: 0.8291, Validation Loss:
1.0724, Validation Accuracy: 0.8552
EarlyStopping counter: 16 out of 20
Epoch 57/150, Training Loss: 1.0412, Training Accuracy: 0.8300, Validation Loss:
1.0800, Validation Accuracy: 0.8373
EarlyStopping counter: 0 out of 20
Epoch 58/150, Training Loss: 1.0493, Training Accuracy: 0.8309, Validation Loss:
1.0399, Validation Accuracy: 0.8750
EarlyStopping counter: 1 out of 20
Epoch 59/150, Training Loss: 1.0570, Training Accuracy: 0.8315, Validation Loss:
1.0600, Validation Accuracy: 0.8571
EarlyStopping counter: 2 out of 20
Epoch 60/150, Training Loss: 1.0288, Training Accuracy: 0.8326, Validation Loss:
1.1026, Validation Accuracy: 0.8155
EarlyStopping counter: 3 out of 20
Epoch 61/150, Training Loss: 1.0303, Training Accuracy: 0.8336, Validation Loss:
1.1130, Validation Accuracy: 0.8036
EarlyStopping counter: 4 out of 20
Epoch 62/150, Training Loss: 1.0867, Training Accuracy: 0.8338, Validation Loss:
1.0532, Validation Accuracy: 0.8770
EarlyStopping counter: 5 out of 20
Epoch 63/150, Training Loss: 1.0441, Training Accuracy: 0.8346, Validation Loss:
```

```
1.0794, Validation Accuracy: 0.8373
EarlyStopping counter: 6 out of 20
Epoch 64/150, Training Loss: 1.0351, Training Accuracy: 0.8354, Validation Loss:
1.0402, Validation Accuracy: 0.8730
EarlyStopping counter: 7 out of 20
Epoch 65/150, Training Loss: 1.0495, Training Accuracy: 0.8361, Validation Loss:
1.1220, Validation Accuracy: 0.7937
EarlyStopping counter: 8 out of 20
Epoch 66/150, Training Loss: 1.0537, Training Accuracy: 0.8367, Validation Loss:
1.0534, Validation Accuracy: 0.8611
EarlyStopping counter: 9 out of 20
Epoch 67/150, Training Loss: 1.0339, Training Accuracy: 0.8376, Validation Loss:
1.0453, Validation Accuracy: 0.8730
EarlyStopping counter: 10 out of 20
Epoch 68/150, Training Loss: 1.0439, Training Accuracy: 0.8383, Validation Loss:
1.0514, Validation Accuracy: 0.8690
EarlyStopping counter: 11 out of 20
Epoch 69/150, Training Loss: 1.0256, Training Accuracy: 0.8391, Validation Loss:
1.0698, Validation Accuracy: 0.8591
EarlyStopping counter: 12 out of 20
Epoch 70/150, Training Loss: 1.0319, Training Accuracy: 0.8399, Validation Loss:
1.0481, Validation Accuracy: 0.8770
EarlyStopping counter: 13 out of 20
Epoch 71/150, Training Loss: 1.0266, Training Accuracy: 0.8408, Validation Loss:
1.0562, Validation Accuracy: 0.8591
EarlyStopping counter: 14 out of 20
Epoch 72/150, Training Loss: 1.0253, Training Accuracy: 0.8416, Validation Loss:
1.0420, Validation Accuracy: 0.8790
EarlyStopping counter: 15 out of 20
Epoch 73/150, Training Loss: 1.0153, Training Accuracy: 0.8426, Validation Loss:
1.0433, Validation Accuracy: 0.8829
EarlyStopping counter: 16 out of 20
Epoch 74/150, Training Loss: 1.0199, Training Accuracy: 0.8434, Validation Loss:
1.0872, Validation Accuracy: 0.8234
EarlyStopping counter: 17 out of 20
Epoch 75/150, Training Loss: 1.0456, Training Accuracy: 0.8440, Validation Loss:
1.0824, Validation Accuracy: 0.8373
EarlyStopping counter: 18 out of 20
Epoch 76/150, Training Loss: 1.0311, Training Accuracy: 0.8446, Validation Loss:
1.0733, Validation Accuracy: 0.8353
EarlyStopping counter: 19 out of 20
Epoch 77/150, Training Loss: 1.0167, Training Accuracy: 0.8454, Validation Loss:
1.0924, Validation Accuracy: 0.8294
EarlyStopping counter: 20 out of 20
Early stopping
```


Training and Validation Accuracy Over Epochs
Accuracy
Epoch
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0
10
20
30
40
50
60
70
80
Training Accuracy
Validation Accuracy



[10]:
```
# Save the model state_dict
torch.save(model.state_dict(), 'complex_model95_final.pth')
```

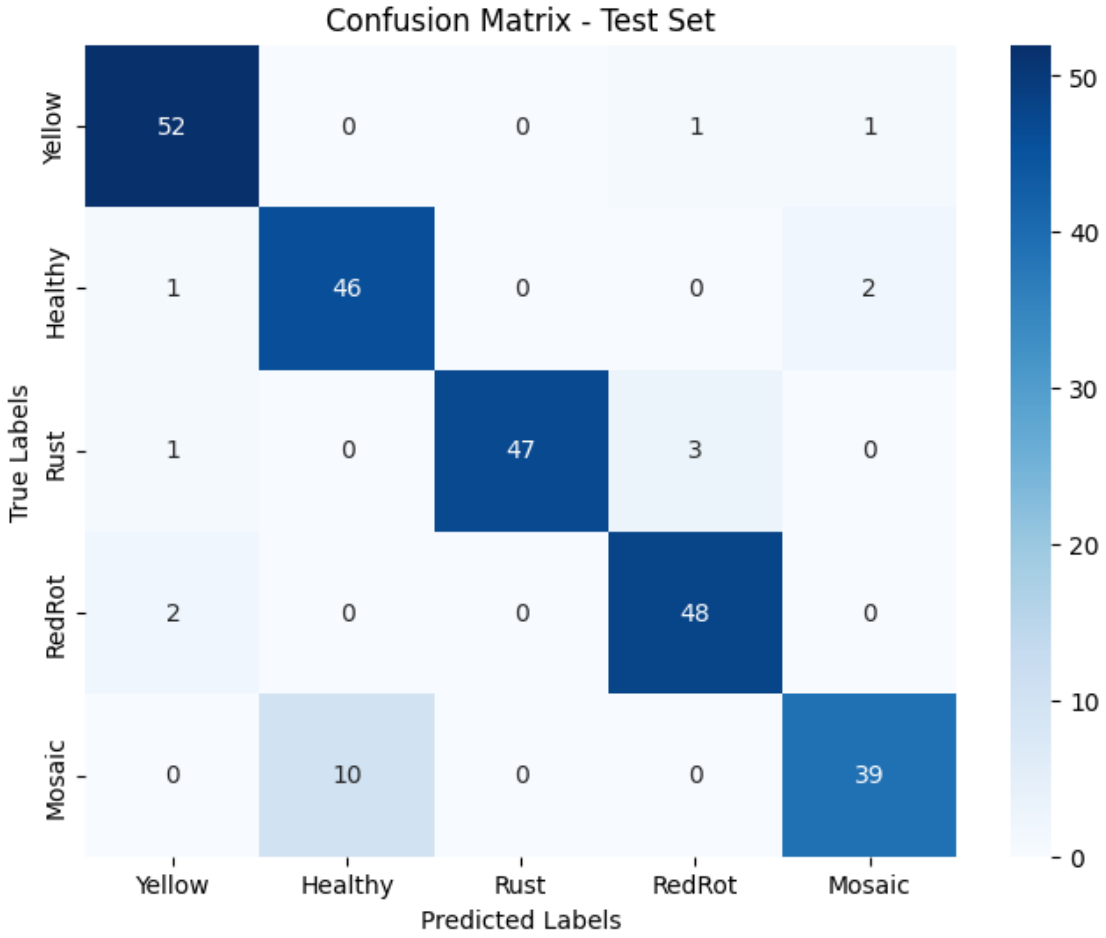
[ ]:
```

```

[11]:
```
def evaluate_model(loader, custom_labels, dataset_name):
    model.eval()  # Set the model to evaluation mode
    all_predictions = []
    all_labels = []

    with torch.no_grad():
        for inputs, labels in loader:
            try:
                inputs = inputs.view(inputs.size(0), 3, 488, 488)  # Ensure the␣
 ↪input size is correct
                outputs = model(inputs)
                predictions = torch.argmax(outputs, dim=1)
                all_predictions.extend(predictions.tolist())
                all_labels.extend(labels.tolist())  # Populate the true labels
            except Exception as e:
```

```
                print(f"An error occurred: {e}")

    # Compute accuracy for the dataset
    if len(all_labels) > 0:
        accuracy = sum(p == l for p, l in zip(all_predictions, all_labels)) /␣
 ↪len(all_labels)
        print(f"{dataset_name} Accuracy: {accuracy:.1%}")
    else:
        print(f"No labels available for computing {dataset_name} accuracy.")

    # Analyze classification breakdown for the dataset
    for class_label, custom_label in enumerate(custom_labels):
        correct = sum(p == class_label and l == class_label for p, l in␣
 ↪zip(all_predictions, all_labels))
        total_in_set = all_labels.count(class_label)

        if total_in_set == 0:
            print(f"{custom_label}: No examples in the {dataset_name} set")
        else:
            percentage = correct / total_in_set if total_in_set != 0 else 0.0
            print(f"{custom_label}: Correctly classified {correct}/
 ↪{total_in_set} ({percentage:.1%})")

    # Return the lists of all_labels and all_predictions
    return all_labels, all_predictions


# Define custom labels
custom_labels = ['Yellow', 'Healthy', 'Rust', 'RedRot', 'Mosaic']

# Evaluate on the test set
all_labels_test, all_predictions_test = evaluate_model(test_loader,␣
 ↪custom_labels, "\nTest")

# Compute confusion matrix for the test set
conf_matrix = confusion_matrix(all_labels_test, all_predictions_test)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',␣
 ↪xticklabels=custom_labels, yticklabels=custom_labels)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix - Test Set')
plt.show()

# Generate and print classification report
```

```
class_report = classification_report(all_labels_test, all_predictions_test,␣
 ↪target_names=custom_labels)
print("Classification Report - Test Set:\n", class_report)
```

```
Test Accuracy: 91.7%
Yellow: Correctly classified 52/54 (96.3%)
Healthy: Correctly classified 46/49 (93.9%)
Rust: Correctly classified 47/51 (92.2%)
RedRot: Correctly classified 48/50 (96.0%)
Mosaic: Correctly classified 39/49 (79.6%)
```



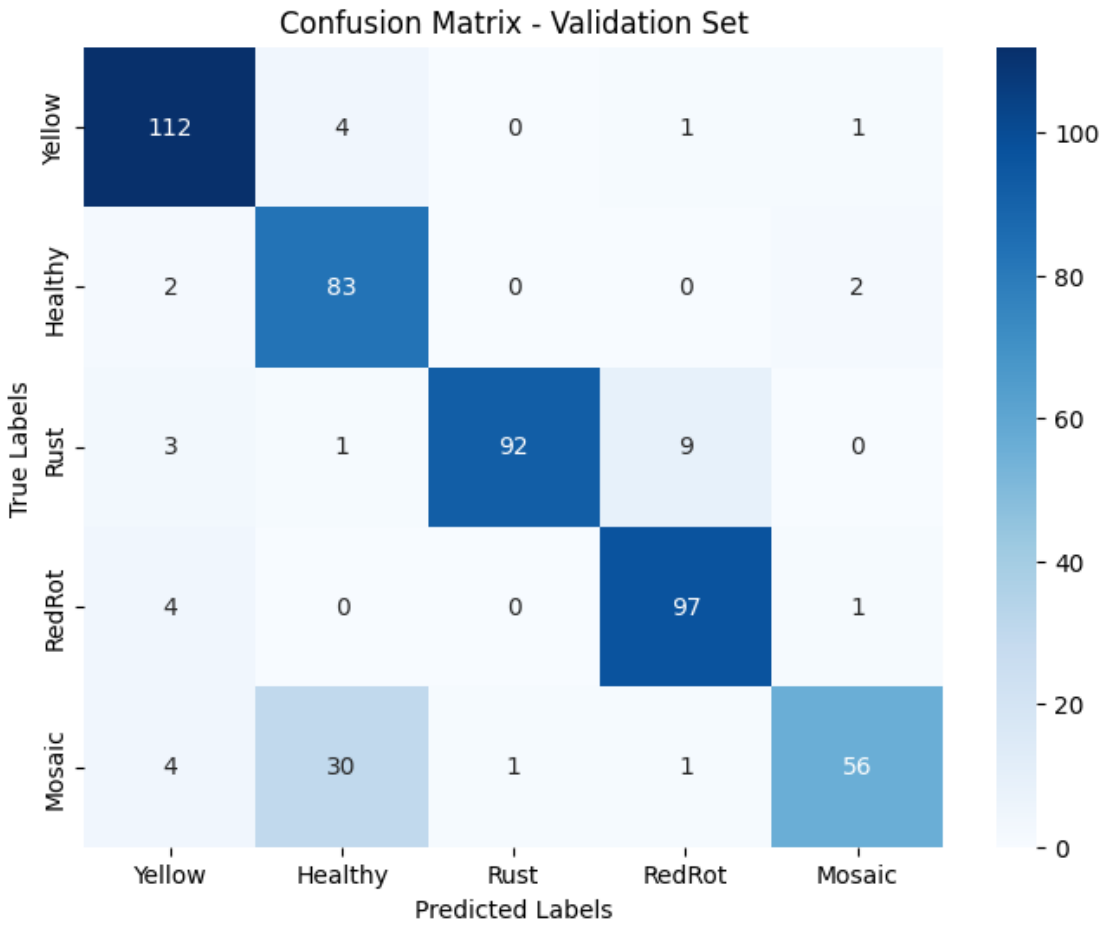

```
Classification Report - Test Set:
              precision    recall  f1-score   support

      Yellow       0.93      0.96      0.95        54
     Healthy       0.82      0.94      0.88        49
        Rust       1.00      0.92      0.96        51
```

```
      RedRot       0.92      0.96      0.94        50
      Mosaic       0.93      0.80      0.86        49

    accuracy                           0.92       253
   macro avg       0.92      0.92      0.92       253
weighted avg       0.92      0.92      0.92       253
```

[12]:
```python
# Generate and print classification report
# Evaluate on the validation set
def evaluate_model2(loader, custom_labels, dataset_name):
    model.eval()  # Set the model to evaluation mode
    all_predictions = []
    all_labels = []

    with torch.no_grad():
        for inputs, labels in loader:
            try:
                inputs = inputs.view(inputs.size(0), 3, 488, 488)  # Ensure the␣
 ↪input size is correct
                outputs = model(inputs)
                predictions = torch.argmax(outputs, dim=1)
                all_predictions.extend(predictions.tolist())
                all_labels.extend(labels.tolist())  # Populate the true labels
            except Exception as e:
                print(f"An error occurred: {e}")

    # Compute accuracy for the dataset
    if len(all_labels) > 0:
        accuracy = sum(p == l for p, l in zip(all_predictions, all_labels)) /␣
 ↪len(all_labels)
        print(f"{dataset_name} Accuracy: {accuracy:.1%}")
    else:
        print(f"No labels available for computing {dataset_name} accuracy.")

    # Analyze classification breakdown for the dataset
    for class_label, custom_label in enumerate(custom_labels):
        correct = sum(p == class_label and l == class_label for p, l in␣
 ↪zip(all_predictions, all_labels))
        total_in_set = all_labels.count(class_label)

        if total_in_set == 0:
            print(f"{custom_label}: No examples in the {dataset_name} set")
        else:
            percentage = correct / total_in_set if total_in_set != 0 else 0.0
            print(f"{custom_label}: Correctly classified {correct}/
 ↪{total_in_set} ({percentage:.1%})")
```

```
    # Return the lists of all_labels and all_predictions
    return all_labels, all_predictions

all_labels_val, all_predictions_val = evaluate_model2(val_loader,␣
 ↪custom_labels, "Validation")

# Compute confusion matrix for the test set
conf_matrix = confusion_matrix(all_labels_val, all_predictions_val)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',␣
 ↪xticklabels=custom_labels, yticklabels=custom_labels)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix - Validation Set')
plt.show()

class_report = classification_report(all_labels_val, all_predictions_val,␣
 ↪target_names=custom_labels)
print("Classification Report - Validation Set:\n", class_report)
```

```
Validation Accuracy: 87.3%
Yellow: Correctly classified 112/118 (94.9%)
Healthy: Correctly classified 83/87 (95.4%)
Rust: Correctly classified 92/105 (87.6%)
RedRot: Correctly classified 97/102 (95.1%)
Mosaic: Correctly classified 56/92 (60.9%)
```



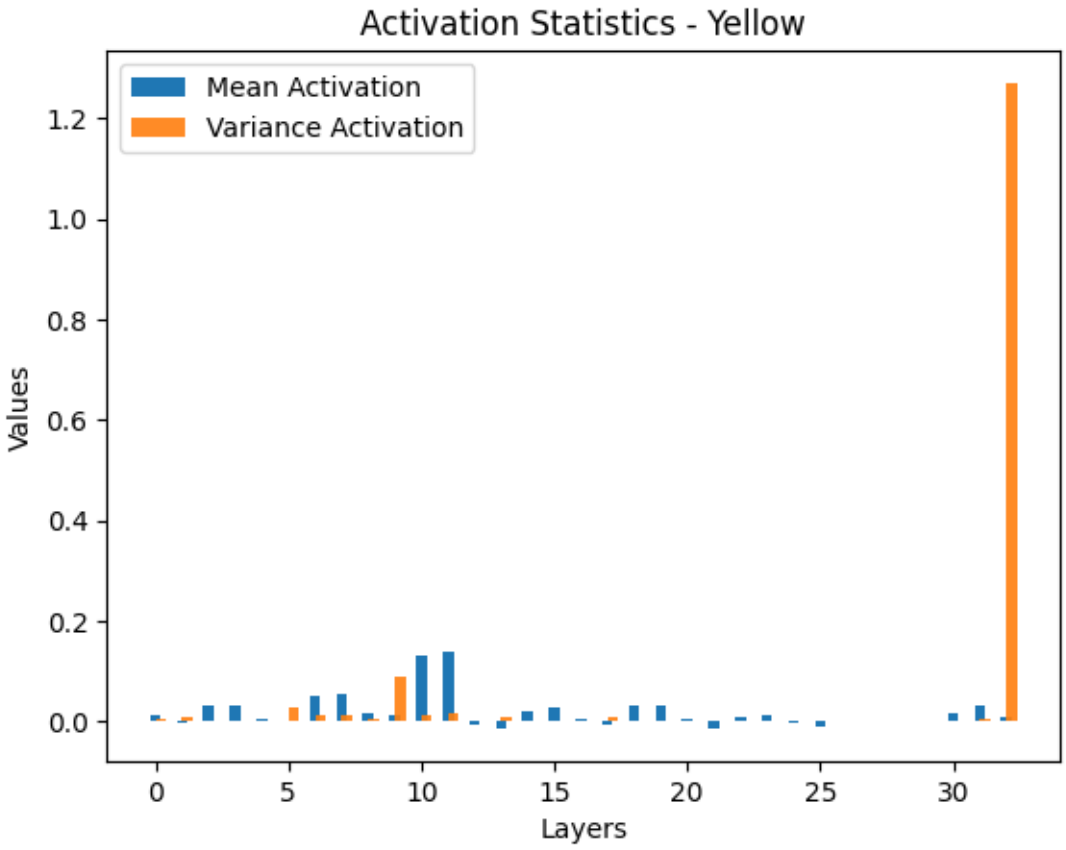

```
Classification Report - Validation Set:
              precision    recall  f1-score   support

      Yellow       0.90      0.95      0.92       118
     Healthy       0.70      0.95      0.81        87
        Rust       0.99      0.88      0.93       105
     RedRot       0.90      0.95      0.92       102
      Mosaic       0.93      0.61      0.74        92

    accuracy                           0.87       504
   macro avg       0.88      0.87      0.86       504
weighted avg       0.89      0.87      0.87       504
```

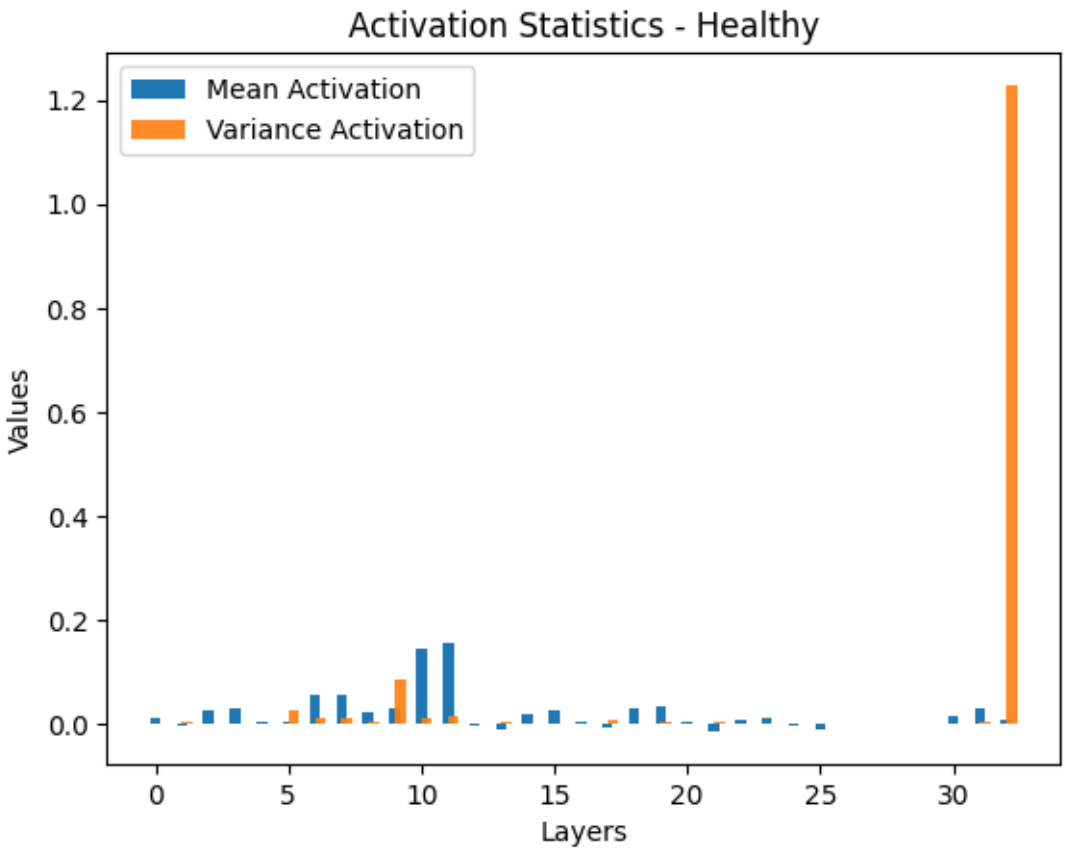
[13]:
```
from matplotlib.backends.backend_pdf import PdfPages
import matplotlib.pyplot as plt
import random
from torchvision import transforms
```

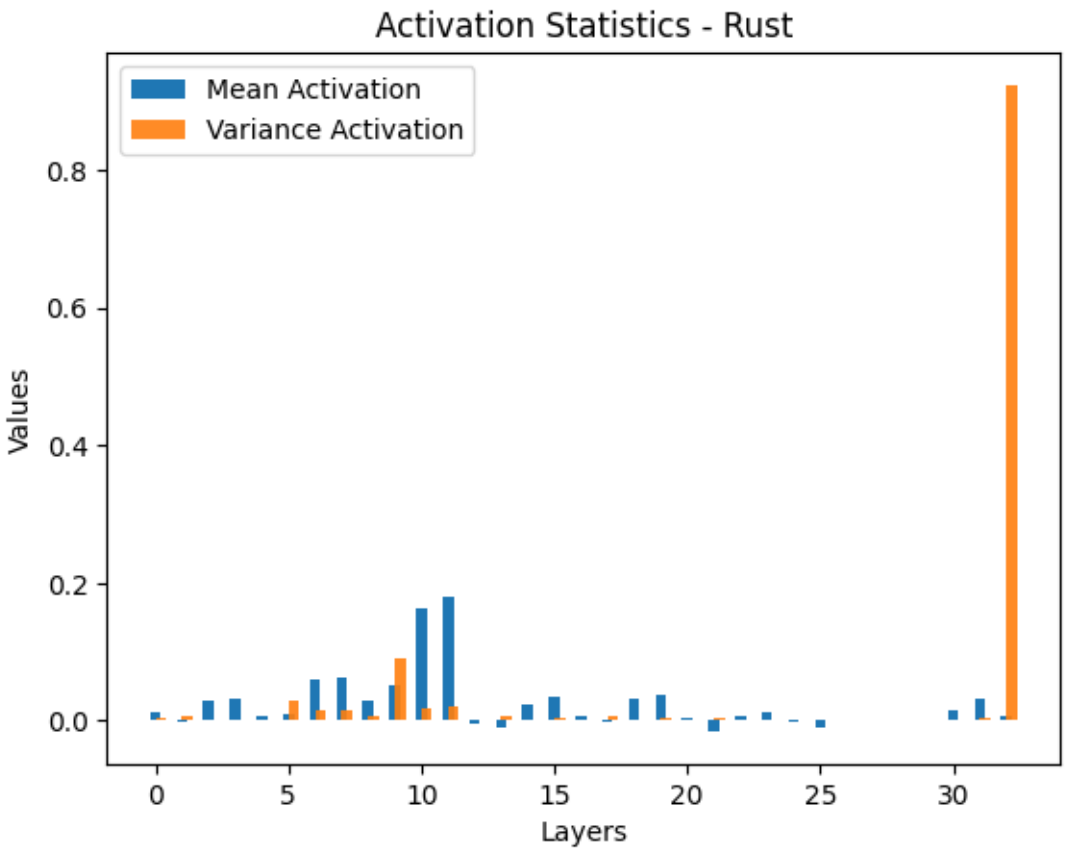
```
from PIL import Image

# Assuming you have already defined the CustomCNN class and loaded the model
# Also, define the 'model' and 'transform' variables before this code snippet

# Create a PdfPages object to store the plots
pdf_pages = PdfPages('activation_plots.pdf')

# List of directories to investigate
directory_paths = [
    '/Users/peternoble/Downloads/Sugarcane_true/training/0',
    '/Users/peternoble/Downloads/Sugarcane_true/training/1',
    '/Users/peternoble/Downloads/Sugarcane_true/training/2',
    '/Users/peternoble/Downloads/Sugarcane_true/training/3',
    '/Users/peternoble/Downloads/Sugarcane_true/training/4',
]

# Custom labels for each directory
custom_labels = ['Yellow', 'Healthy', 'Rust', 'RedRot', 'Mosaic']

# Loop through each directory
for i, directory_path in enumerate(directory_paths):
    # Choose a random image file from the list
    random_image_file = random.choice(os.listdir(directory_path))

    # Construct the full path to the randomly chosen image file
    random_image_path = os.path.join(directory_path, random_image_file)

    # Open the image using PIL
    random_image = Image.open(random_image_path)

    # Define the transformation for the image
    transform = transforms.Compose([
        transforms.Resize((488, 488)),
        transforms.ToTensor(),
    ])

    # Apply the transformation to the image
    input_image = transform(random_image).unsqueeze(0)  # Add batch dimension

    # Create a new figure for each plot
    fig, ax = plt.subplots()

    # Create a list to store the activation statistics
    activation_stats = []

    # Define a hook to store the activation statistics at each layer
```

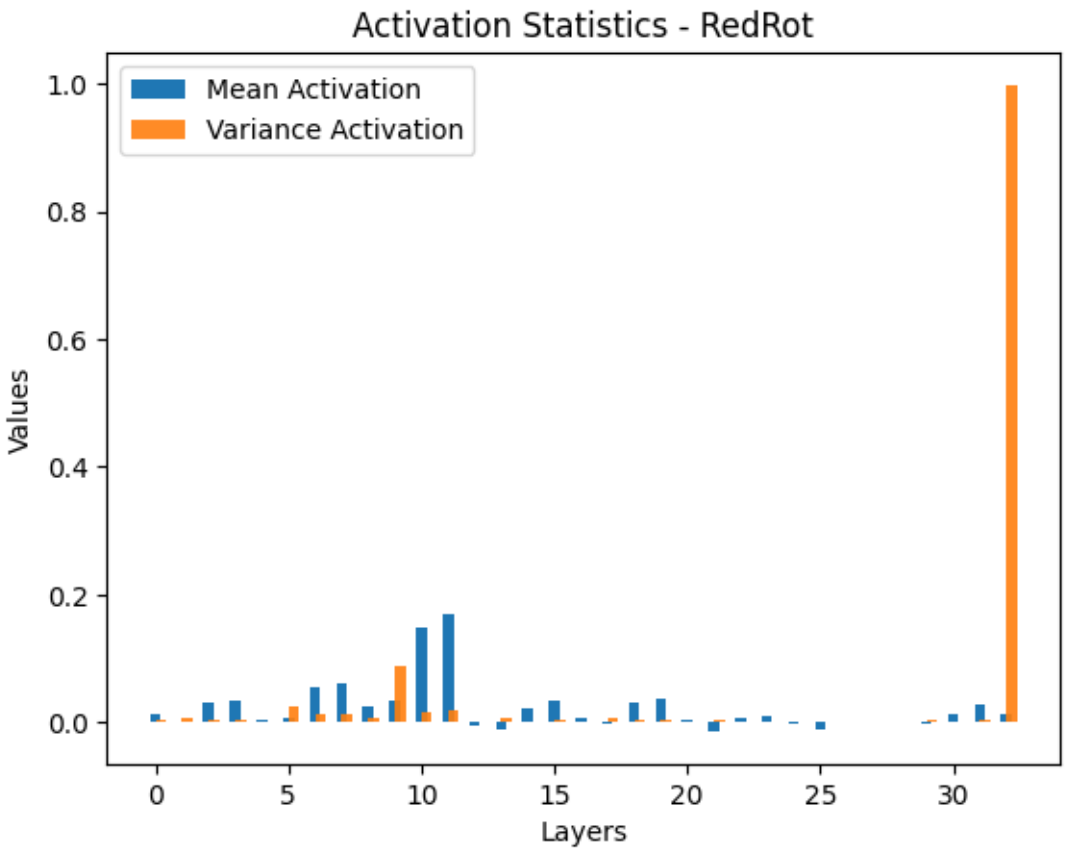
```
    def hook_fn(module, input, output):
        mean_activation = output.mean().item()
        var_activation = output.var().item()
        activation_stats.append((mean_activation, var_activation))

    # Register the hook to each layer in your model
    hooks = []
    for layer in model.children():
        hook = layer.register_forward_hook(hook_fn)
        hooks.append(hook)

    # Forward pass to obtain activation statistics
    with torch.no_grad():
        model(input_image)

    # Remove the hooks
    for hook in hooks:
        hook.remove()

    # Visualize activation statistics
    layer_names = [f'Layer {i+1}' for i in range(len(activation_stats))]
    mean_activations, var_activations = zip(*activation_stats)

    # Create separate x-values for mean and variance
    x_mean = [i for i in range(len(mean_activations))]
    x_var = [i + 0.2 for i in range(len(var_activations))]  # Add a small␣
↪offset for variance

    # Plot the activation statistics
    ax.bar(x_mean, mean_activations, width=0.4, label='Mean Activation')
    ax.bar(x_var, var_activations, width=0.4, label='Variance Activation',␣
↪alpha=0.9)
    ax.set_xlabel('Layers')
    ax.set_ylabel('Values')
    ax.set_title(f'Activation Statistics - {custom_labels[i]}')
    ax.legend()

    # Save the current plot to the PDF
    pdf_pages.savefig(fig)

    # Show the current plot on the screen
    plt.show()

    # Close the current plot
    plt.close()

# Close the PdfPages object
```

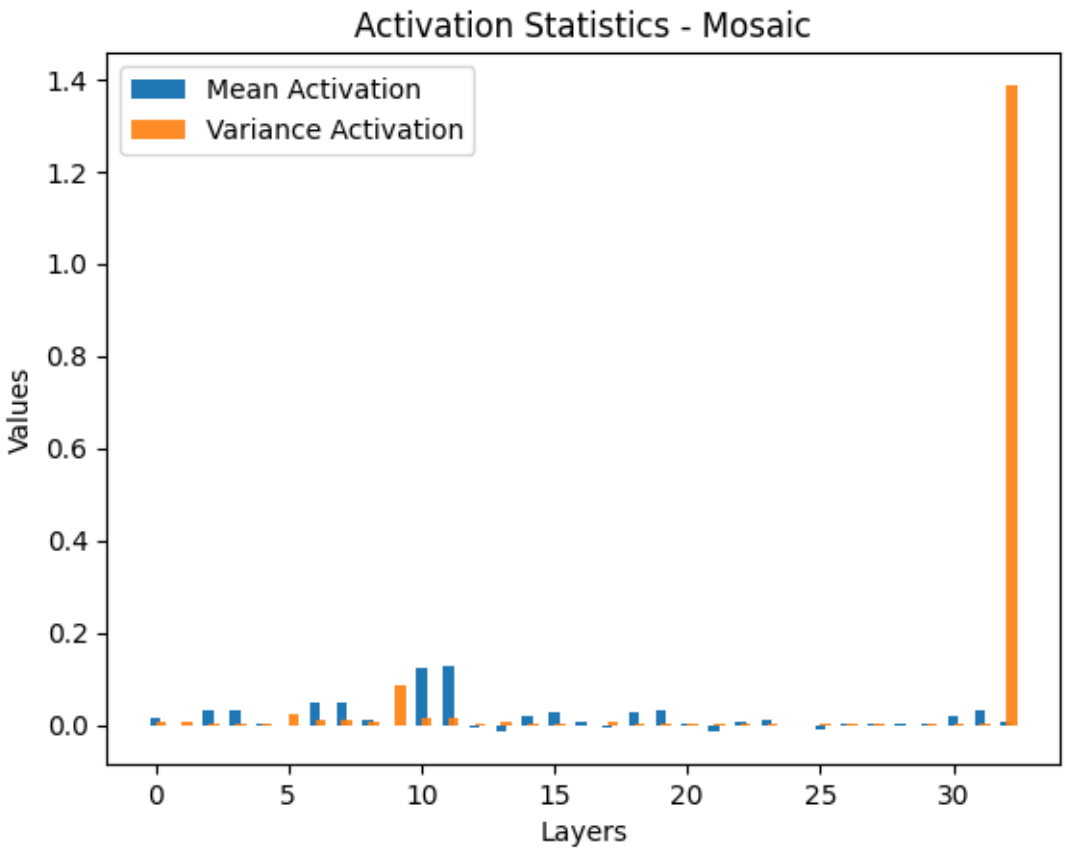
```
pdf_pages.close()
```




Activation Statistics - Healthy
Mean Activation
Variance Activation
Values
Layers
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
5
10
15
20
25
30


Activation Statistics - Rust
Mean Activation
Variance Activation
Values
0.0
0.2
0.4
0.6
0.8
0
5
10
15
20
25
30
Layers


Activation Statistics - RedRot
Mean Activation
Variance Activation
Values
Layers
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20
25
30



[14]:
```python
# Apply the transformation to the image
input_image = transform(random_image).unsqueeze(0)  # Add batch dimension

# Create a list to store the feature maps
feature_maps = []

# Define a hook to store the feature maps at each layer
def hook_fn(module, input, output):
    feature_maps.append(output)

# Register the hook to each layer in your model
hooks = []
for layer in model.children():
    hook = layer.register_forward_hook(hook_fn)
    hooks.append(hook)

# Forward pass to obtain feature maps
with torch.no_grad():
    model(input_image)
```

```
# Print the shape of the feature maps
for i, feature_map in enumerate(feature_maps):
    print(f'Layer {i + 1} - Shape: {feature_map.shape}')

# Visualize the feature maps from the 2nd layer
layer_to_visualize = 1  # 0-based index, corresponds to the 2nd layer
feature_map_normalized = feature_maps[layer_to_visualize].squeeze().numpy()
```

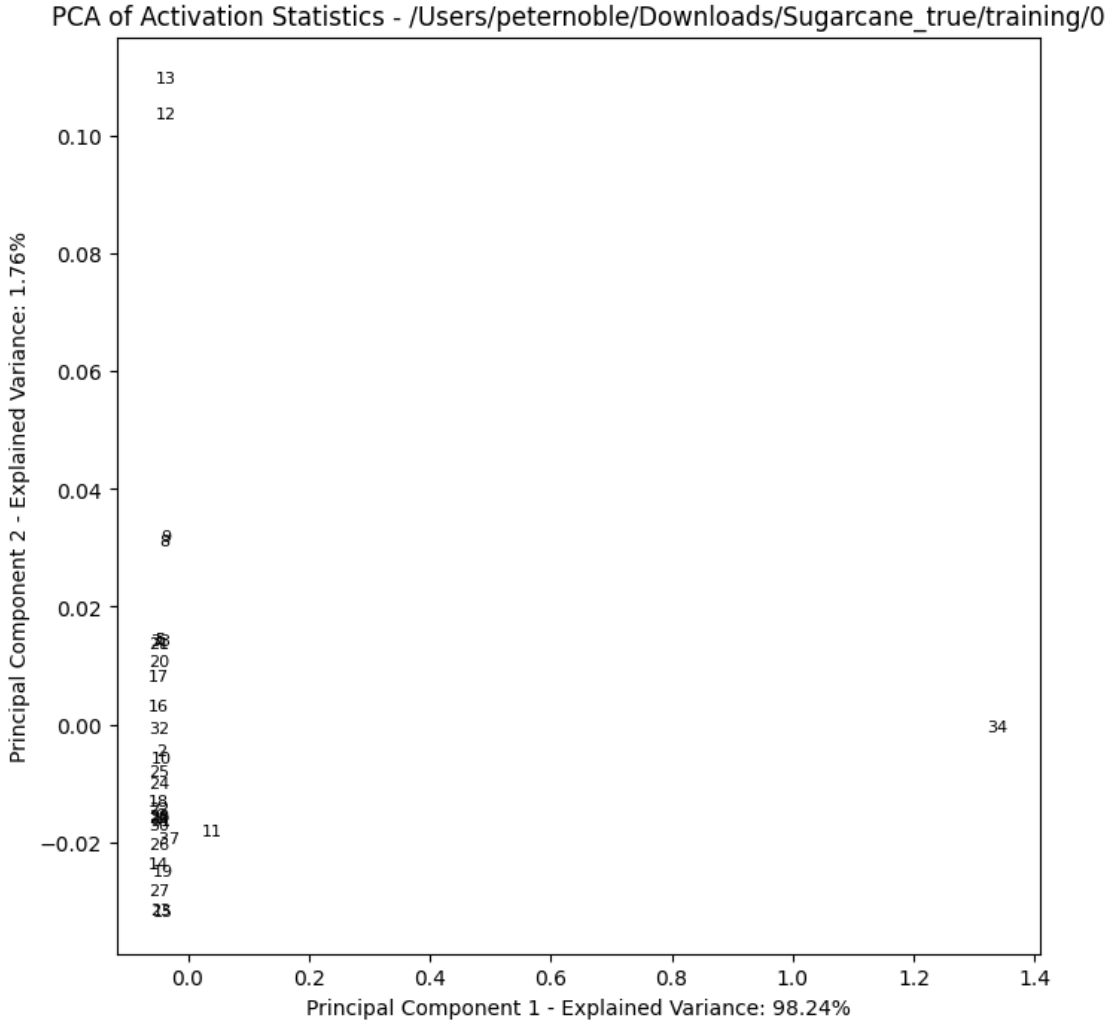
```
Layer 1 - Shape: torch.Size([1, 32, 489, 489])
Layer 2 - Shape: torch.Size([1, 32, 489, 489])
Layer 3 - Shape: torch.Size([1, 32, 489, 489])
Layer 4 - Shape: torch.Size([1, 32, 244, 244])
Layer 5 - Shape: torch.Size([1, 16, 245, 245])
Layer 6 - Shape: torch.Size([1, 16, 245, 245])
Layer 7 - Shape: torch.Size([1, 16, 245, 245])
Layer 8 - Shape: torch.Size([1, 16, 122, 122])
Layer 9 - Shape: torch.Size([1, 4, 123, 123])
Layer 10 - Shape: torch.Size([1, 4, 123, 123])
Layer 11 - Shape: torch.Size([1, 4, 123, 123])
Layer 12 - Shape: torch.Size([1, 4, 61, 61])
Layer 13 - Shape: torch.Size([1, 16, 62, 62])
Layer 14 - Shape: torch.Size([1, 16, 62, 62])
Layer 15 - Shape: torch.Size([1, 16, 62, 62])
Layer 16 - Shape: torch.Size([1, 16, 31, 31])
Layer 17 - Shape: torch.Size([1, 32, 32, 32])
Layer 18 - Shape: torch.Size([1, 32, 32, 32])
Layer 19 - Shape: torch.Size([1, 32, 32, 32])
Layer 20 - Shape: torch.Size([1, 32, 16, 16])
Layer 21 - Shape: torch.Size([1, 64, 17, 17])
Layer 22 - Shape: torch.Size([1, 64, 17, 17])
Layer 23 - Shape: torch.Size([1, 64, 17, 17])
Layer 24 - Shape: torch.Size([1, 64, 8, 8])
Layer 25 - Shape: torch.Size([1, 128, 9, 9])
Layer 26 - Shape: torch.Size([1, 128, 9, 9])
Layer 27 - Shape: torch.Size([1, 128, 9, 9])
Layer 28 - Shape: torch.Size([1, 128, 4, 4])
Layer 29 - Shape: torch.Size([1, 488, 5, 5])
Layer 30 - Shape: torch.Size([1, 488, 5, 5])
Layer 31 - Shape: torch.Size([1, 488, 5, 5])
Layer 32 - Shape: torch.Size([1, 488, 2, 2])
Layer 33 - Shape: torch.Size([1, 5])
```

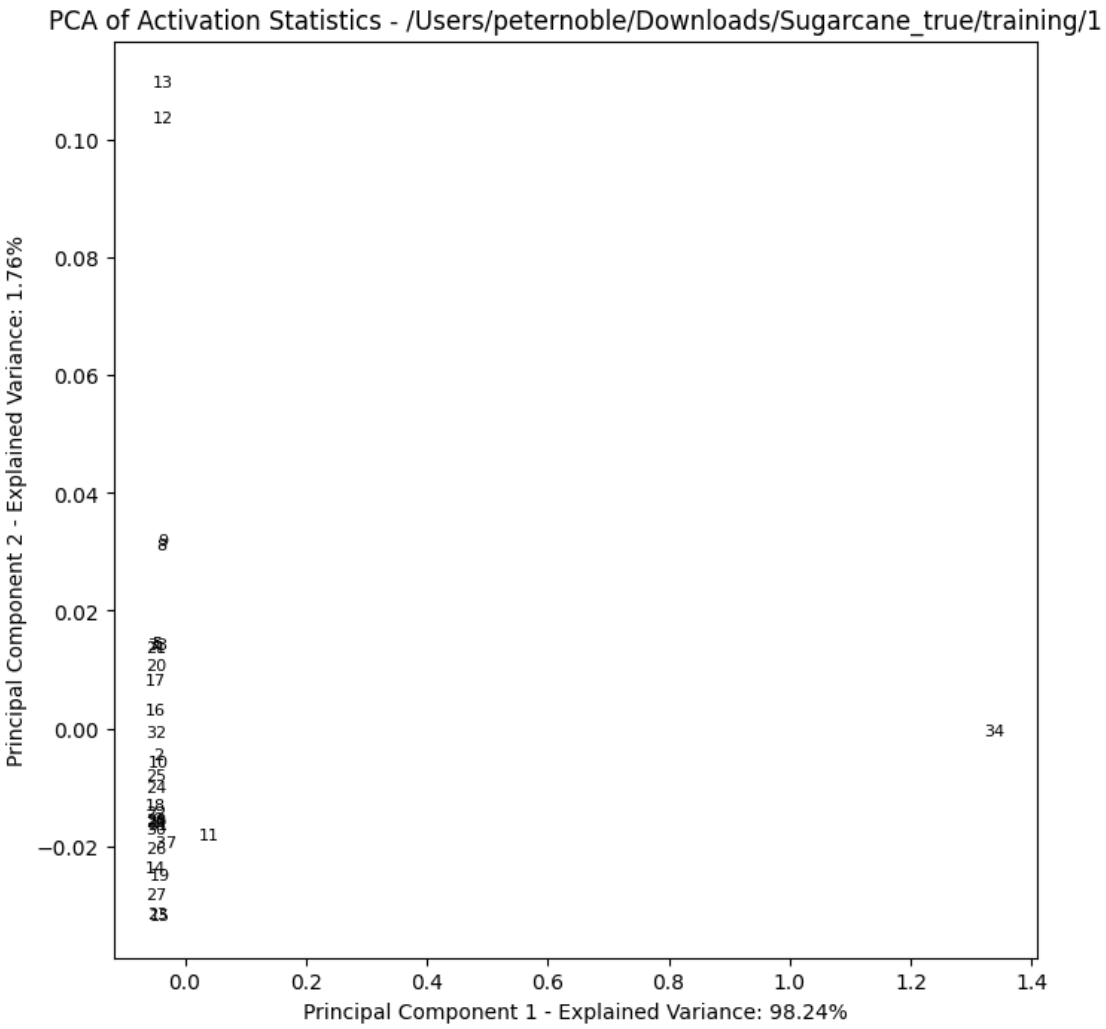
[15]:
```
# CustomCNN class definition goes here
import os
import numpy as np
import matplotlib.pyplot as plt
```

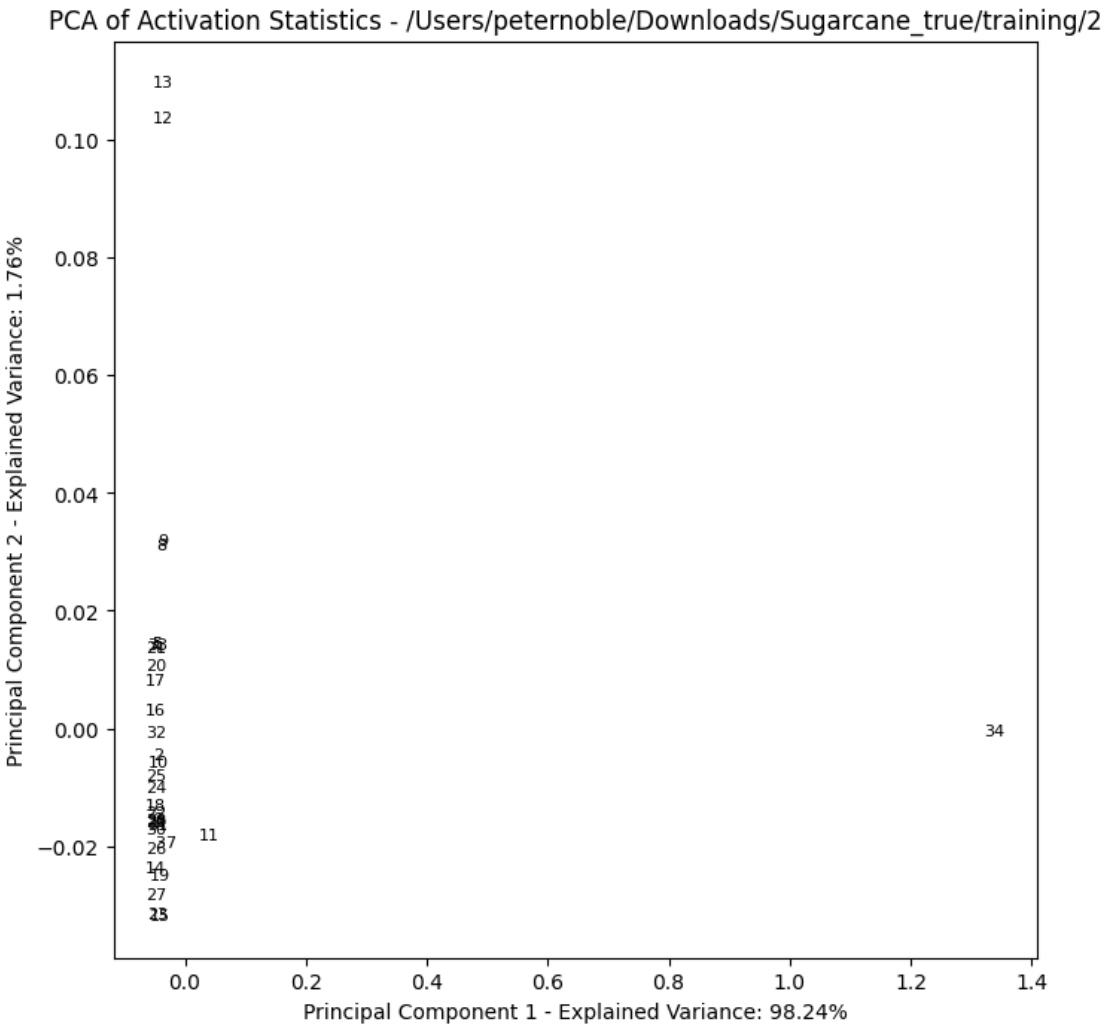
```
from sklearn.decomposition import PCA

# Function to visualize PCA results with labels and explained variance
def visualize_pca(mean_activations, var_activations, layer_numbers,␣
 ↪directory_path):
    # Combine mean and variance into a single matrix
    activations_matrix = list(zip(mean_activations, var_activations))

    # Apply PCA
    pca = PCA(n_components=2)
    activations_pca = pca.fit_transform(activations_matrix)

    # Get the contribution of explained variances
    contribution_pc1 = pca.explained_variance_ratio_[0] * 100
    contribution_pc2 = pca.explained_variance_ratio_[1] * 100

    # Visualize PCA results with dots labeled by layer numbers
    plt.figure(figsize=(8, 8))

    # Annotate explained variance on axes
 #   plt.text(activations_pca[:, 0].max() + 0.1, 0, f'PC1 ({contribution_pc1:.
 ↪2f}%)', fontsize=10, ha='left', va='center')
 #   plt.text(0, activations_pca[:, 1].max() + 0.1, f'PC2 ({contribution_pc2:.
 ↪2f}%)', fontsize=10, ha='center', va='bottom')

    # Label dots with layer numbers
    for i, (x, y) in enumerate(activations_pca):
        plt.scatter(x, y, alpha=0)  # Make the dot invisible
        plt.text(x, y, str(layer_numbers[i]), fontsize=8, ha='center',␣
 ↪va='center')

    plt.title(f'PCA of Activation Statistics - {directory_path}')
    plt.xlabel('Principal Component 1 - Explained Variance: {:.2f}%'.
 ↪format(contribution_pc1))
    plt.ylabel('Principal Component 2 - Explained Variance: {:.2f}%'.
 ↪format(contribution_pc2))
    plt.show()

# List of directory paths
directory_paths = [
    '/Users/peternoble/Downloads/Sugarcane_true/training/0',
    '/Users/peternoble/Downloads/Sugarcane_true/training/1',
    '/Users/peternoble/Downloads/Sugarcane_true/training/2',
    '/Users/peternoble/Downloads/Sugarcane_true/training/3',
    '/Users/peternoble/Downloads/Sugarcane_true/training/4',
]
```

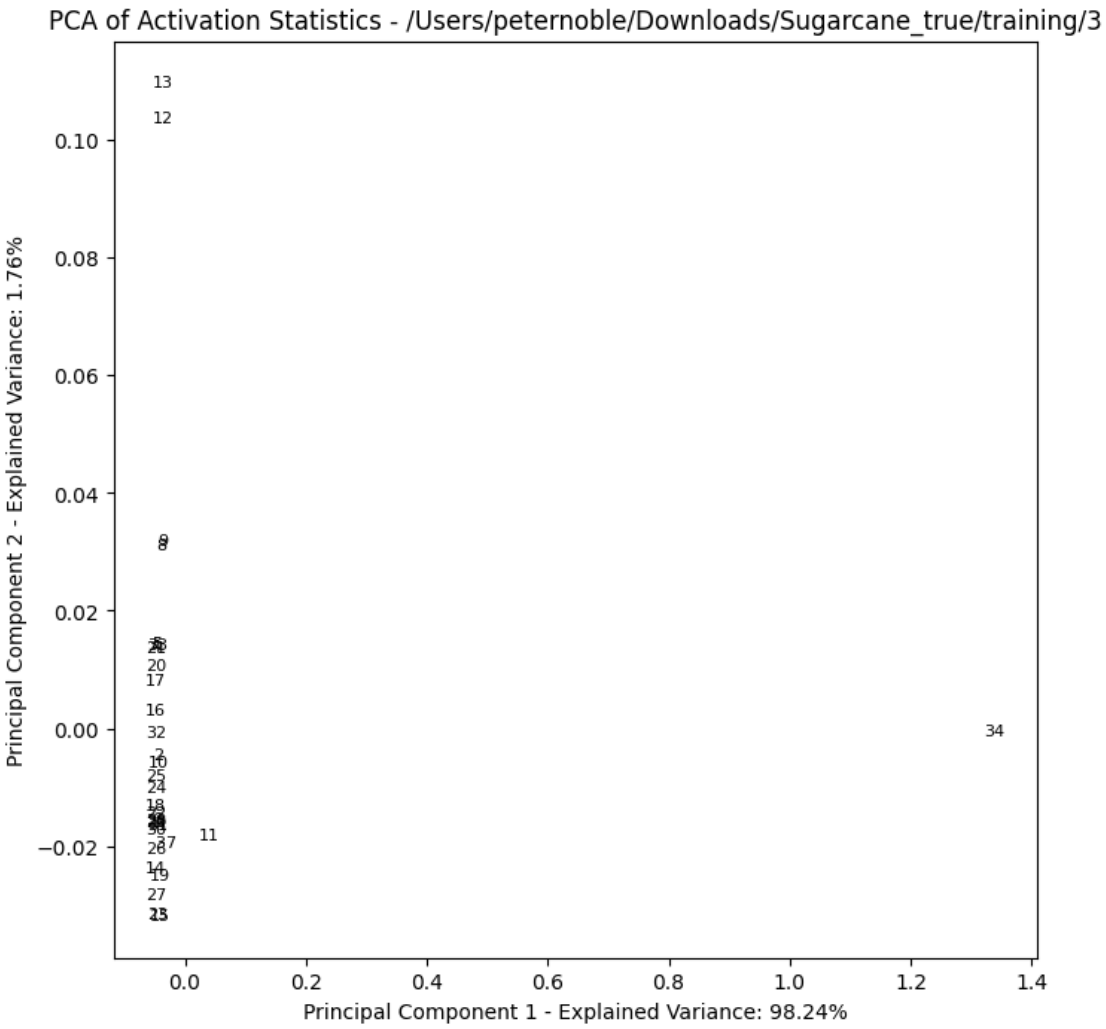
```
# Initialize lists to accumulate activation statistics for all datasets
all_mean_activations = []
all_var_activations = []

# Initialize layer numbers
layer_numbers = []

for directory_path in directory_paths:
    model = CustomCNN()

    # ... (rest of the code remains unchanged)

    # Accumulate activation statistics for the current dataset
    all_mean_activations.extend(mean_activations)
    all_var_activations.extend(var_activations)

    # Add layer numbers for the current dataset
    layer_numbers.extend([i + 2 for i in range(len(mean_activations))])

    # Visualize PCA of activation statistics for each dataset
    visualize_pca(mean_activations, var_activations, layer_numbers,␣
 ↪directory_path)

# Combine mean and variance into a single matrix for all datasets
all_activations_matrix = list(zip(all_mean_activations, all_var_activations))

# Apply PCA on the combined activation statistics
pca_all_datasets = PCA(n_components=2)
activations_pca_all_datasets = pca_all_datasets.
 ↪fit_transform(all_activations_matrix)

# Visualize PCA results for all datasets
plt.figure(figsize=(8, 8))

# Get the contribution of explained variances
contribution_pc1 = pca_all_datasets.explained_variance_ratio_[0] * 100
contribution_pc2 = pca_all_datasets.explained_variance_ratio_[1] * 100

# Annotate explained variance on axes
#plt.text(activations_pca_all_datasets[:, 0].max() + 0.1, 0, f'PC1␣
 ↪({contribution_pc1:.2f}%)', fontsize=10, ha='left', va='center')
#plt.text(0, activations_pca_all_datasets[:, 1].max() + 0.1, f'PC2␣
 ↪({contribution_pc2:.2f}%)', fontsize=10, ha='center', va='bottom')

# Label dots with layer numbers
```

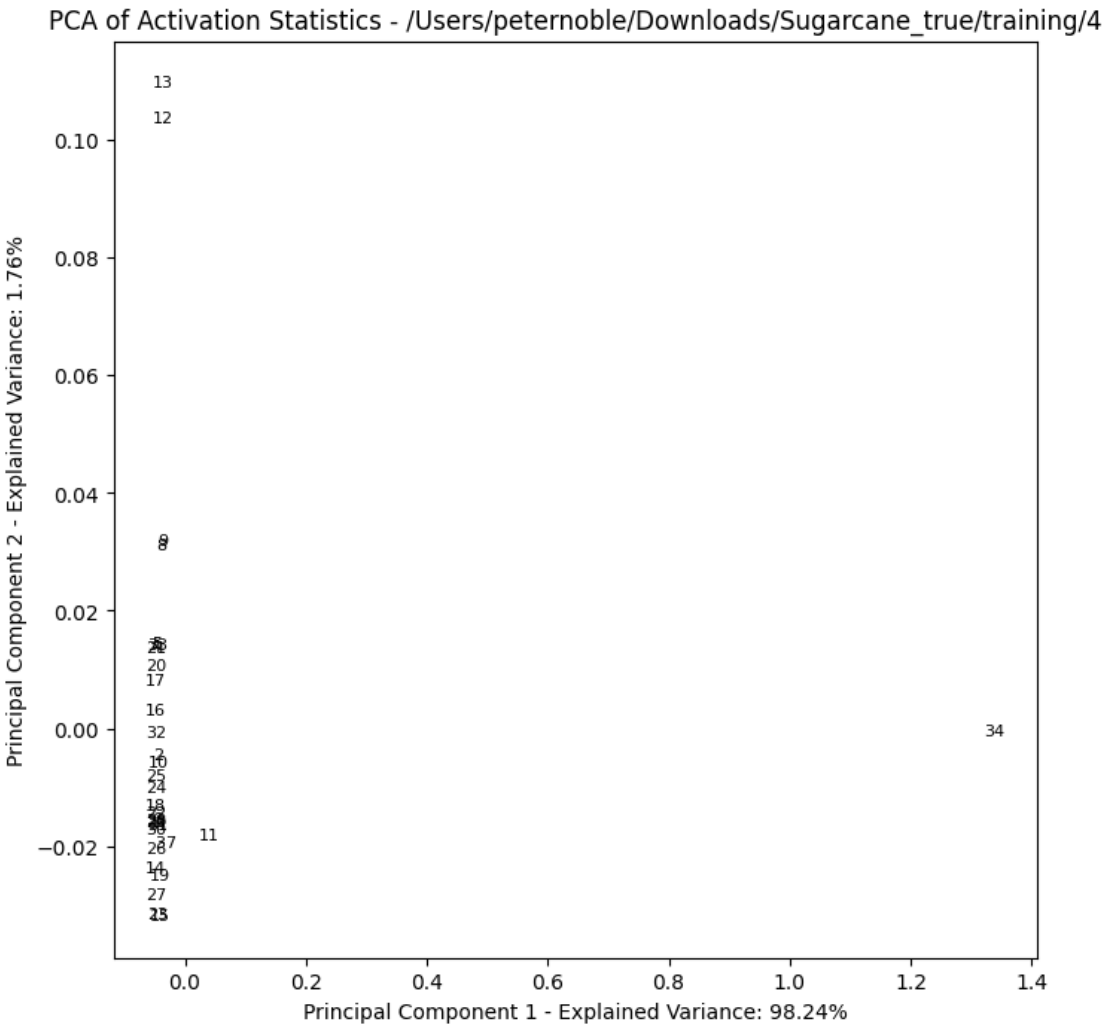
```
for i, (x_val, y_val) in enumerate(zip(activations_pca_all_datasets[:, 0],␣
 ↪activations_pca_all_datasets[:, 1])):
    plt.scatter(x_val, y_val, color='white', alpha=0)  # Make the dot invisible
    plt.text(x_val, y_val, str(layer_numbers[i]), color='white', fontsize=8,␣
 ↪ha='center', va='center')

plt.scatter(activations_pca_all_datasets[:, 0], activations_pca_all_datasets[:,␣
 ↪1])
plt.title('PCA of Activation Statistics - All Datasets')
plt.xlabel('Principal Component 1 - Explained Variance: {:.2f}%'.
 ↪format(contribution_pc1))
plt.ylabel('Principal Component 2 - Explained Variance: {:.2f}%'.
 ↪format(contribution_pc2))
plt.show()
```

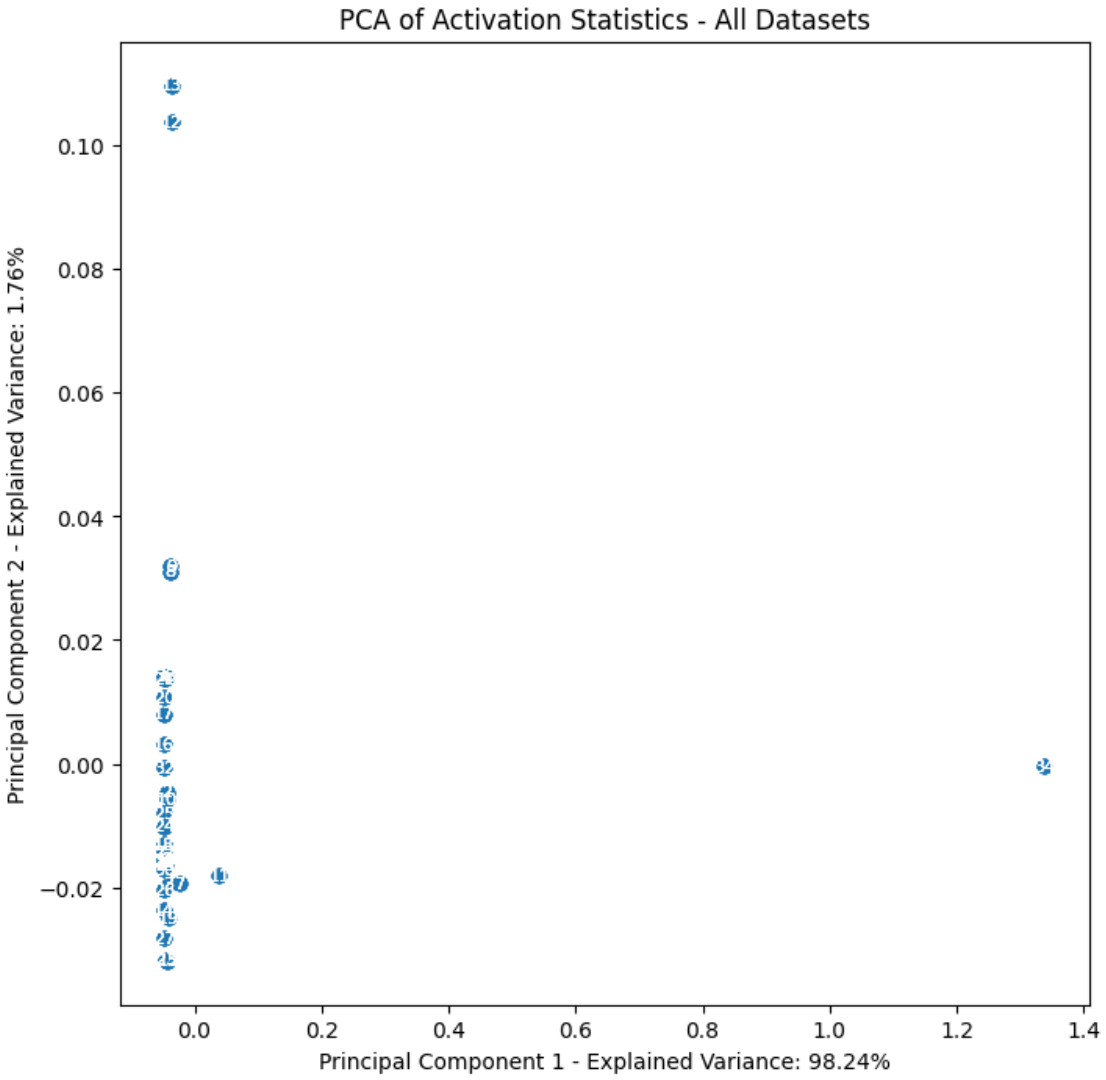



PCA of Activation Statistics - /Users/peternoble/Downloads/Sugarcane_true/training/1
Principal Component 2 - Explained Variance: 1.76%
Principal Component 1 - Explained Variance: 98.24%
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
13
12
34
11


PCA of Activation Statistics - /Users/peternoble/Downloads/Sugarcane_true/training/2
Principal Component 2 - Explained Variance: 1.76%
Principal Component 1 - Explained Variance: 98.24%
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
13
12
34
11


PCA of Activation Statistics - /Users/peternoble/Downloads/Sugarcane_true/training/3
Principal Component 2 - Explained Variance: 1.76%
Principal Component 1 - Explained Variance: 98.24%
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
13
12
34
11


PCA of Activation Statistics - /Users/peternoble/Downloads/Sugarcane_true/training/4
Principal Component 2 - Explained Variance: 1.76%
Principal Component 1 - Explained Variance: 98.24%
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
13
12
34
11


PCA of Activation Statistics - All Datasets
Principal Component 2 - Explained Variance: 1.76%
Principal Component 1 - Explained Variance: 98.24%
0.10
0.08
0.06
0.04
0.02
0.00
−0.02
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

[ ]:

[ ]:

[ ]: