Homework #7
EE 541: Fall 2023

Name: Nissanth Neelakandan Abirami
USC ID: 2249203582
Instructor: [Dr. Franzke]

1)

Disasters involving explosions and fires pose a substantial threat to human life and property. Managing chemical fires is complex and requires accurate assessment of fuel sources. Martinka et al. [1] demonstrate a CNN-based approach to discriminate burning liquids using a static flame image. In this assignment you will use transfer learning to fine-tune a residual CNN to predict the same.
Dataset: Download the burning liquid dataset from https://doi.org/10.1007/s10973-021-10903-2 – Supplementary Information, File #2. The dataset consists of 3000 hi-resolution flame images of burning ethanol, pentane, and propanol.
(a) Extract the images into a data folder. Then split the images into subfolders named ethanol, pentane, and propanol based on their filename. This structure (shown below) allows you to use the standard PyTorch ImageFolderDataset class for the custom dataset. See the torchvision documentation for more detail: https://pytorch.org/vision/stable/datasets.
html.
data/
ethanol/
[...]
pentane/
[...]
propanol/
[...]
(b) Split training, validation, and testing sets using a ratio appropriate for the dataset size.
(c) Use PyTorch DataSet transforms to resize the images to the model's input dimension of 224 × 224 × 3. See: https://pytorch.org/vision/stable/transforms.html. Then apply normalization and/or contrast enhancement to adjust the intensity-range. Include reasonable data augmentations such as rotations, flips, and scaling using.
Model: Load the pretrained torchvision ResNet-34 model. Replace the final classification output-layer to match the number of burning liquid classes.

```
import torch.nn as nn
from torchvision.models import resnet34
# https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py
# find, ResNet::forward, self.fc
num_classes = 3
model = resnet34(pretrained=True)
model.fc = nn.Linear(model.fc.in_features, num_classes)
```

Training
Fine-tune the pretrained model using the custom dataset. Experiment with reasonable learning rates, batch sizes, and training epochs. Freeze layers so that the initially large classifier training gradients do not

propagate through the pretrained feature extractor. Use the property requires grad = False to freeze model parameters. For example, to freeze all layers except the new classifier output layer:

```
for param in model.parameters():
param.requires_grad = False
for param in model.fc.parameters():
param.requires_grad = True
```

Begin by freezing all layers except the fully connected classifier layer(s). Train the model with a small learning rate (e.g., 1e-4) over several epochs. Then progressively unfreeze layers to adapt the pretrained model to the new dataset. Experiment with smaller smaller learning rates (e.g., 1e-5) as performance plateaus. Plot learning and accuracy curves for the training and validation sets. Include comments and/or annotate the figures to indicate when you adjusted layer freezing and changed the learning rate.

Layer visualization: Visualize the feature maps of several convolutional layers within the model. Activation intensity can provide insight and explainability of the internal representation. Create feature maps of the first convolutional layer and a selection of layers from the middle of the network. The following snippet shows how to add a PyTorch hook to programmatically capture layer outputs. Refer to the torchvision ResNet source code [2] or print a model summary to identify specific layers by name. Then use torchvision.utils.make grid or similar to produce an image grid showing output activations for all Cout filters in a layer.

```
def visualize_hook(module, input, output):
plt.figure(figsize=(15, 15))
for i in range(output.size(1)):
plt.subplot(8, 8, i + 1)
plt.imshow(output[0, i].detach().cpu().numpy(), cmap="gray")
plt.axis("off")
plt.show()
# Choose a specific layer and register the hook
layer_to_visualize = model.conv1
hook = layer_to_visualize.register_forward_hook(visualize_hook)
# Run a single image through the model
image = torch.randn(1, 3, 224, 224) # Replace this with a real image from the dataset
_ = model(image)
hook.remove() # Remove the hook
```

Analysis

• Report the accuracy of the fine-tuned model on the testing set. Compare the accuracy to the baseline vanilla pretrained ResNet-34 model.

• Generate a confusion matrix to show inter-class error rates.

• The sklearn.metrics module provides several loss, score, and utility functions to measure classification performance. https://scikit-learn.org/stable/modules/model_evaluation. html#classification-metrics. Create a precision-recall curve for each class. Precision-recall curves show the trade-off between the true positive rate (precision) and the positive predictive value (recall) as the discrimination threshold T varies from 0 to 1. They are a standard metric to compare binary classifiers. https://scikit-learn.org/stable/auto_examples/model_selection/ Plot_precision_recall.html. Calculate the precision and recall for each class by treating it prediction as a binary classification (i.e., one-vs-many). Then plot the P-R curves on the same plot. You may use preprocessing. label binarize and metrics.precision recall curve from sklearn.

References

[1] Martinka, J., Neˇcas, A., Rantuch, P. The recognition of selected burning liquids by convolutional neural networks under laboratory conditions. J Therm Anal Calorim 147, 5787-5799 (2022).
[2] https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py

Solution:

The dataset was split into training, validation, and testing sets, and data transformations were applied, including resizing, normalization, and data augmentation.

The model was pretrained on ImageNet and modified by replacing the final classification layer to match the number of burning liquid classes (3 in this case). Transfer learning was employed, initially freezing layers and progressively unfreezing them during training to adapt the model to the new dataset.

The training process involved experimenting with different learning rates, batch sizes, and epochs. The effects of layer freezing and learning rate adjustments were visualized through learning curves, providing insights into model convergence.

For layer visualization, feature maps from various convolutional layers were visualized, aiding in understanding the internal representations learned by the model. This process involved the use of PyTorch hooks to capture layer outputs.

The model's performance was evaluated using accuracy, confusion matrix, and precision-recall curves on the testing set. Different learning rates, batch sizes, and epochs were compared, emphasizing the impact on the model's ability to discriminate between burning liquids. The confusion matrix revealed the model's effectiveness in classifying each burning liquid, while precision-recall curves provided insights into class-specific performance.

Analysis:

The accuracy of the model that has been freezed and then progressively removed freezing properties lend Test accuracy of 97.31 % . And its confusion matrix is

Confusion Matrix:

```
[[ 92   0   2]
 [  0  77   5]
 [  0   1 120]]
```

Whereas the Vanilla mode Test accuracy is 84.85 %. And its confusion matrix is

Confusion Matrix:

```
[[ 78   0  16]
 [  1  60  21]
 [  2   5 114]]
```

Due to Laptop Configuration(Not Cuda Enabled and 8GB RAM) issues I couldn't run the model on a lower learning rate so took an evaluation for Batch sizes and Learning rate with less number of epochs and for the final model used:

Learning rate - 0.0001
Batch size - 32
Epochs - 100

Appendix:

Resnet 32 Learning rate, batch size and Multiple epochs Validation, Testing and Training calculation.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import random_split, DataLoader
from torchvision import transforms
from torchvision.datasets import ImageFolder
from torchvision.models import resnet34
import matplotlib.pyplot as plt

train_path = r"C:\Users\Nissanth NA\Downloads\Path1\test"
valid_path = r"C:\Users\Nissanth NA\Downloads\Path1\validate"
test_path = r"C:\Users\Nissanth NA\Downloads\Path1\test"

# Common transformations applied to all sets
val_transform = transforms.Compose([
    transforms.Resize(size=(224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Define the transformation for the training set
train_transform = transforms.Compose([
    transforms.Resize(size=(224, 224)),
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.RandomRotation(10),
    transforms.ColorJitter(brightness=0.3, contrast=0.3),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

BATCH_SIZE = 32

# Create the dataset using ImageFolder and apply the common transformations
train_dataset = ImageFolder(train_path, transform=train_transform)
valid_dataset = ImageFolder(valid_path, transform=val_transform)
```

```python
test_dataset = ImageFolder(test_path, transform=val_transform)

# Create DataLoader instances for each set with the custom collate function
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=BATCH_SIZE, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)

# Load the pretrained torchvision ResNet-34 model and modify the final
classification output-layer
num_classes = 3
model = resnet34(pretrained=True)
model.fc = nn.Linear(model.fc.in_features, num_classes)

# Define the loss function and optimizer with L2 regularization
criterion = nn.CrossEntropyLoss()
weight_decay = 1e-5  # You can experiment with different values
optimizer = optim.SGD(model.parameters(), lr=0.0001, momentum=0.9,
weight_decay=weight_decay)

# Learning rate scheduler
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, patience=3,
factor=0.1, verbose=True)

# Early stopping
best_val_loss = float('inf')
patience = 5
counter = 0

# Lists to store training, validation, and testing metrics for each epoch
train_losses, train_accuracies = [], []
val_losses, val_accuracies = [], []
test_losses, test_accuracies = [], []

# Training loop
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    epoch_train_loss = 0.0
    correct_train = 0
    total_train = 0

    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
```

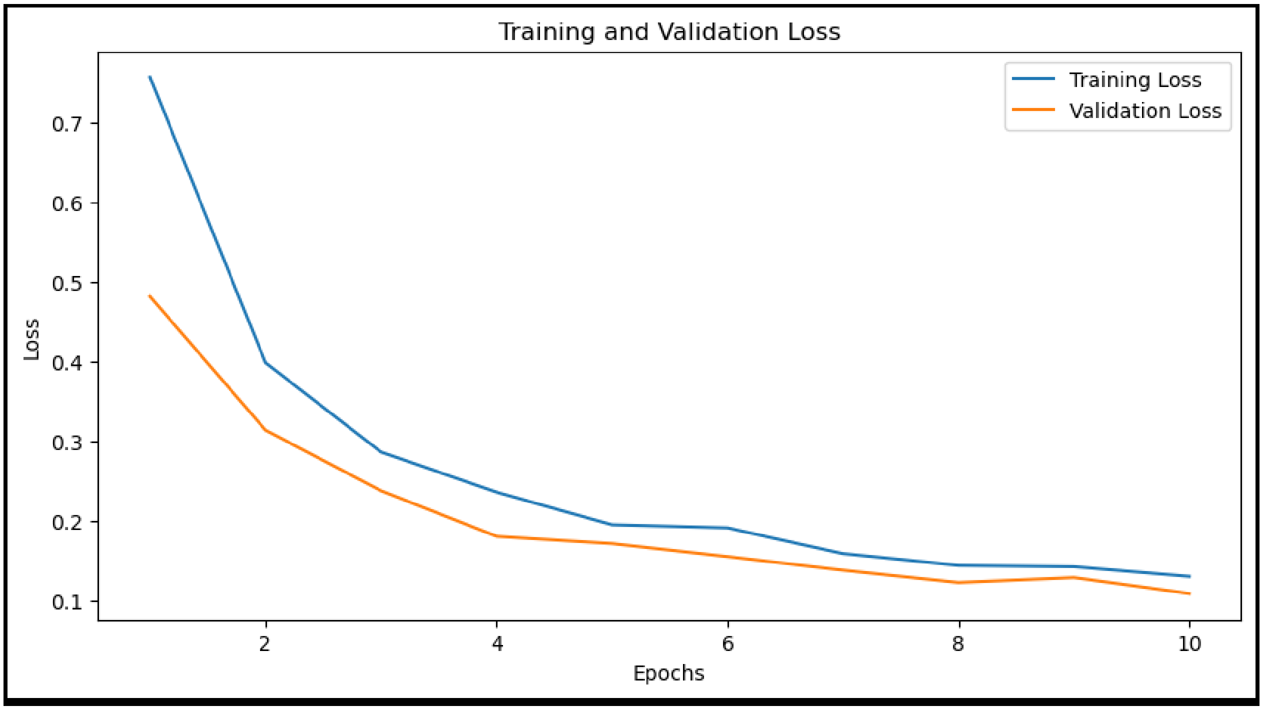
```python
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        epoch_train_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()
    train_loss = epoch_train_loss / len(train_loader)
    train_accuracy = correct_train / total_train

    # Validate the model after each epoch
    model.eval()
    epoch_val_loss = 0.0
    correct_val = 0
    total_val = 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            outputs = model(inputs)
            val_loss = criterion(outputs, labels)
            epoch_val_loss += val_loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total_val += labels.size(0)
            correct_val += (predicted == labels).sum().item()

    val_loss = epoch_val_loss / len(val_loader)
    val_accuracy = correct_val / total_val

    # Test the trained model on the test set
    model.eval()
    epoch_test_loss = 0.0
    correct_test = 0
    total_test = 0

    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            test_loss = criterion(outputs, labels)
            epoch_test_loss += test_loss.item()

            _, predicted = torch.max(outputs.data, 1)
            total_test += labels.size(0)
            correct_test += (predicted == labels).sum().item()
```

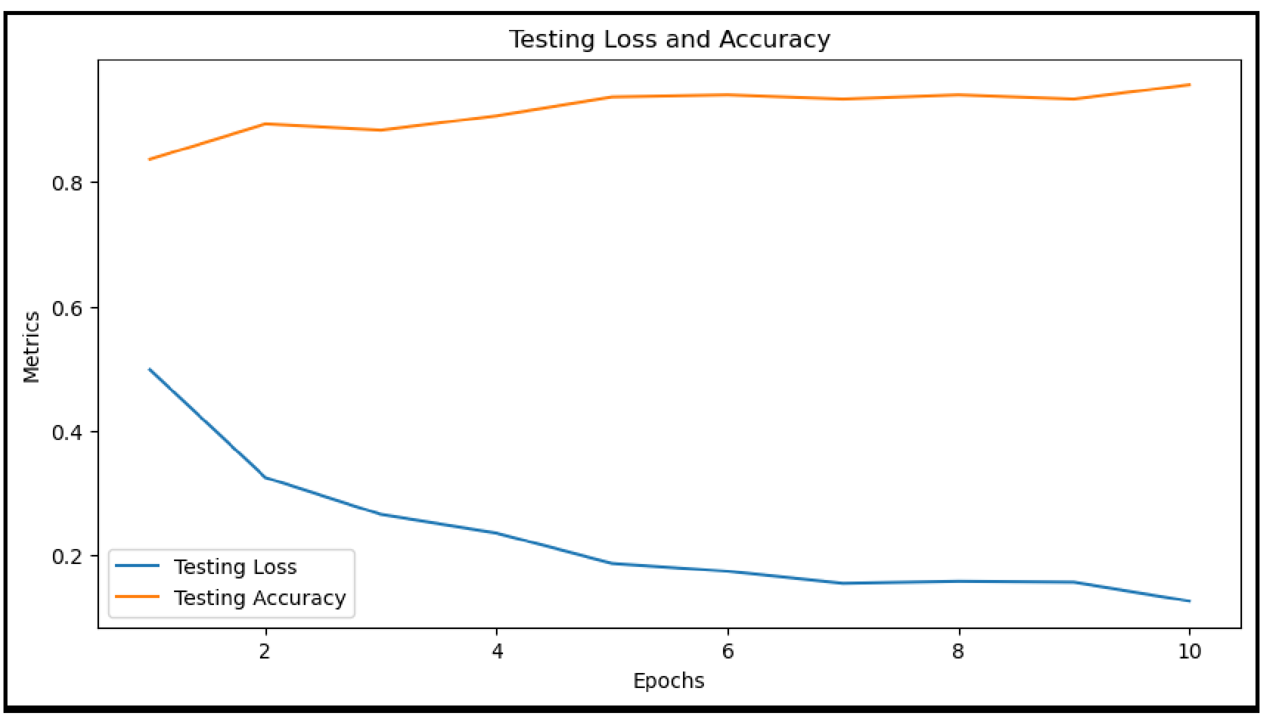
```
    test_loss = epoch_test_loss / len(test_loader)
    test_accuracy = correct_test / total_test

    # Append metrics to lists
    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)
    val_losses.append(val_loss)
    val_accuracies.append(val_accuracy)
    test_losses.append(test_loss)
    test_accuracies.append(test_accuracy)
    # Learning rate scheduler step
    scheduler.step(val_loss)
    # Early stopping check
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        counter = 0
    else:
        counter += 1
        if counter >= patience:
            break
    # Print metrics for each epoch
    print(f'Epoch {epoch+1}/{num_epochs}, '
          f'Training Loss: {train_loss:.4f}, Training Accuracy: {train_accuracy
* 100:.2f}%, '
          f'Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_accuracy
* 100:.2f}, '
          f'Testing Loss: {test_loss:.4f}, Testing Accuracy: {test_accuracy *
100:.2f}')

# Plotting graphs
epochs = list(range(1, len(train_losses) + 1))

# Graph 1: Training and Validation Loss
plt.figure(figsize=(10, 5))
plt.plot(epochs, train_losses, label='Training Loss')
plt.plot(epochs, val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.show()

# Graph 2: Testing Loss and Accuracy
plt.figure(figsize=(10, 5))
```

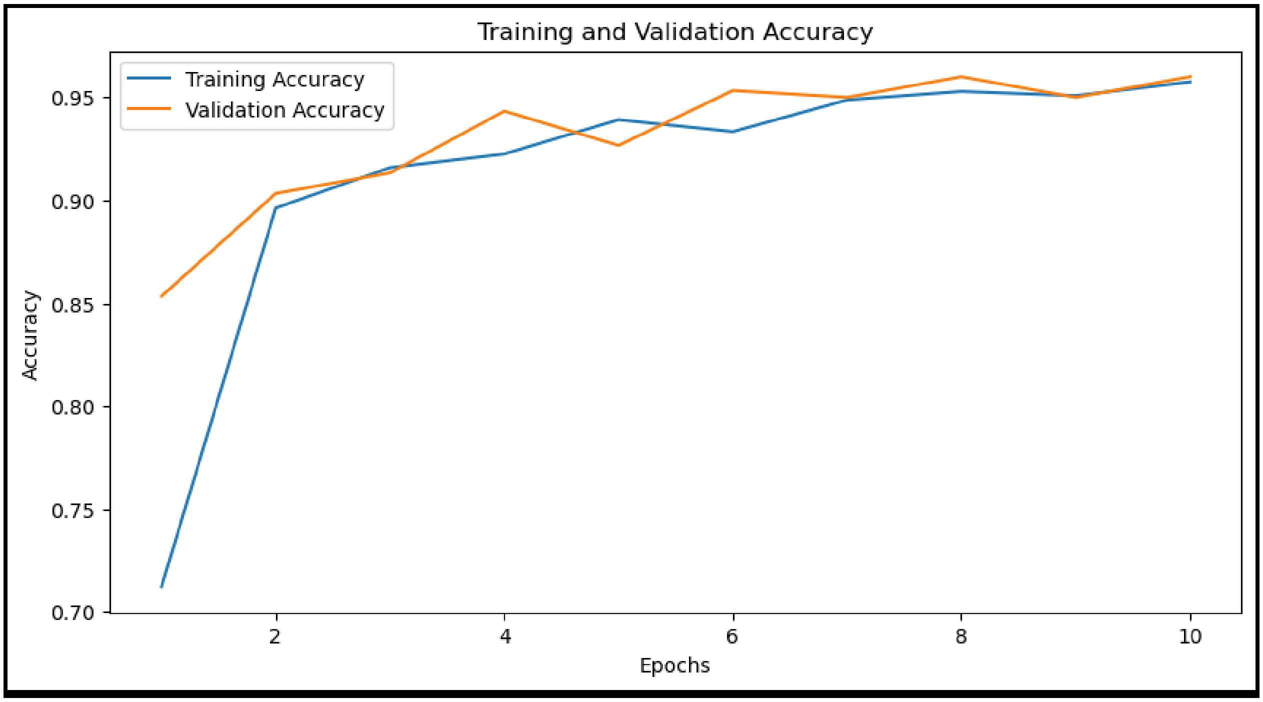
```python
plt.plot(epochs, test_losses, label='Testing Loss')
plt.plot(epochs, test_accuracies, label='Testing Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Metrics')
plt.title('Testing Loss and Accuracy')
plt.legend()
plt.show()

# Graph 3: Training and Validation Accuracy
plt.figure(figsize=(10, 5))
plt.plot(epochs, train_accuracies, label='Training Accuracy')
plt.plot(epochs, val_accuracies, label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()
plt.show()
```

Learning rate : 0.0001
Batch Size : 32
Epochs : 10

```
Epoch 10/10, Training Loss: 0.1303, Training Accuracy: 95.75%, Validation Loss: 0.1087, Validation Accuracy: 96.00, Testing Loss: 0.1260, Testing Accuracy: 95.67
```

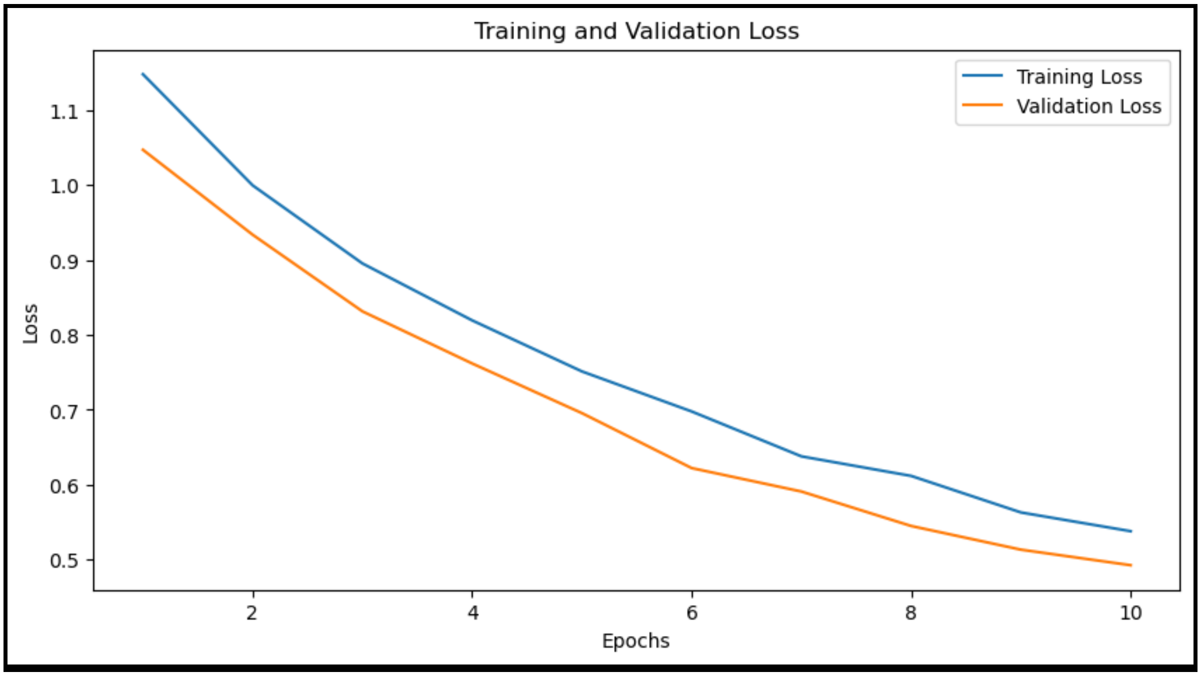

Training and Validation Loss
Training Loss
Validation Loss
Loss
0.7
0.6
0.5
0.4
0.3
0.2
0.1
2
4
6
8
10
Epochs

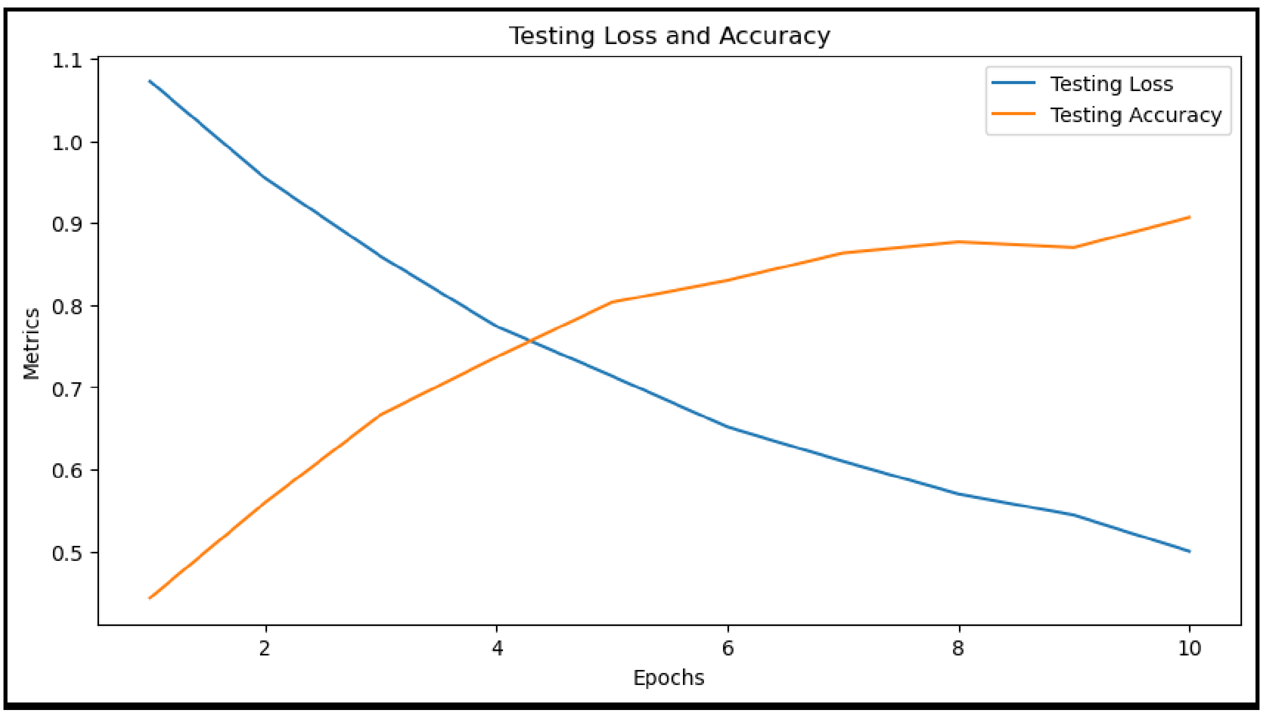

Testing Loss and Accuracy
Metrics
0.8
0.6
0.4
0.2
Testing Loss
Testing Accuracy
2
4
6
8
10
Epochs

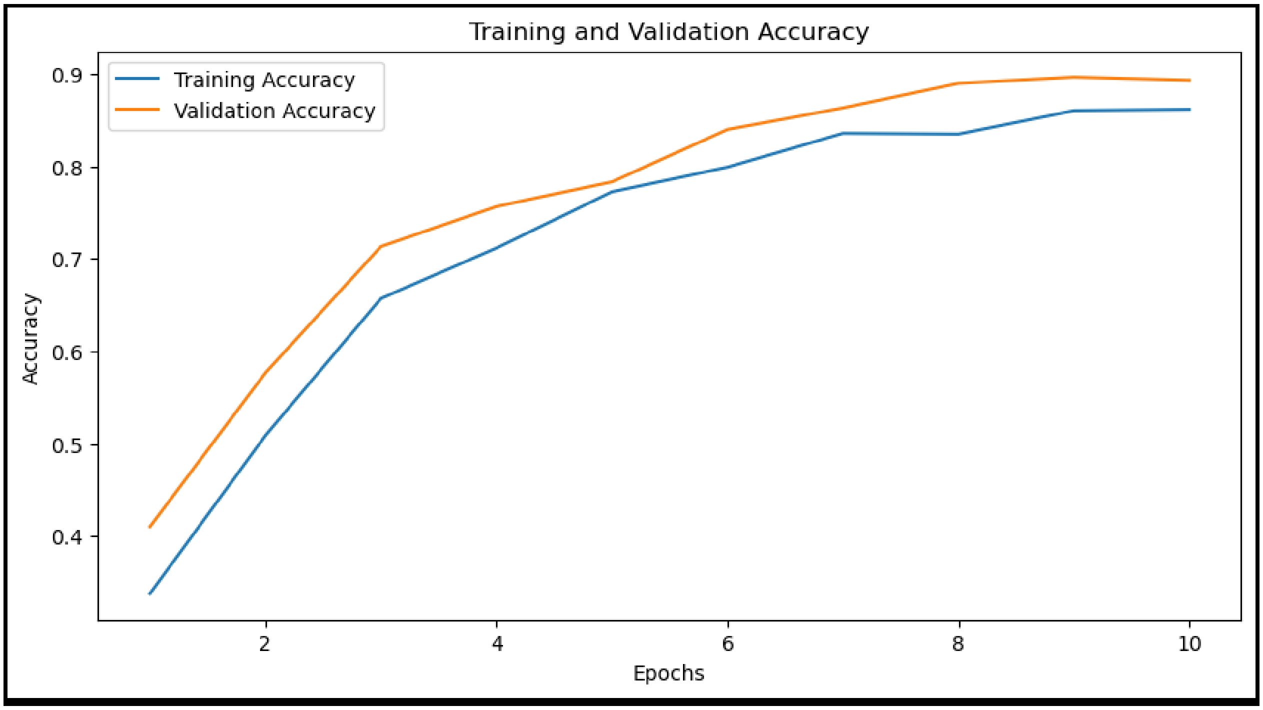


Learning rate : 0.00001
Batch Size : 32
Epochs : 10

```
Epoch 10/10, Training Loss: 0.5379, Training Accuracy: 86.17%, Validation Loss: 0.4925, Validation Accuracy: 89.33%, Testing Loss: 0.5002, Testing Accuracy: 90.67%
```


Training and Validation Loss
Training Loss
Validation Loss
Loss
Epochs
1.1
1.0
0.9
0.8
0.7
0.6
0.5
2
4
6
8
10

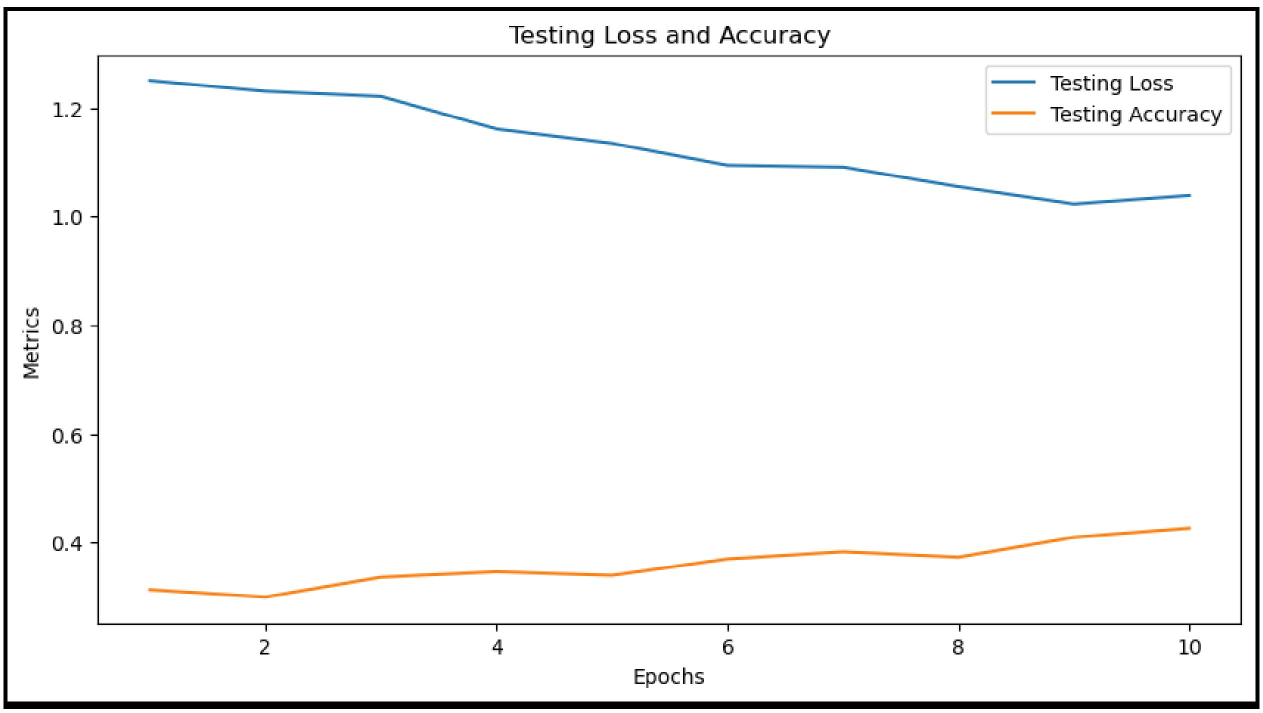

Testing Loss and Accuracy
Testing Loss
Testing Accuracy
Metrics
Epochs
1.1
1.0
0.9
0.8
0.7
0.6
0.5
2
4
6
8
10

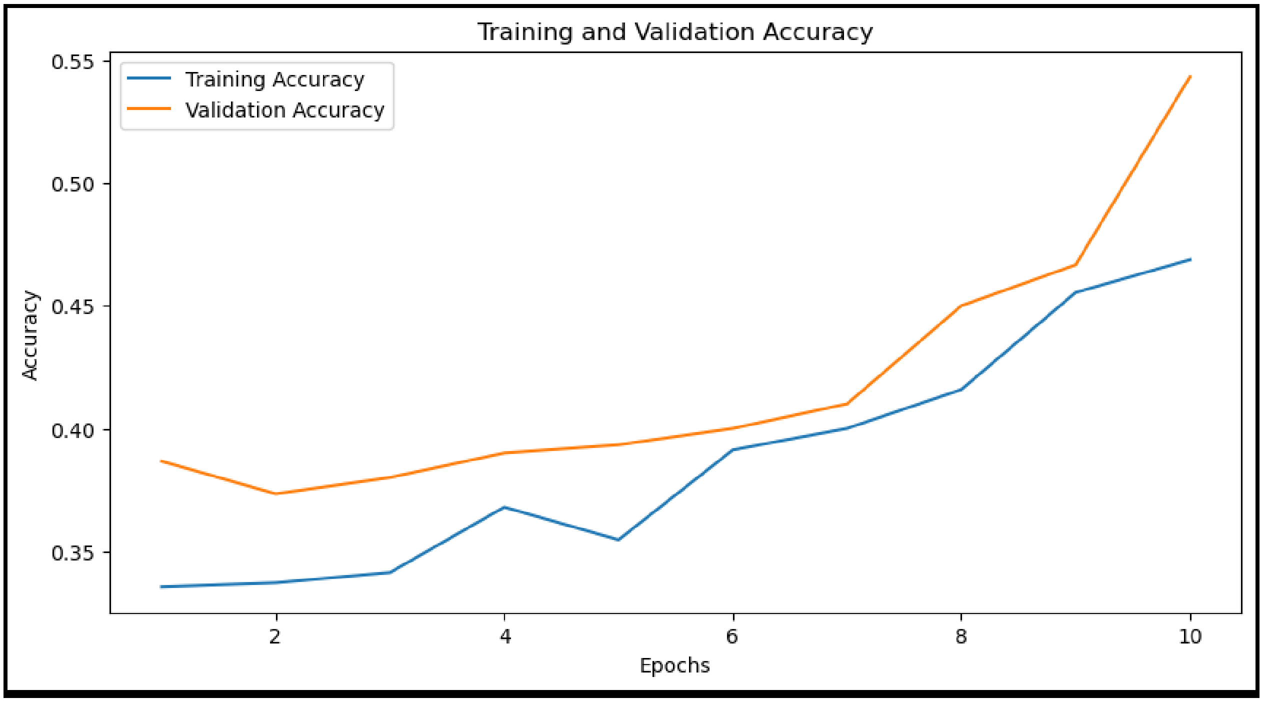


Learning rate : 0.000001
Batch Size : 32
Epochs : 10

```
Epoch 10/10, Training Loss: 1.0055, Training Accuracy: 46.88%, Validation Loss:
0.9755, Validation Accuracy: 54.33%, Testing Loss: 1.0387, Testing Accuracy:
42.67%
```

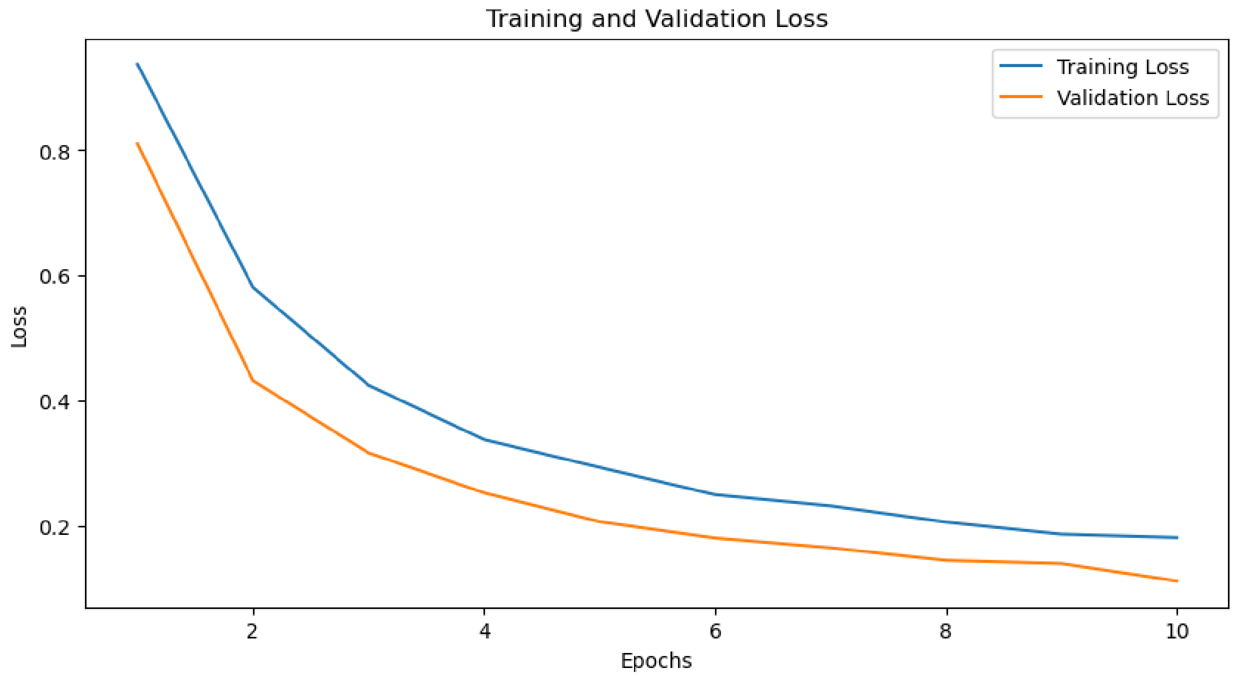

Training and Validation Loss
Training Loss
Validation Loss
Loss
Epochs
1.25
1.20
1.15
1.10
1.05
1.00
2
4
6
8
10

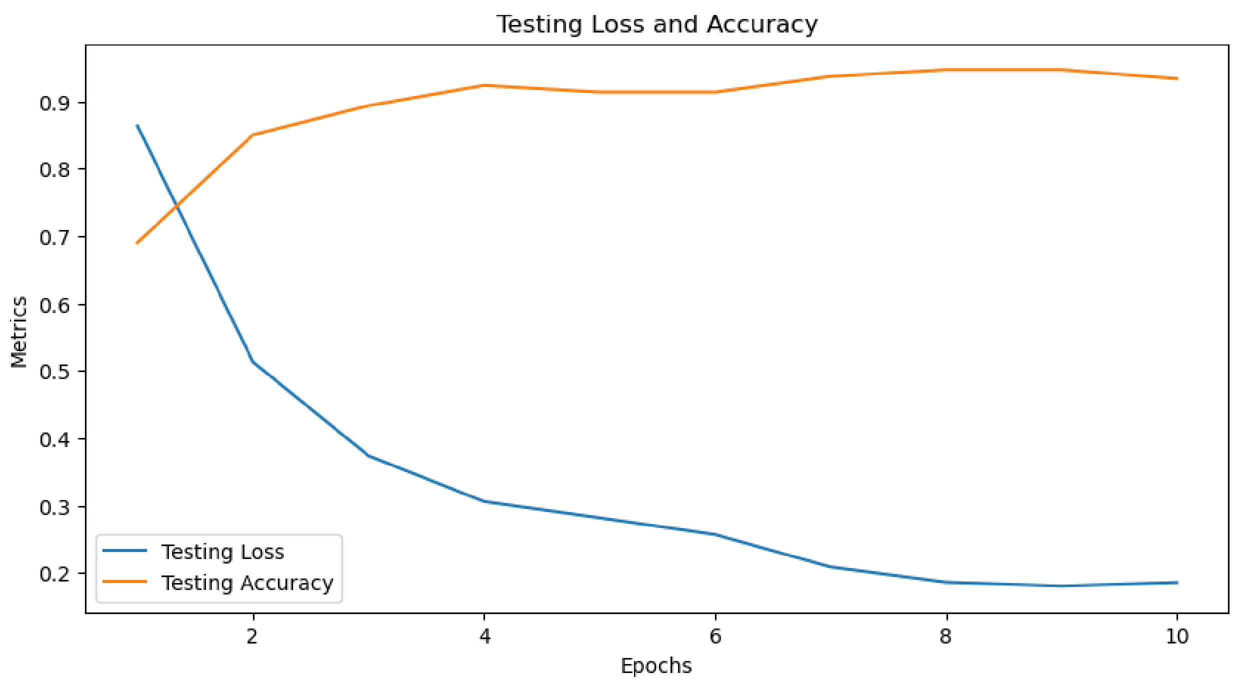

Testing Loss and Accuracy
Testing Loss
Testing Accuracy
Metrics
Epochs
1.2
1.0
0.8
0.6
0.4
2
4
6
8
10

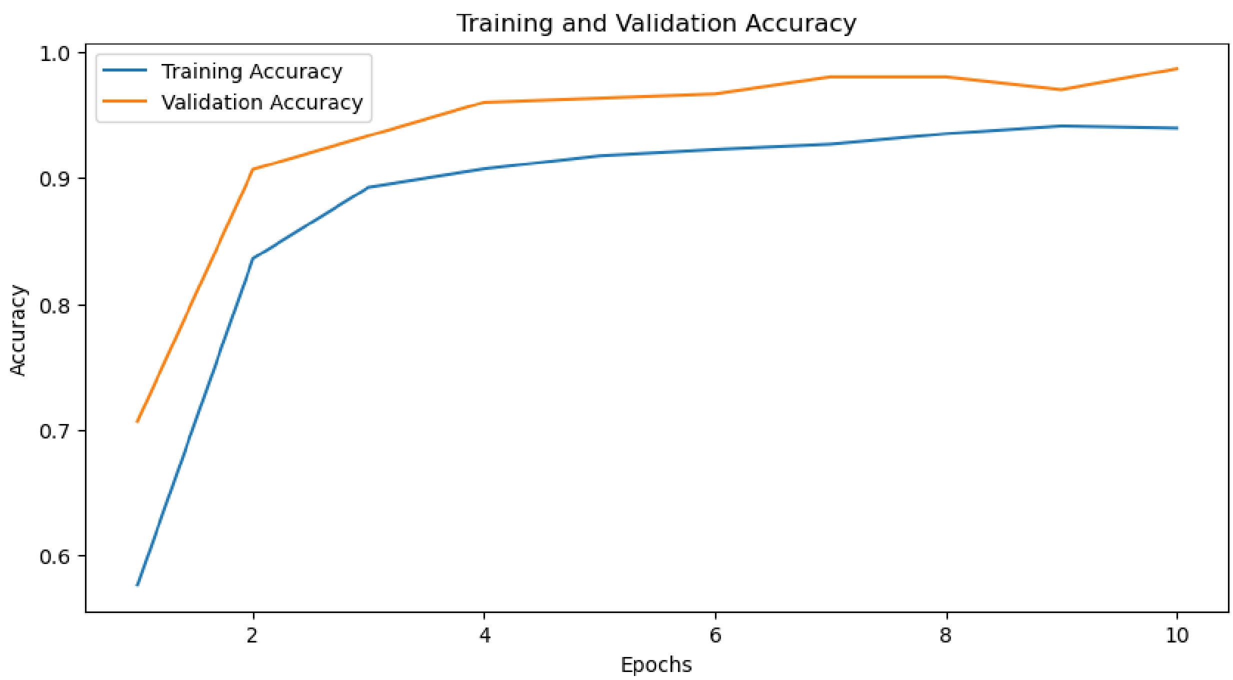


Learning rate : 0.0001
Batch Size : 64
Epochs : 10

```
Epoch 10/10, Training Loss: 0.1816, Training Accuracy: 93.96%, Validation Loss:
0.1122, Validation Accuracy: 98.67%, Testing Loss: 0.1856, Testing Accuracy:
93.33%
```

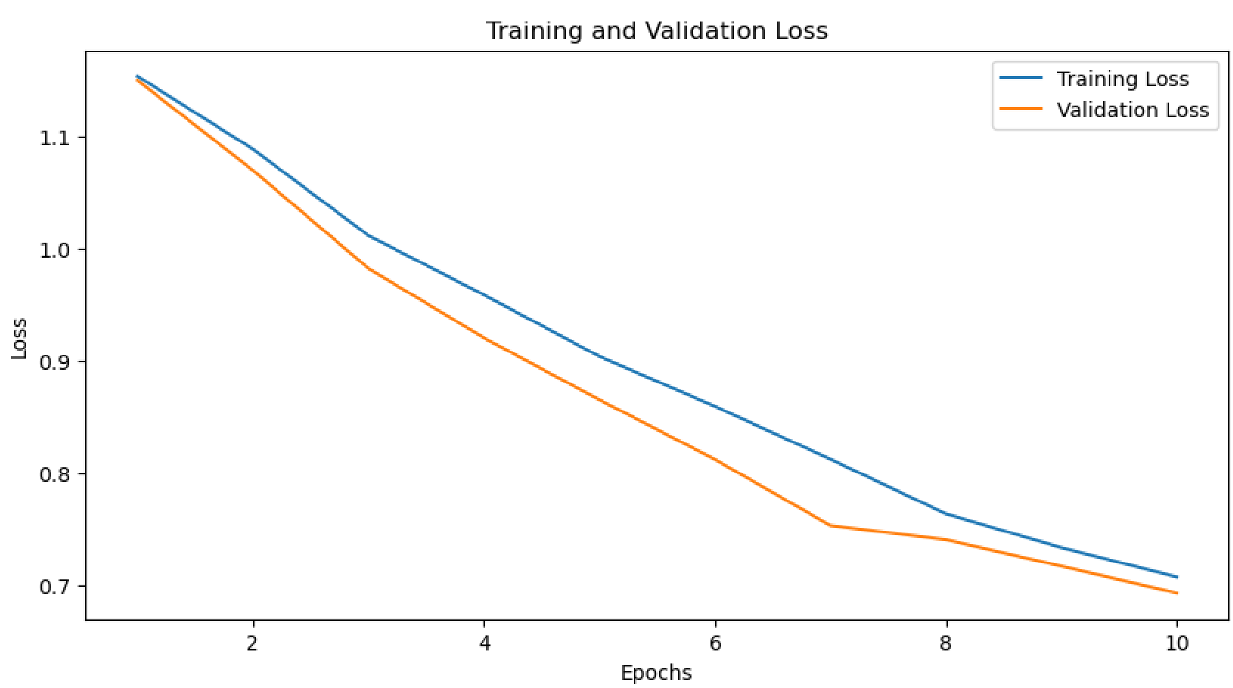

Training and Validation Loss
Training Loss
Validation Loss
Loss
Epochs
0.2
0.4
0.6
0.8
2
4
6
8
10

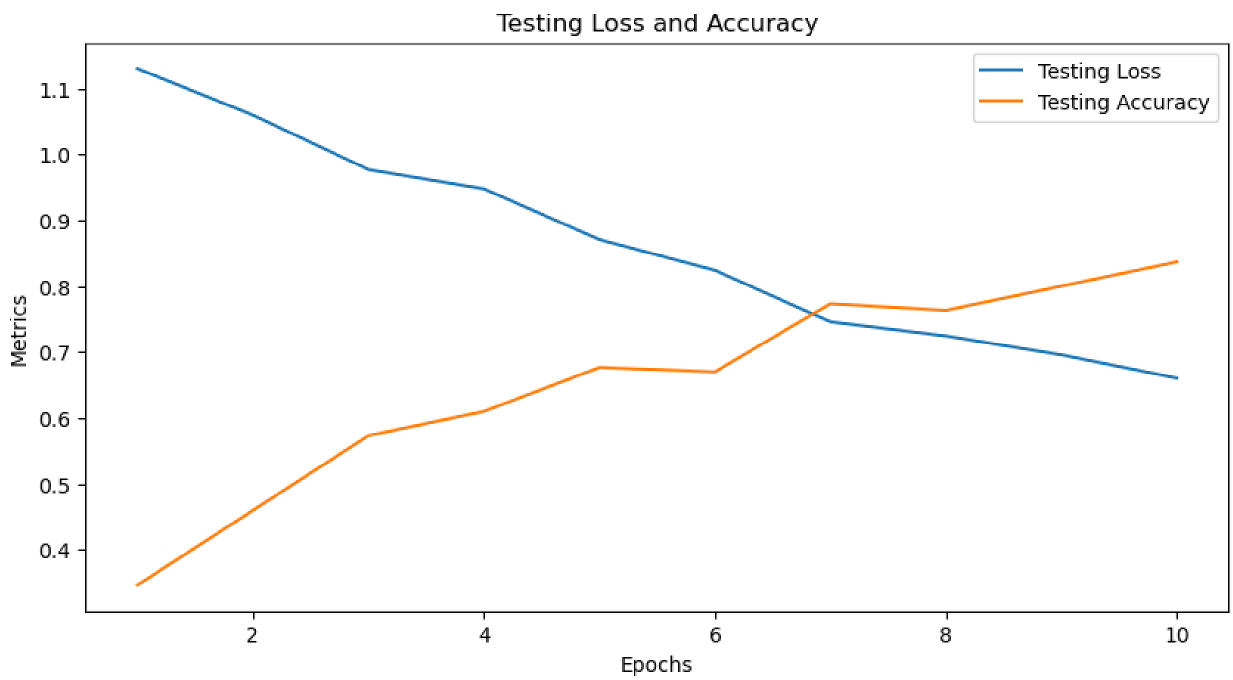

Testing Loss and Accuracy
Testing Loss
Testing Accuracy
Metrics
Epochs
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
2
4
6
8
10

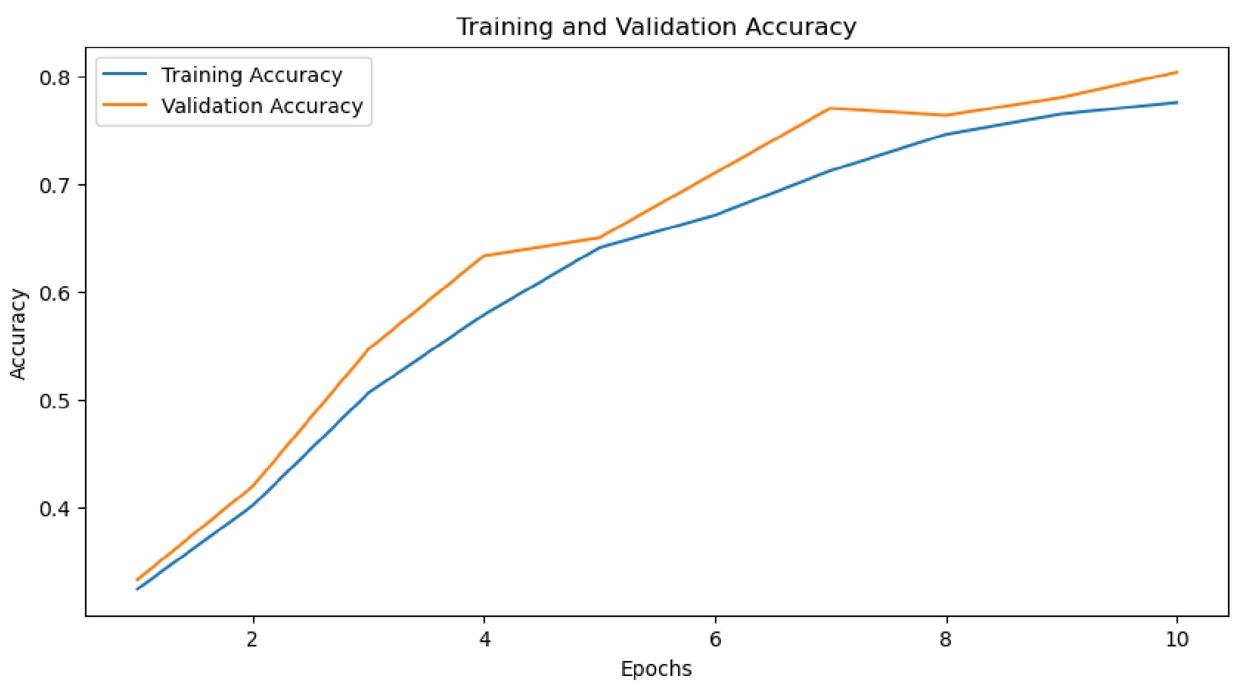


Learning rate : 0.00001
Batch Size : 64
Epochs : 10

```
Epoch 10/10, Training Loss: 0.7074, Training Accuracy: 77.54%, Validation Loss: 0.6930, Validation Accuracy: 80.33%, Testing Loss: 0.6605, Testing Accuracy: 83.67%
```



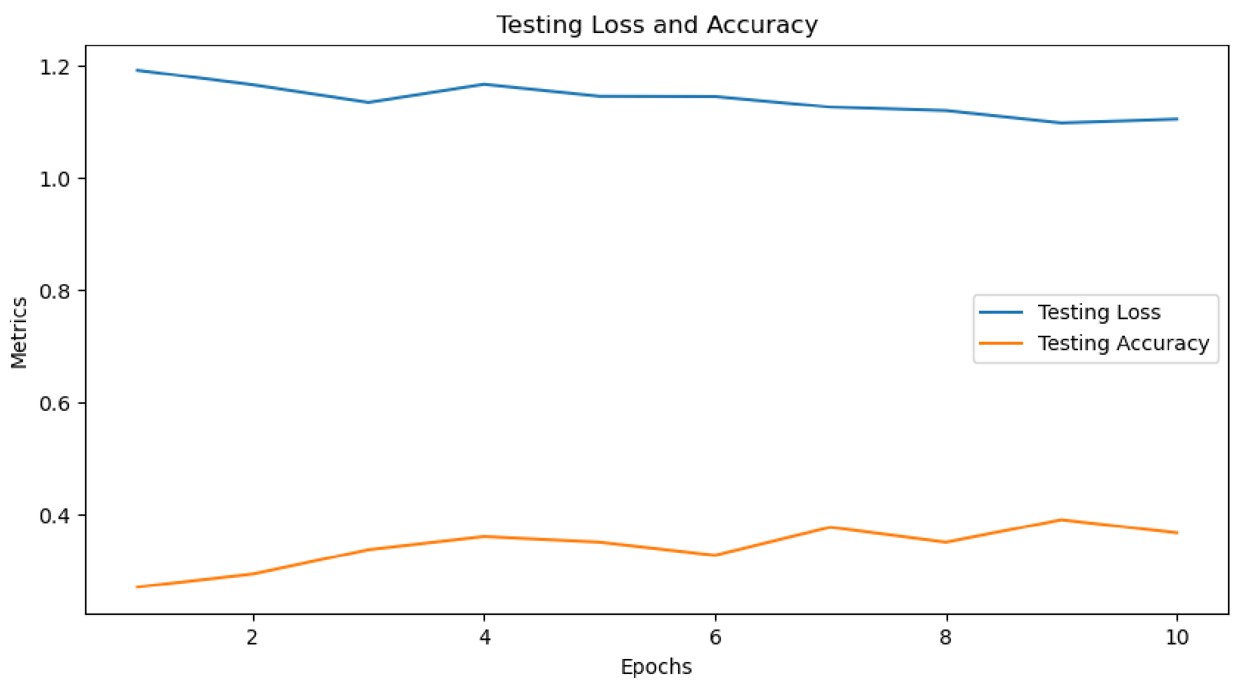


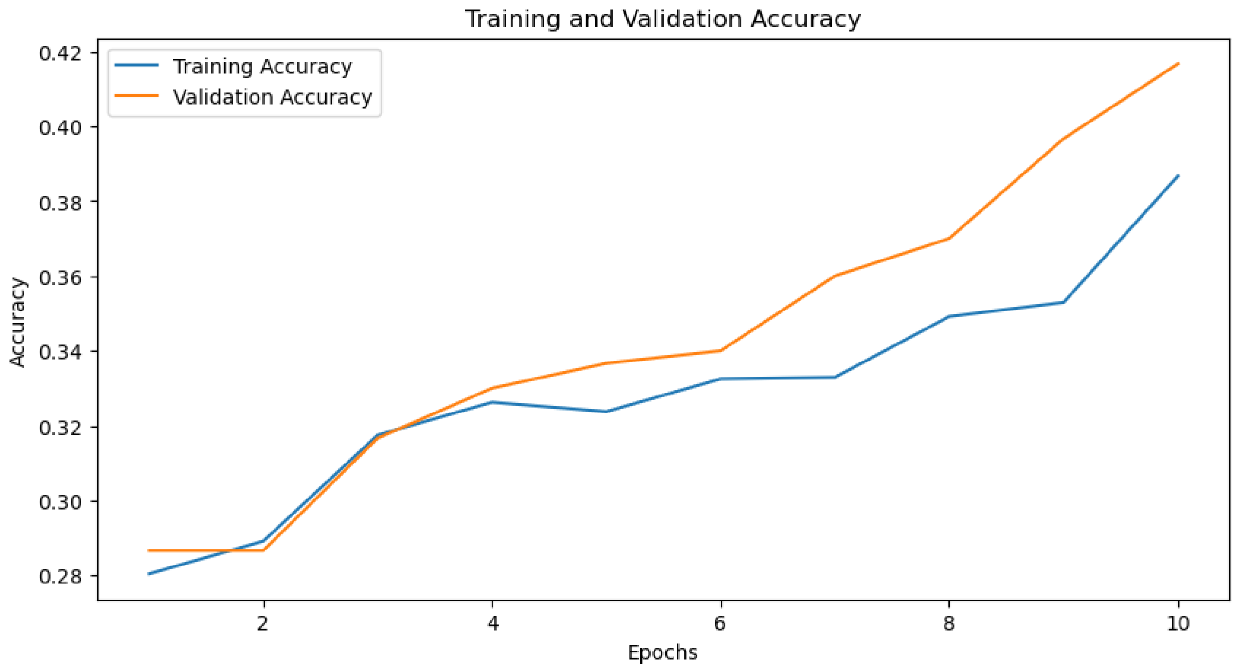


Learning rate : 0.000001
Batch Size : 64
Epochs : 10

```
Epoch 10/10, Training Loss: 1.0987, Training Accuracy: 38.67%, Validation Loss: 1.0714, Validation Accuracy: 41.67%, Testing Loss: 1.1050, Testing Accuracy: 36.67%
```

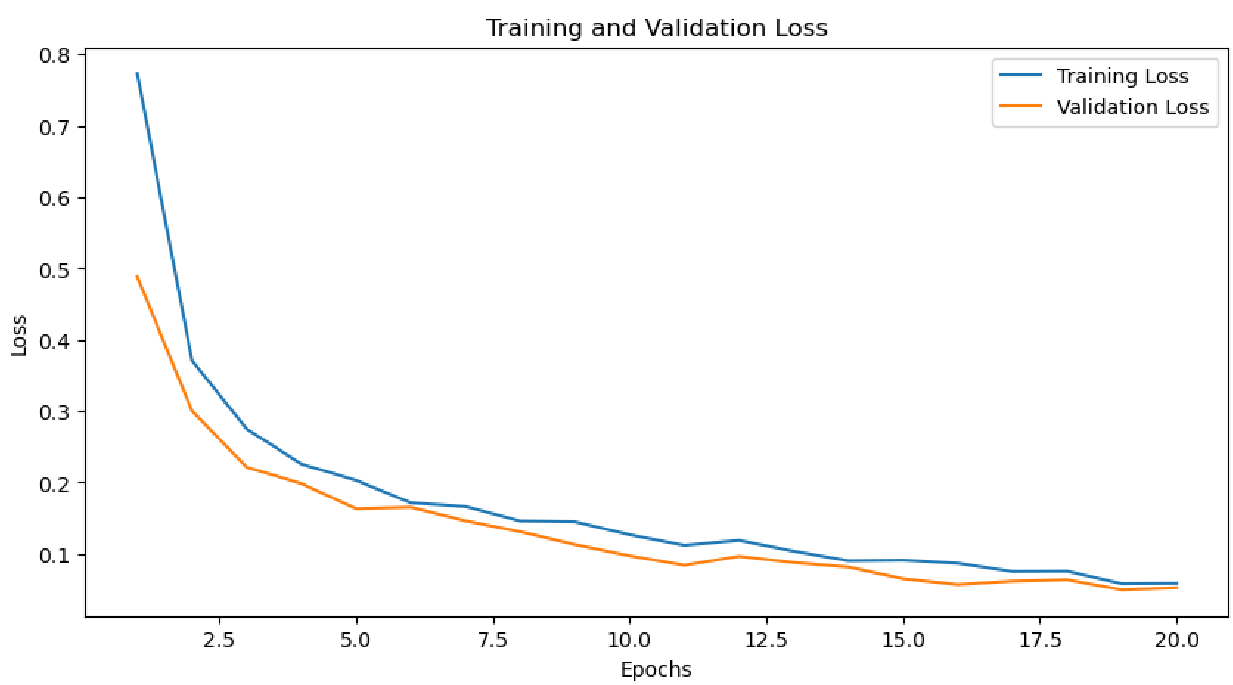

Training and Validation Loss
Training Loss
Validation Loss
Loss
Epochs
1.20
1.18
1.16
1.14
1.12
1.10
1.08
2
4
6
8
10

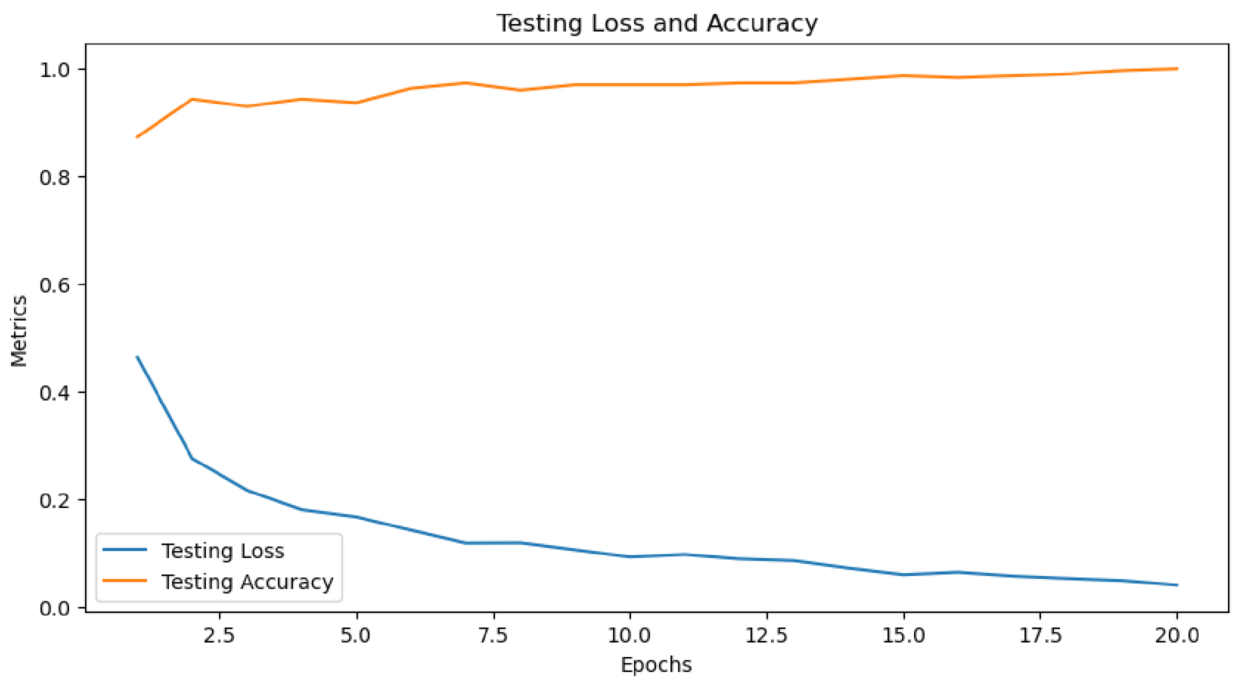

Testing Loss and Accuracy
Testing Loss
Testing Accuracy
Metrics
Epochs
1.2
1.0
0.8
0.6
0.4
2
4
6
8
10

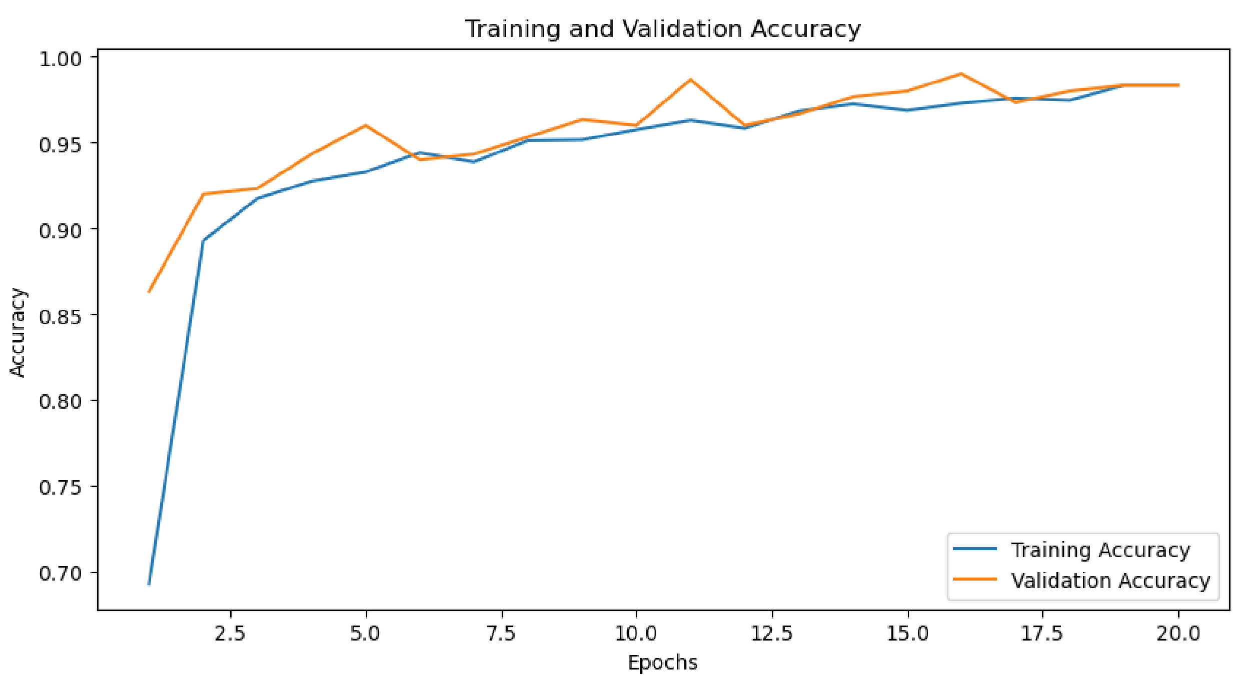


Learning rate : 0.0001
Batch Size : 32
Epochs : 20

```
Epoch 20/20, Training Loss: 0.0588, Training Accuracy: 98.33%, Validation Loss:
0.0525, Validation Accuracy: 98.33%, Testing Loss: 0.0397, Testing Accuracy:
100.00%
```

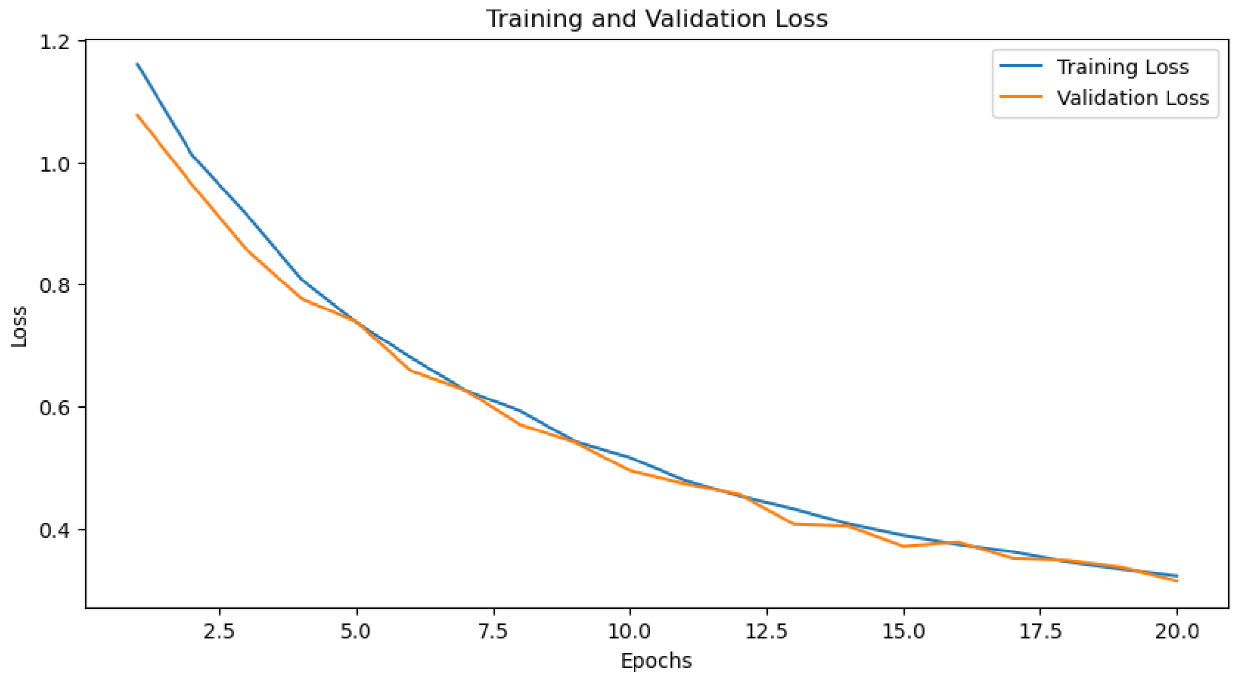


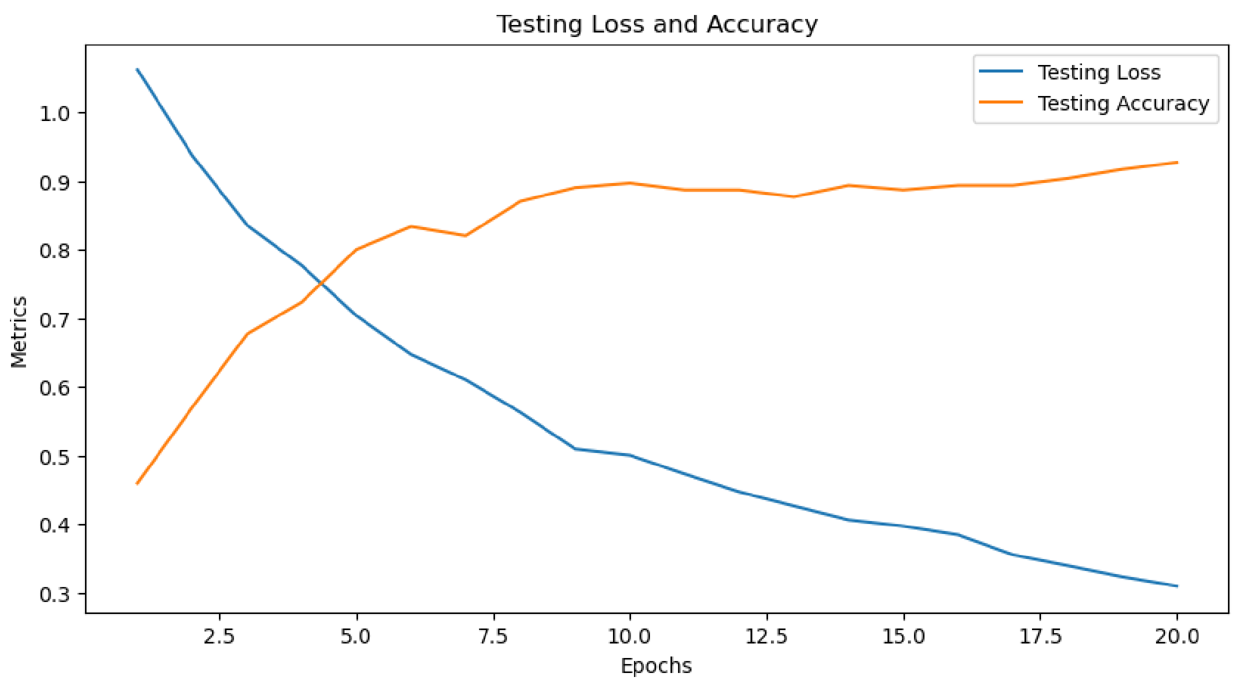


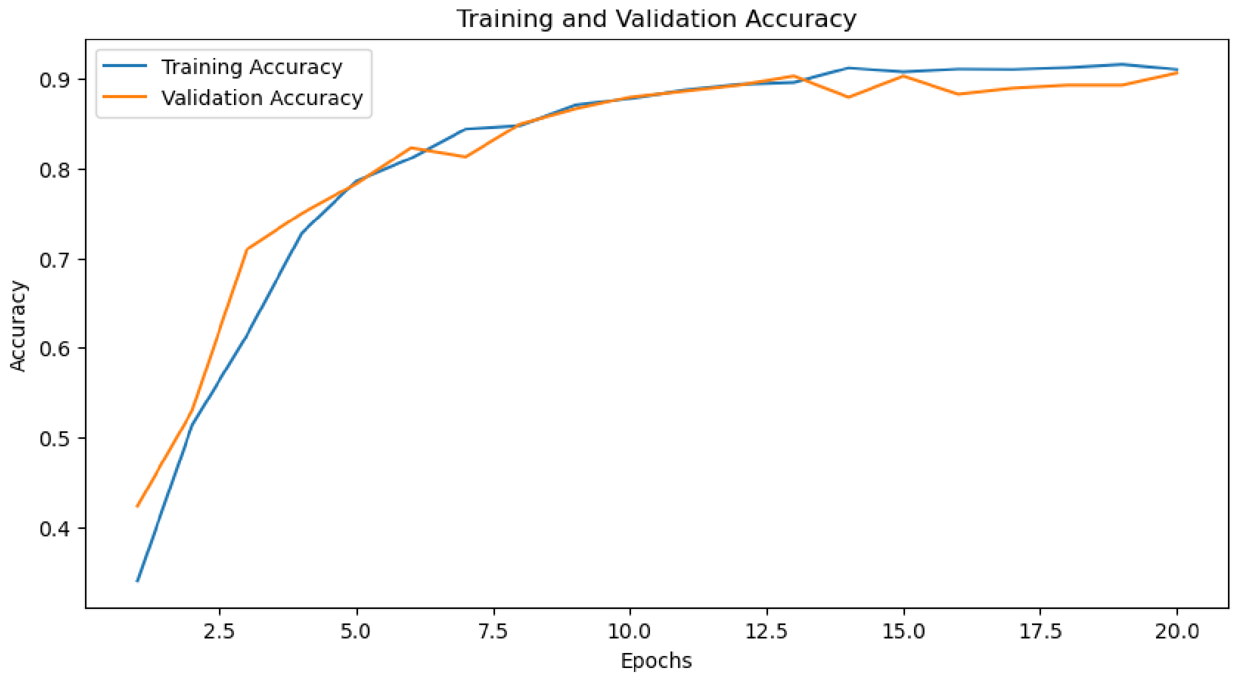


Learning rate : 0.00001
Batch Size : 32
Epochs : 20

```
Epoch 20/20, Training Loss: 0.3230, Training Accuracy: 91.08%, Validation Loss:
0.3144, Validation Accuracy: 90.67%, Testing Loss: 0.3102, Testing Accuracy:
92.67%
```

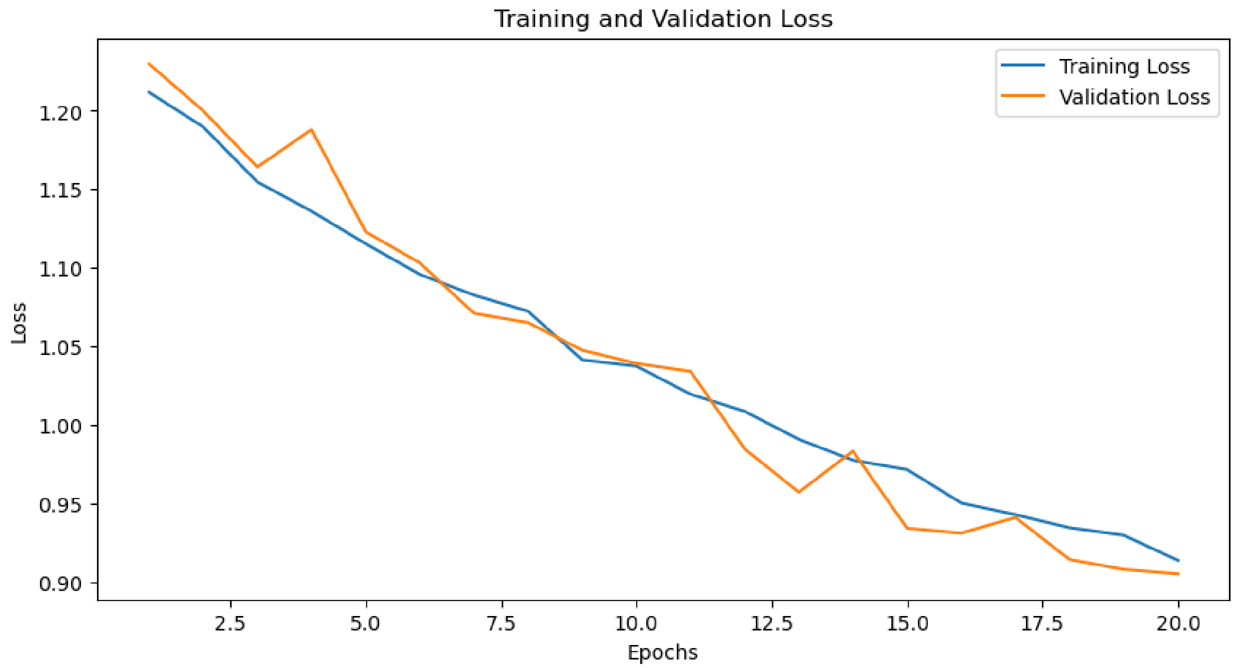

Training and Validation Loss
Training Loss
Validation Loss
Loss
Epochs
1.2
1.0
0.8
0.6
0.4
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

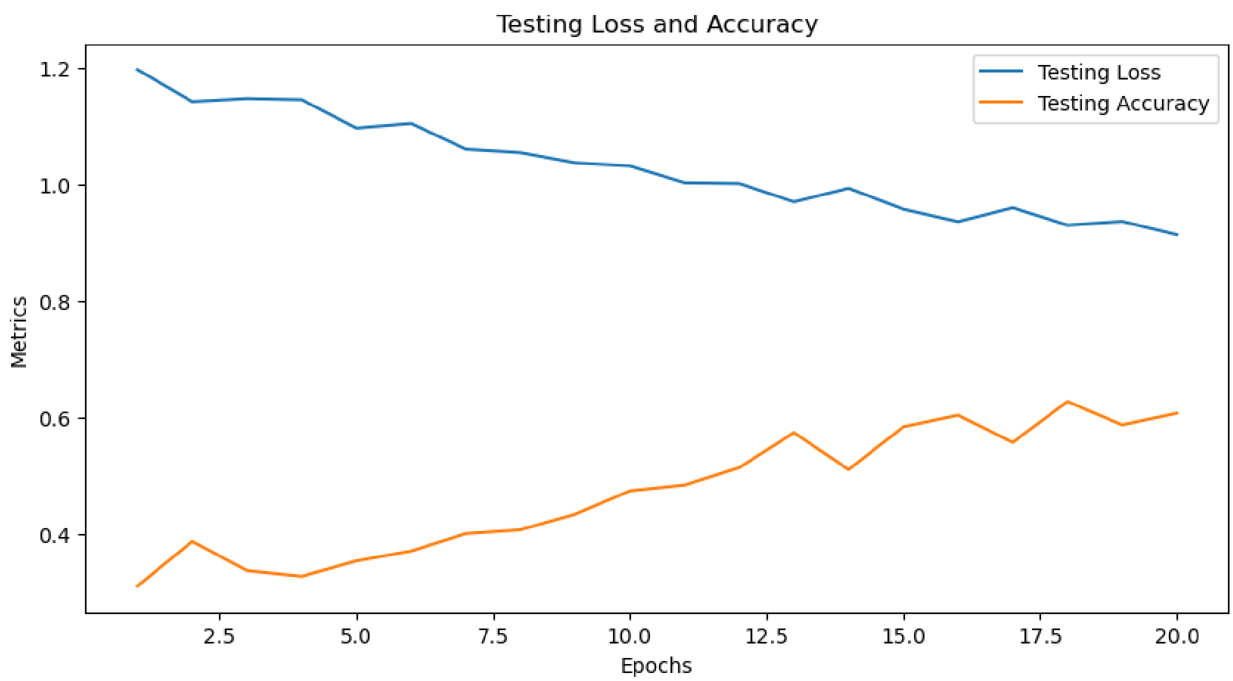

Testing Loss and Accuracy
Testing Loss
Testing Accuracy
Metrics
Epochs
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

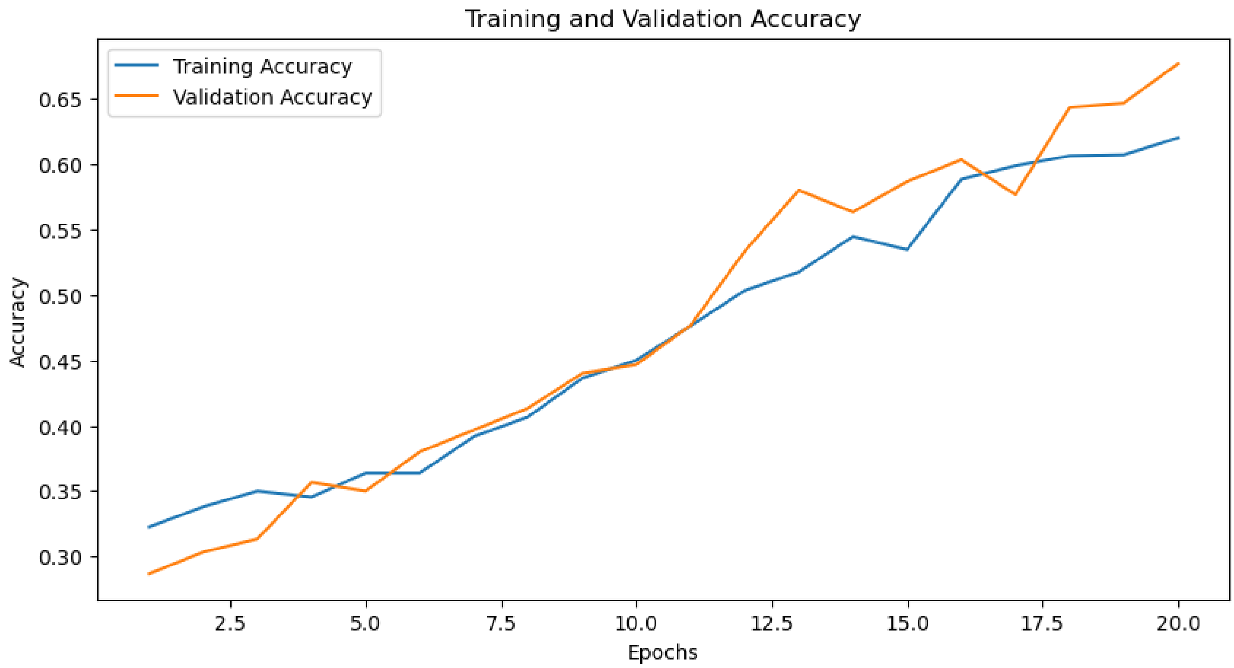


Learning rate : 0.000001
Batch Size : 32
Epochs : 20

```
Epoch 20/20, Training Loss: 0.9138, Training Accuracy: 62.00%, Validation Loss:
0.9052, Validation Accuracy: 67.67%, Testing Loss: 0.9132, Testing Accuracy:
60.67%
```

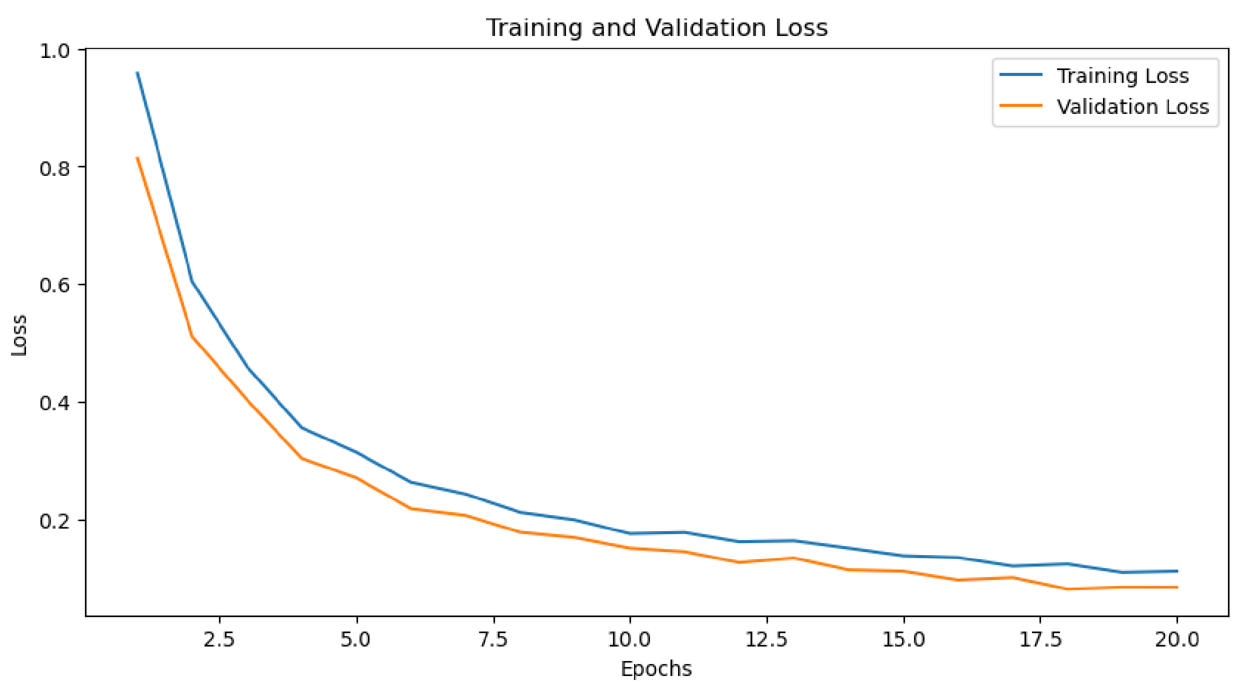

Training and Validation Loss
Training Loss
Validation Loss
1.20
1.15
1.10
1.05
1.00
0.95
0.90
Loss
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epochs

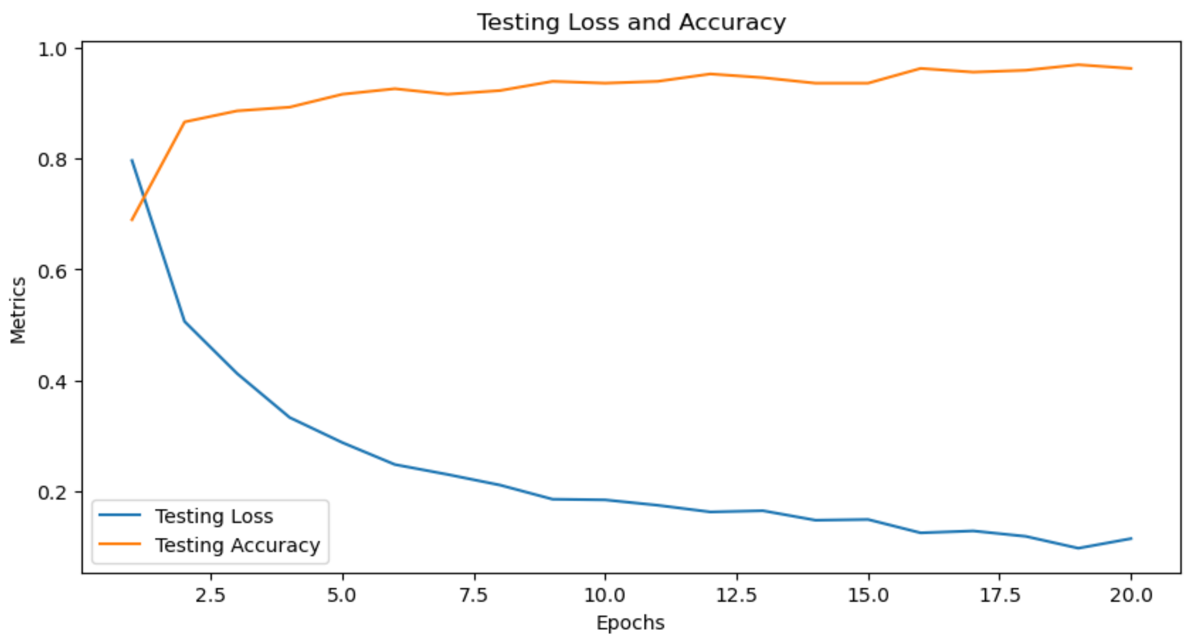

Testing Loss and Accuracy
Testing Loss
Testing Accuracy
1.2
1.0
0.8
0.6
0.4
Metrics
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epochs

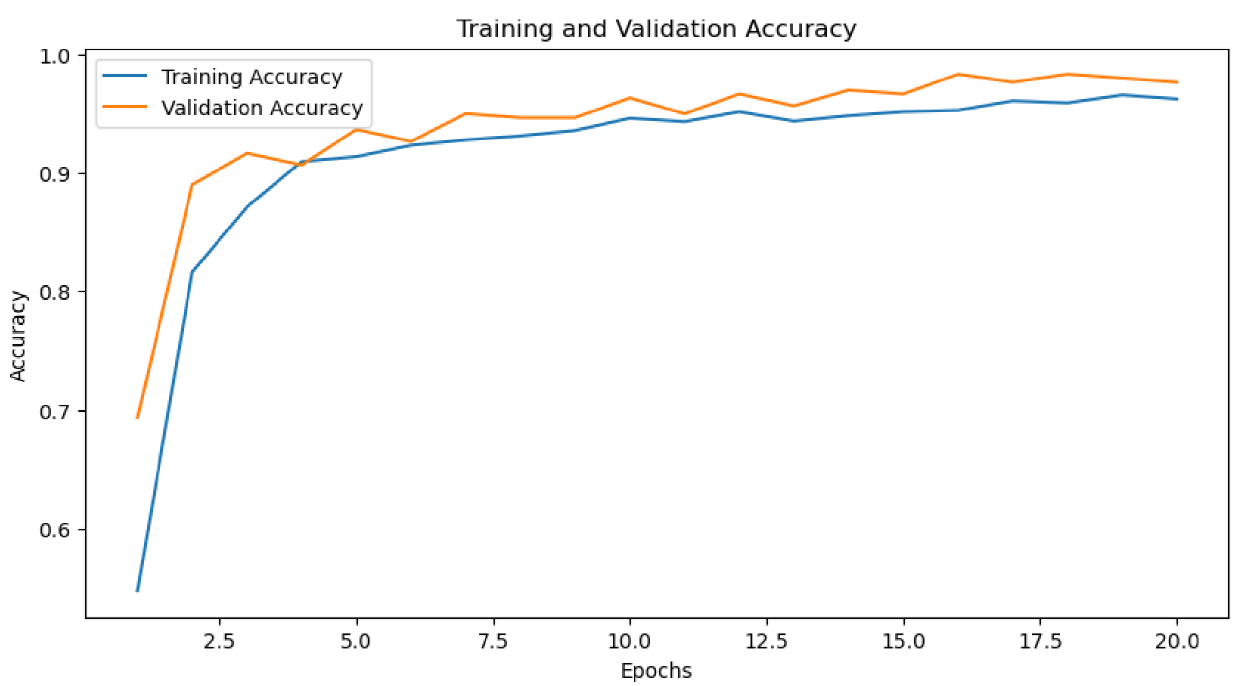


Learning rate : 0.0001
Batch Size : 64
Epochs : 20

```
Epoch 20/20, Training Loss: 0.1127, Training Accuracy: 96.25%, Validation Loss:
0.0855, Validation Accuracy: 97.67%, Testing Loss: 0.1138, Testing Accuracy:
96.33%
```

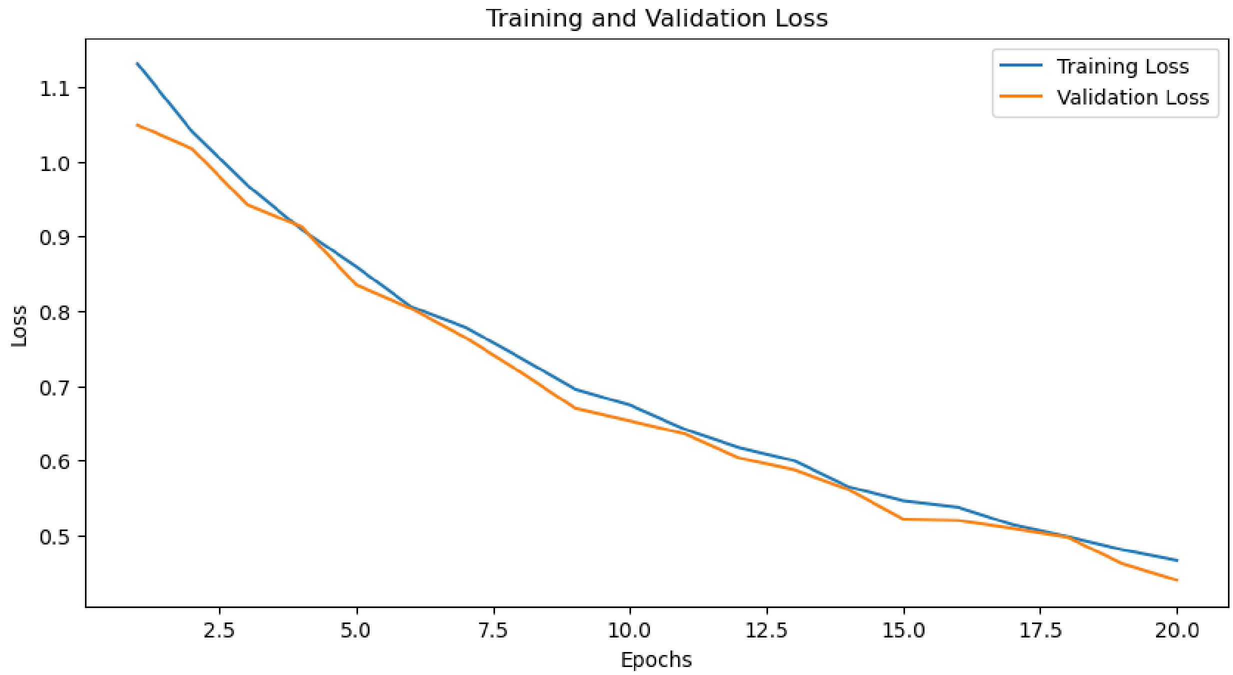


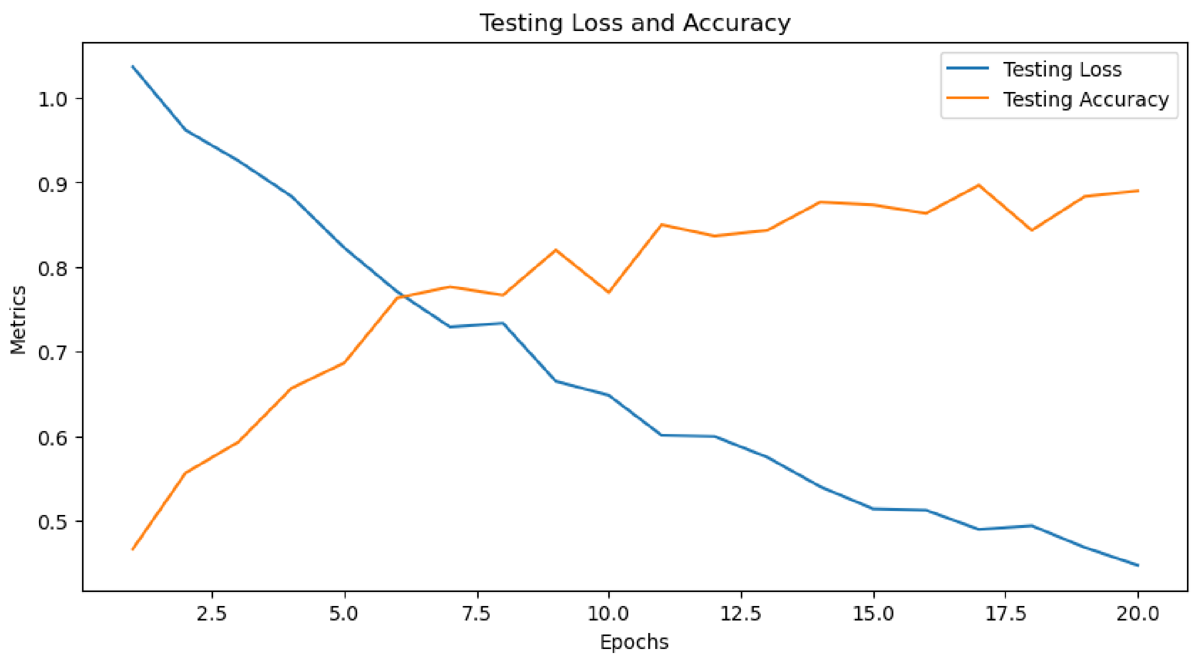


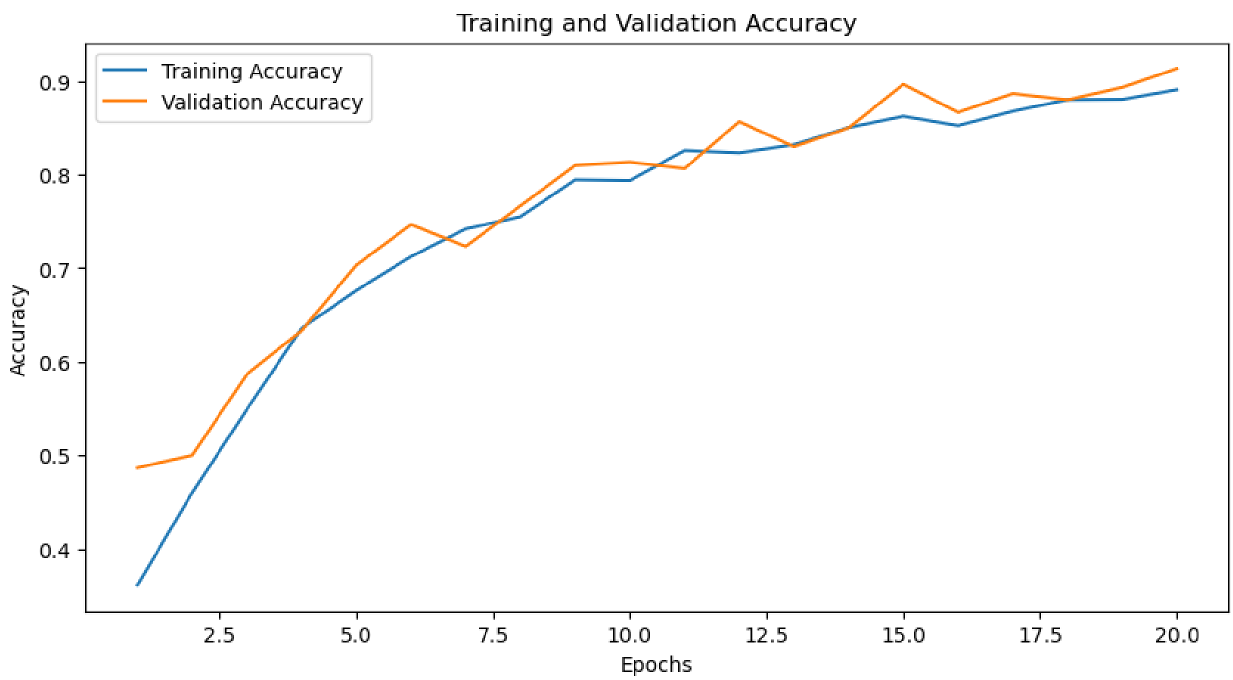


Learning rate : 0.00001
Batch Size : 64
Epochs : 20

```
Epoch 20/20, Training Loss: 0.4668, Training Accuracy: 89.08%, Validation Loss: 0.4405, Validation Accuracy: 91.33%, Testing Loss: 0.4476, Testing Accuracy: 89.00%
```

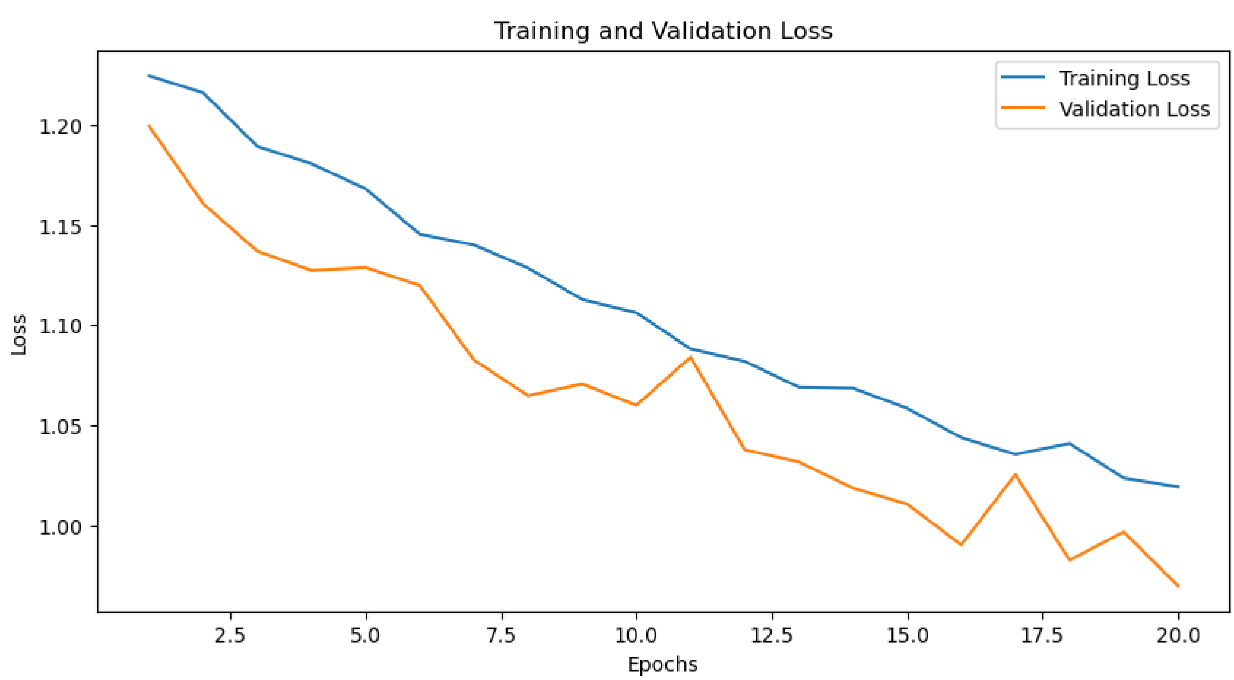

Training and Validation Loss
Training Loss
Validation Loss
Loss
Epochs
1.1
1.0
0.9
0.8
0.7
0.6
0.5
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

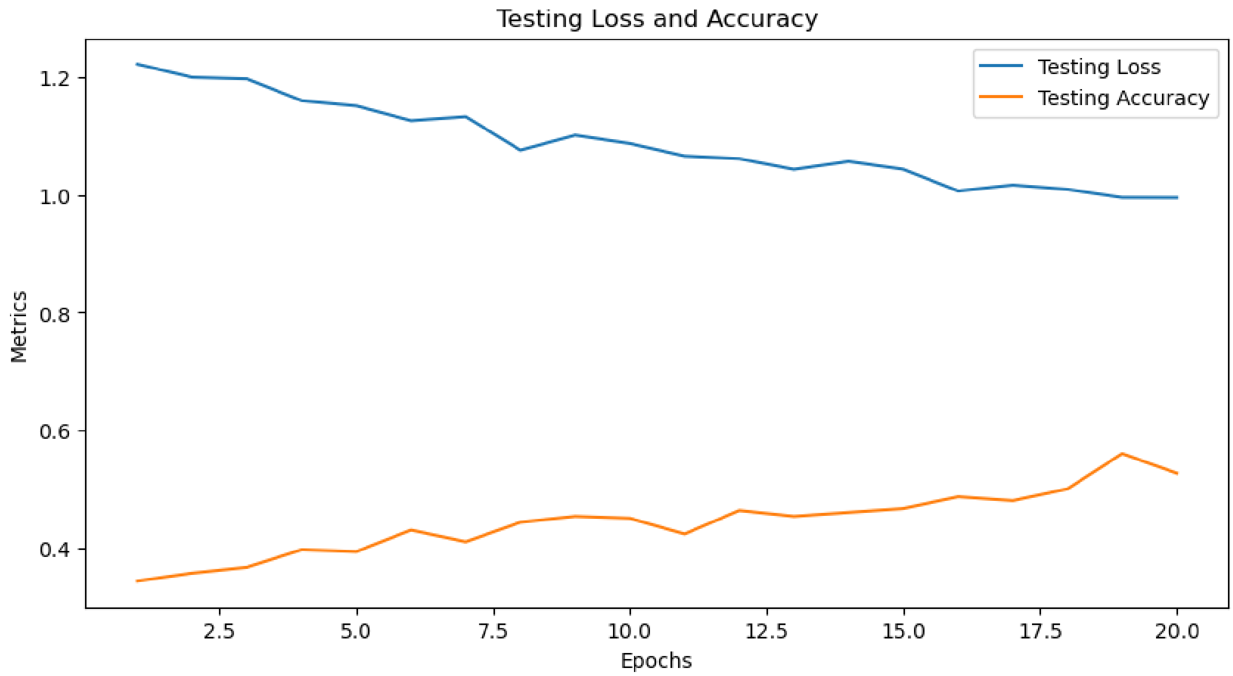

Testing Loss and Accuracy
Testing Loss
Testing Accuracy
Metrics
Epochs
1.0
0.9
0.8
0.7
0.6
0.5
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

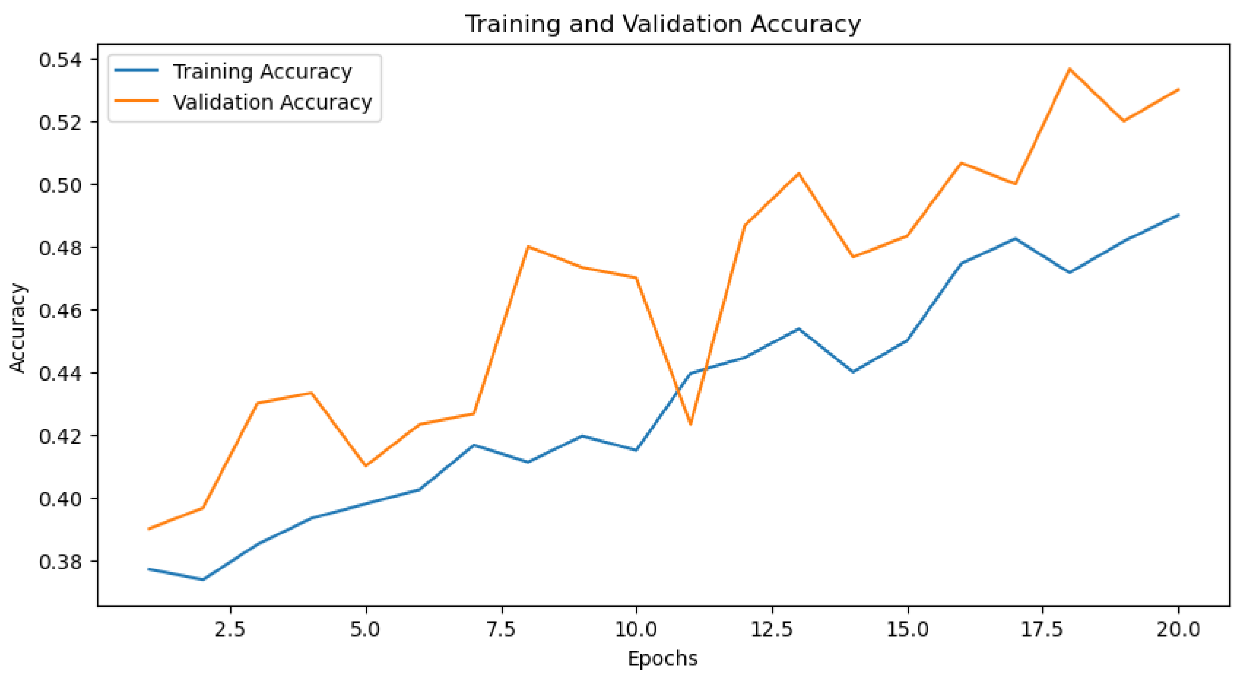


Learning rate : 0.000001
Batch Size : 64
Epochs : 20

```
Epoch 20/20, Training Loss: 1.0193, Training Accuracy: 49.00%, Validation Loss:
0.9699, Validation Accuracy: 53.00%, Testing Loss: 0.9956, Testing Accuracy:
52.67%
```







a)

Extraction of Zip file

```
import os
import patoolib  # Ensure you have patoolib installed: pip install patool
from shutil import move
```

```python
import random

# Set the paths
rar_file_path = 'C:\\Users\\Nissanth
NA\\Downloads\\10973_2021_10903_MOESM2_ESM.rar'
extracted_folder = 'C:\\Users\\Nissanth NA\\Downloads\\Path1'  # Change this to
your desired extraction path

# Create a temporary folder for extraction
temp_folder = 'C:\\Users\\Nissanth NA\\Downloads\\temp_extraction'
os.makedirs(temp_folder, exist_ok=True)

# Extract the RAR file to the temporary folder
patoolib.extract_archive(rar_file_path, outdir=temp_folder)

# Move files from the subfolder to the target extraction folder based on
chemical name
source_subfolder = os.path.join(temp_folder, 'S1_Raw_Photographs_Full_Study')
target_folder = os.path.join(extracted_folder, 'train')  # Change this to your
desired target path
os.makedirs(target_folder, exist_ok=True)

# Define the subfolders (classes)
subfolders = ['Ethanol', 'Pentane', 'Propanol']

# Define the ratio for train, validate, test
train_ratio = 0.7
validate_ratio = 0.2
test_ratio = 0.1

for subfolder in subfolders:
    subfolder_path = os.path.join(extracted_folder, 'train', subfolder)
    os.makedirs(subfolder_path, exist_ok=True)
    subfolder_path = os.path.join(extracted_folder, 'validate', subfolder)
    os.makedirs(subfolder_path, exist_ok=True)
    subfolder_path = os.path.join(extracted_folder, 'test', subfolder)
    os.makedirs(subfolder_path, exist_ok=True)

# List all images in the source subfolder
images = os.listdir(source_subfolder)

# Shuffle the list of images randomly
random.shuffle(images)
```

```python
# Move images to train, validate, test folders
for filename in images:
    chemical_name = filename.split('_')[0]  # Extract the chemical name
    random_num = random.random()

    if random_num < train_ratio:
        folder_type = 'train'
    elif random_num < train_ratio + validate_ratio:
        folder_type = 'validate'
    else:
        folder_type = 'test'

    target_folder = os.path.join(extracted_folder, folder_type, chemical_name)
    move(os.path.join(source_subfolder, filename), os.path.join(target_folder,
filename))

# Remove the temporary folder
os.rmdir(source_subfolder)  # Remove the empty subfolder
os.rmdir(temp_folder)  # Remove the temporary folder

print("Extraction and organization completed successfully.")
```

```python
from torch.utils.data import random_split, DataLoader
from torchvision import transforms
from torchvision.datasets import ImageFolder

# Specify the actual path to your dataset
train_path = r"C:\Users\Nissanth NA\Downloads\Path1\test"
valid_path = r"C:\Users\Nissanth NA\Downloads\Path1\validate"
test_path = r"C:\Users\Nissanth NA\Downloads\Path1\test"

# Common transformations applied to all sets
val_transform = transforms.Compose([
    transforms.Resize(size=(224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Define the transformation for the training set
train_transform = transforms.Compose([
    transforms.Resize(size=(224, 224)),
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),
```

```
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.RandomRotation(10),
    transforms.ColorJitter(brightness=0.3, contrast=0.3),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

BATCH_SIZE = 32

# Create the dataset using ImageFolder and apply the common transformations
train_dataset = ImageFolder(train_path, transform=train_transform)
valid_dataset = ImageFolder(valid_path, transform=val_transform)
test_dataset = ImageFolder(test_path, transform=val_transform)

# Create DataLoader instances for each set with the custom collate function
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=BATCH_SIZE, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

```
import torch
import torch.nn as nn
from torchvision.models import resnet34
from torchsummary import summary

# Define the number of classes for your task
num_classes = 3

# Load the pretrained ResNet-34 model
model = resnet34(pretrained=True)

# Modify the final classification layer to match the number of classes
model.fc = nn.Linear(model.fc.in_features, num_classes)

for param in model.parameters():
    param.requires_grad = False
for param in model.fc.parameters():
    param.requires_grad = True

# Print the model summary
print(summary(model, (3, 224, 224))) # Assuming input images are RGB with size
224x224
```

```
c:\Users\Nissanth
NA\anaconda3\Lib\site-packages\torchvision\models\_utils.py:208: UserWarning:
The parameter 'pretrained' is deprecated since 0.13 and may be removed in the
future, please use 'weights' instead.
  warnings.warn(
c:\Users\Nissanth
NA\anaconda3\Lib\site-packages\torchvision\models\_utils.py:223: UserWarning:
Arguments other than a weight enum or `None` for 'weights' are deprecated since
0.13 and may be removed in the future. The current behavior is equivalent to
passing `weights=ResNet34_Weights.IMAGENET1K_V1`. You can also use
`weights=ResNet34_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 112, 112]           9,408
       BatchNorm2d-2         [-1, 64, 112, 112]             128
              ReLU-3         [-1, 64, 112, 112]               0
         MaxPool2d-4           [-1, 64, 56, 56]               0
            Conv2d-5           [-1, 64, 56, 56]          36,864
       BatchNorm2d-6           [-1, 64, 56, 56]             128
              ReLU-7           [-1, 64, 56, 56]               0
            Conv2d-8           [-1, 64, 56, 56]          36,864
       BatchNorm2d-9           [-1, 64, 56, 56]             128
             ReLU-10           [-1, 64, 56, 56]               0
       BasicBlock-11           [-1, 64, 56, 56]               0
           Conv2d-12           [-1, 64, 56, 56]          36,864
      BatchNorm2d-13           [-1, 64, 56, 56]             128
             ReLU-14           [-1, 64, 56, 56]               0
           Conv2d-15           [-1, 64, 56, 56]          36,864
      BatchNorm2d-16           [-1, 64, 56, 56]             128
             ReLU-17           [-1, 64, 56, 56]               0
       BasicBlock-18           [-1, 64, 56, 56]               0
           Conv2d-19           [-1, 64, 56, 56]          36,864
      BatchNorm2d-20           [-1, 64, 56, 56]             128
             ReLU-21           [-1, 64, 56, 56]               0
           Conv2d-22           [-1, 64, 56, 56]          36,864
...
Params size (MB): 81.20
Estimated Total Size (MB): 178.06
----------------------------------------------------------------
```

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import random_split, DataLoader
from torchvision import transforms, utils
```

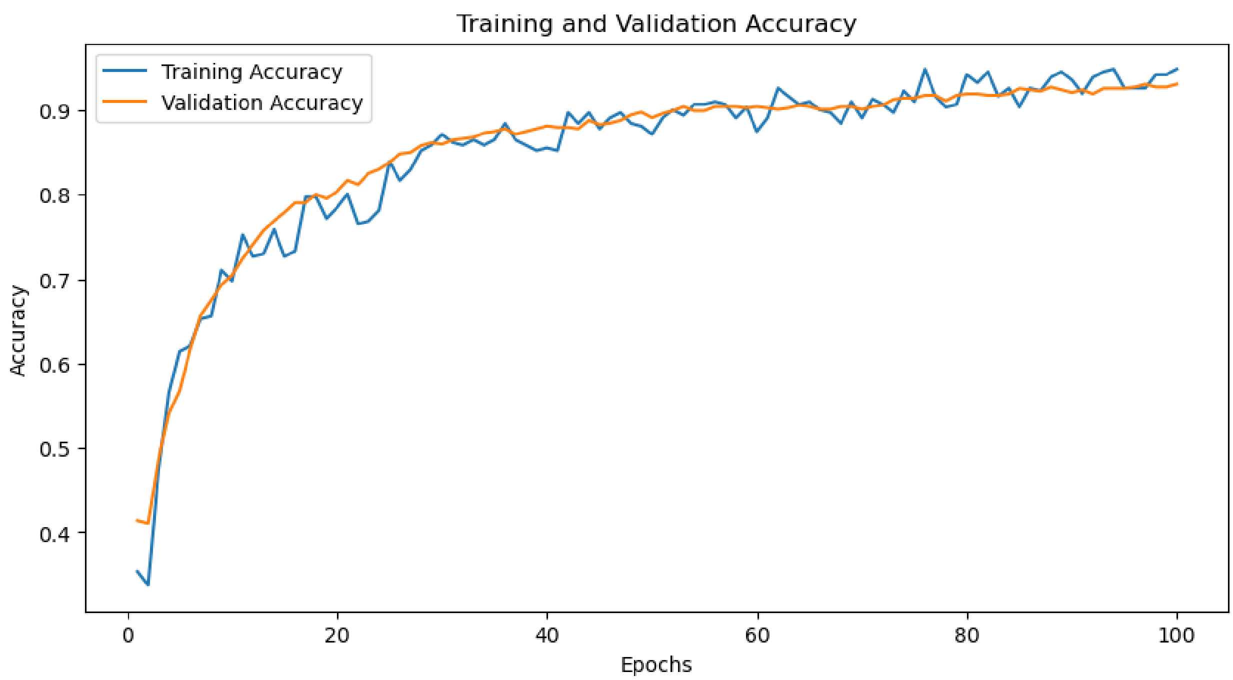
```python
from torchvision.datasets import ImageFolder
from torchvision.models import resnet34
import matplotlib.pyplot as plt
from sklearn.metrics import precision_recall_curve, auc
from sklearn.preprocessing import label_binarize
from PIL import Image
import torchvision.transforms.functional as F
import numpy as np
from itertools import cycle
from sklearn.metrics import confusion_matrix

# Define the loss function and optimizer with L2 regularization
criterion = nn.CrossEntropyLoss()
weight_decay = 1e-5  # You can experiment with different values
optimizer = optim.SGD(model.fc.parameters(), lr=0.0001, momentum=0.9,
weight_decay=weight_decay)

# Learning rate scheduler
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, patience=3,
factor=0.1, verbose=True)

# Early stopping
best_val_loss = float('inf')
patience = 5
counter = 0

# Lists to store training, validation, and testing metrics for each epoch
train_losses, train_accuracies = [], []
val_losses, val_accuracies = [], []

# Training loop
num_epochs = 100
for epoch in range(num_epochs):
    model.train()
    epoch_train_loss = 0.0
    correct_train = 0
    total_train = 0

    for data, targets in train_loader:
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
```

```
    epoch_train_loss += loss.item()
    _, predicted = torch.max(outputs.data, 1)
    #total_train += labels.len
    correct_train += (predicted == targets).sum().item()

train_loss = epoch_train_loss / len(train_loader)
train_accuracy = (correct_train / len(train_loader.dataset))

# Validate the model after each epoch
model.eval()
epoch_val_loss = 0.0
correct_val = 0
total_val = 0

with torch.no_grad():
    for inputs, labels in valid_loader:
        outputs = model(inputs)
        val_loss = criterion(outputs, labels)
        epoch_val_loss += val_loss.item()

        _, predicted = torch.max(outputs.data, 1)
        #total_val += labels.size(0)
        correct_val += (predicted == labels).sum().item()

val_loss = epoch_val_loss / len(valid_loader)
val_accuracy = (correct_val / len(valid_loader.dataset))

# Append metrics to lists
train_losses.append(train_loss)
train_accuracies.append(train_accuracy)
val_losses.append(val_loss)
val_accuracies.append(val_accuracy)


# Learning rate scheduler step
scheduler.step(val_loss)

# Early stopping check
if val_loss < best_val_loss:
    best_val_loss = val_loss
    counter = 0
else:
    counter += 1
```

```
        if counter >= patience:
            break

    # Print metrics for each epoch
    print(f'Epoch {epoch+1}/{num_epochs}, '
          f'Training Loss: {train_loss:.4f}, Training Accuracy: {train_accuracy
* 100:.2f}%, '
          f'Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_accuracy
* 100:.2f}%, ')

    # Unfreeze layers progressively
    if epoch == 19:
        # Unfreeze the fourth layer
        for param in model.layer4.parameters():
            param.requires_grad = True
        # Adjust learning rate
        optimizer = optim.SGD([
            {'params': model.fc.parameters()},
            {'params': model.layer4.parameters(), 'lr': 0.0001}
        ], lr=0.0001, momentum=0.9, weight_decay=weight_decay)

    elif epoch == 39:
        # Unfreeze the third layer
        for param in model.layer3.parameters():
            param.requires_grad = True
        # Adjust learning rate
        optimizer = optim.SGD([
            {'params': model.fc.parameters()},
            {'params': model.layer4.parameters()},
            {'params': model.layer3.parameters(), 'lr': 0.0001}
        ], lr=0.0001, momentum=0.9, weight_decay=weight_decay)

    elif epoch == 59:
        # Unfreeze the second layer
        for param in model.layer2.parameters():
            param.requires_grad = True
        # Adjust learning rate
        optimizer = optim.SGD([
            {'params': model.fc.parameters()},
            {'params': model.layer4.parameters()},
            {'params': model.layer3.parameters()},
            {'params': model.layer2.parameters(), 'lr': 0.0001}
        ], lr=0.0001, momentum=0.9, weight_decay=weight_decay)
```

```python
    elif epoch == 79:
        # Unfreeze the first layer
        for param in model.layer1.parameters():
            param.requires_grad = True
        # Adjust learning rate
        optimizer = optim.SGD([
            {'params': model.fc.parameters()},
            {'params': model.layer4.parameters()},
            {'params': model.layer3.parameters()},
            {'params': model.layer2.parameters()},
            {'params': model.layer1.parameters(), 'lr': 0.0001}
        ], lr=0.0001, momentum=0.9, weight_decay=weight_decay)

# Plotting graphs
epochs = list(range(1, len(train_losses) + 1))

# Graph 1: Training and Validation Loss
plt.figure(figsize=(10, 5))
plt.plot(epochs, train_losses, label='Training Loss')
plt.plot(epochs, val_losses, label='Validation Loss')
vertical_lines = [19, 39, 59, 79]
for line in vertical_lines:
    plt.axvline(x=line, color='red')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.show()

# Graph 2: Training and Validation Accuracy
plt.figure(figsize=(10, 5))
plt.plot(epochs, train_accuracies, label='Training Accuracy')
plt.plot(epochs, val_accuracies, label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()
plt.show()
```

```
Epoch 100/100, Training Loss: 0.1724, Training Accuracy: 94.86%, Validation
Loss: 0.1814, Validation Accuracy: 93.07%,
```

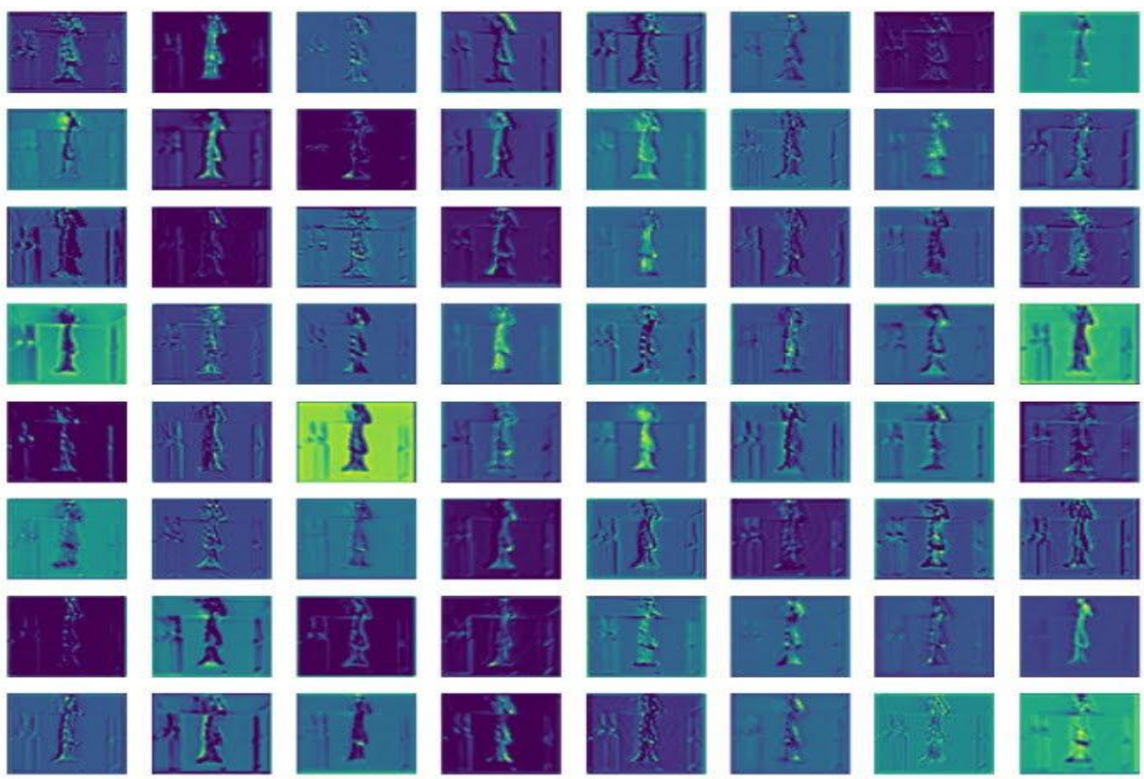



```
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt
```

```
from sklearn.metrics import confusion_matrix, precision_recall_curve,
average_precision_score
from sklearn.preprocessing import label_binarize
from sklearn.utils import resample

# Lists to store testing metrics
test_losses, test_accuracies = [], []

# Testing loop with feature map visualization
model.eval()
epoch_test_loss = 0.0
correct_test = 0
total_test = 0
all_labels = []
all_predictions = []
all_probabilities = []

# Define a hook to capture feature maps
def visualize_hook(module, input, output):
    plt.figure(figsize=(15, 15))
    for i in range(output.size(1)):
        plt.subplot(8, 8, i + 1)
        plt.imshow(output[0, i].detach().cpu().numpy(), cmap="gray")
        plt.axis("off")
    plt.show()

# Choose a specific layer to visualize
layer_to_visualize = model.layer1[0].conv1  # Change this to the desired layer
in your model
hook = layer_to_visualize.register_forward_hook(visualize_hook)

image = r"C:\Users\Nissanth
NA\Downloads\Path1\test\Pentane\Pentane_Full_0474.JPG"
im = Image.open(image)
x = val_transform(im).unsqueeze(0)
_ = model(x)

with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        probabilities = F.softmax(outputs, dim=1)  # Apply softmax
        all_probabilities.append(probabilities.numpy())
        test_loss = criterion(outputs, labels)
        epoch_test_loss += test_loss.item()
```

```
        _, predicted = torch.max(probabilities, 1)
        total_test += labels.size(0)
        correct_test += (predicted == labels).sum().item()

        all_labels.append(labels.numpy())
        all_predictions.append(predicted.numpy())

# Calculate and print test metrics
test_loss = epoch_test_loss / len(test_loader)
test_accuracy = correct_test / total_test
print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy * 100:.2f}%")

# Remove the hook
hook.remove()

num_classes = 3

# Flatten the lists
all_labels = [item for sublist in all_labels for item in sublist]
all_predictions = [item for sublist in all_predictions for item in sublist]

# Generate confusion matrix
cm = confusion_matrix(all_labels, all_predictions)
print("Confusion Matrix:")
print(cm)

# Binarize the labels for precision-recall curve
binarized_labels = label_binarize(all_labels, classes=list(range(num_classes)))

precision = dict()
recall = dict()
average_precision = dict()
for i in range(num_classes):
    binarized_labels_class_i = binarized_labels[:, i]
    predicted_probabilities_class_i = np.array(all_probabilities[i])[:, i]
    print(f"Class {i}: Ground Truth Shape = {binarized_labels_class_i.shape},
Predicted Shape = {predicted_probabilities_class_i.shape}")

    # Resample predicted probabilities to match the length of the ground truth
    predicted_probabilities_class_i_resampled =
resample(predicted_probabilities_class_i,
n_samples=len(binarized_labels_class_i), random_state=42)
```

```
    precision[i], recall[i], _ =
precision_recall_curve(binarized_labels_class_i,
predicted_probabilities_class_i_resampled)
    average_precision[i] = average_precision_score(binarized_labels_class_i,
predicted_probabilities_class_i_resampled)

# Plot Precision-Recall curves
plt.figure(figsize=(10, 7))
for i in range(num_classes):
    plt.plot(recall[i], precision[i], label=f'Class {i} (AP =
{average_precision[i]:.2f})')

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve for Each Class')
plt.legend(loc='best')
plt.show()
```

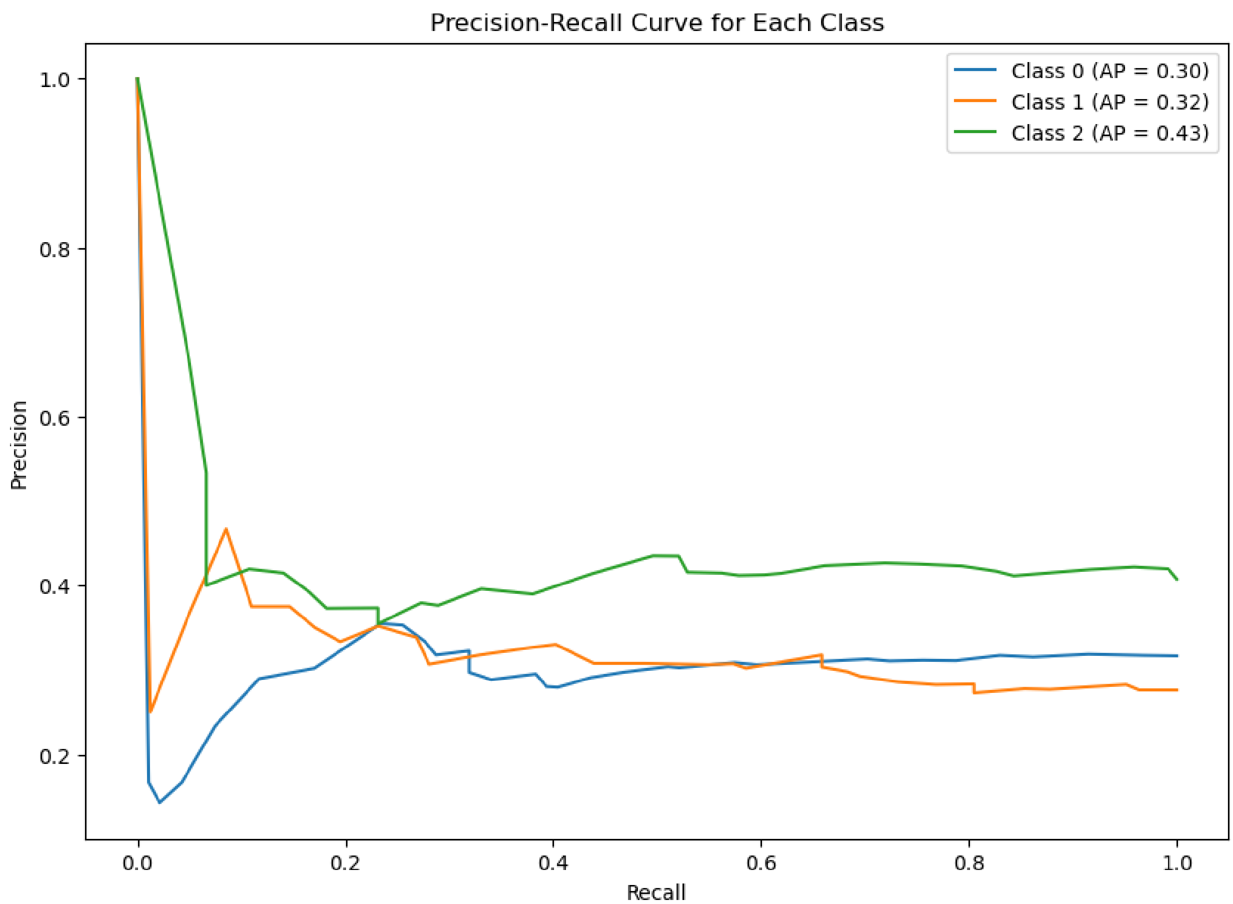


```
Test Loss: 0.1366, Test Accuracy: 97.31%
```

Confusion Matrix:

```
[[ 92   0   2]
 [  0  77   5]
 [  0   1 120]]
```

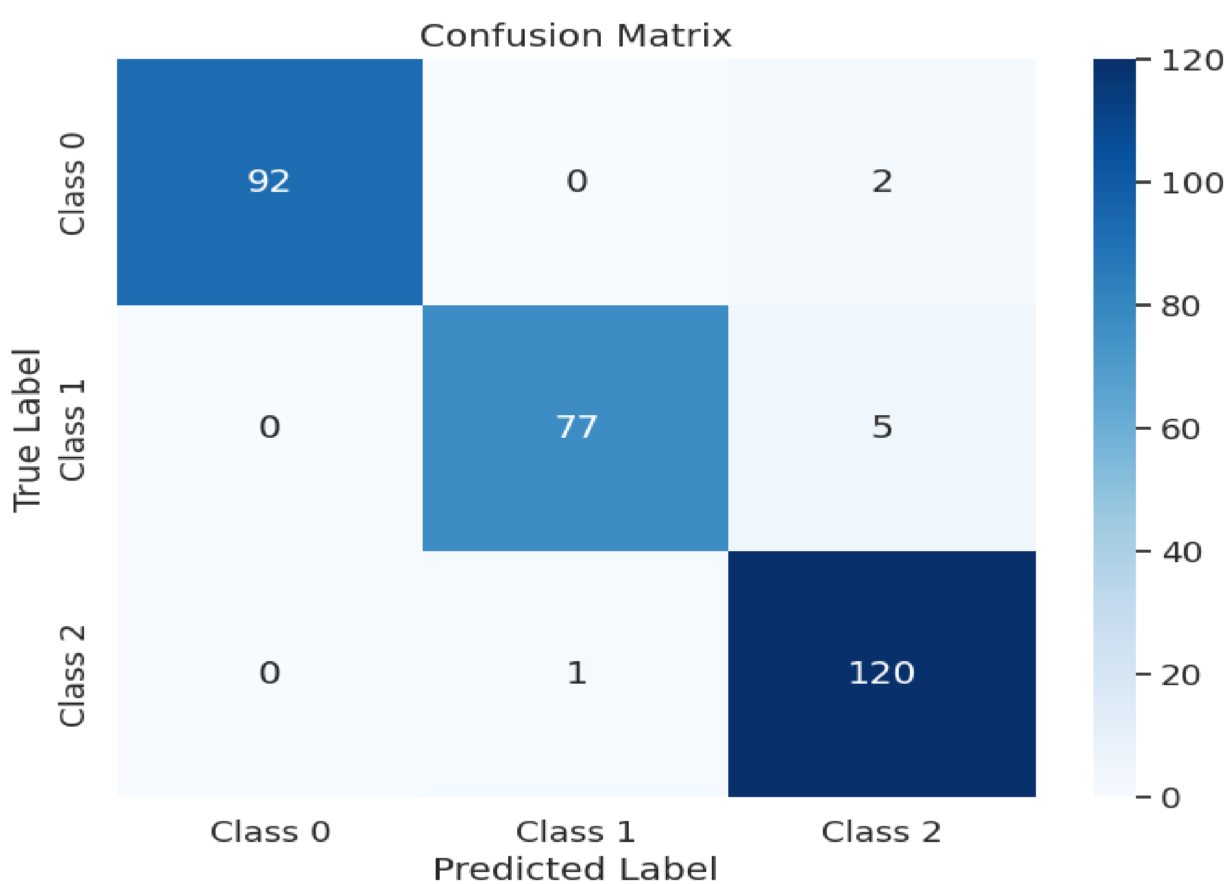
```
Class 0: Ground Truth Shape = (297,), Predicted Shape = (32,)
Class 1: Ground Truth Shape = (297,), Predicted Shape = (32,)
Class 2: Ground Truth Shape = (297,), Predicted Shape = (32,)
```



Confusion matrix Visualization:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
```

```
import seaborn as sns

conf_matrix_data = np.array([[92, 0, 2],
                             [0, 77, 5],
                             [0, 1, 120]])


# Plot the confusion matrix using seaborn and matplotlib
plt.figure(figsize=(8, 6))
sns.set(font_scale=1.2)
sns.heatmap(conf_matrix_data, annot=True, fmt="d", cmap="Blues",
            xticklabels=["Class 0", "Class 1", "Class 2"],
            yticklabels=["Class 0", "Class 1", "Class 2"])
plt.title("Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```



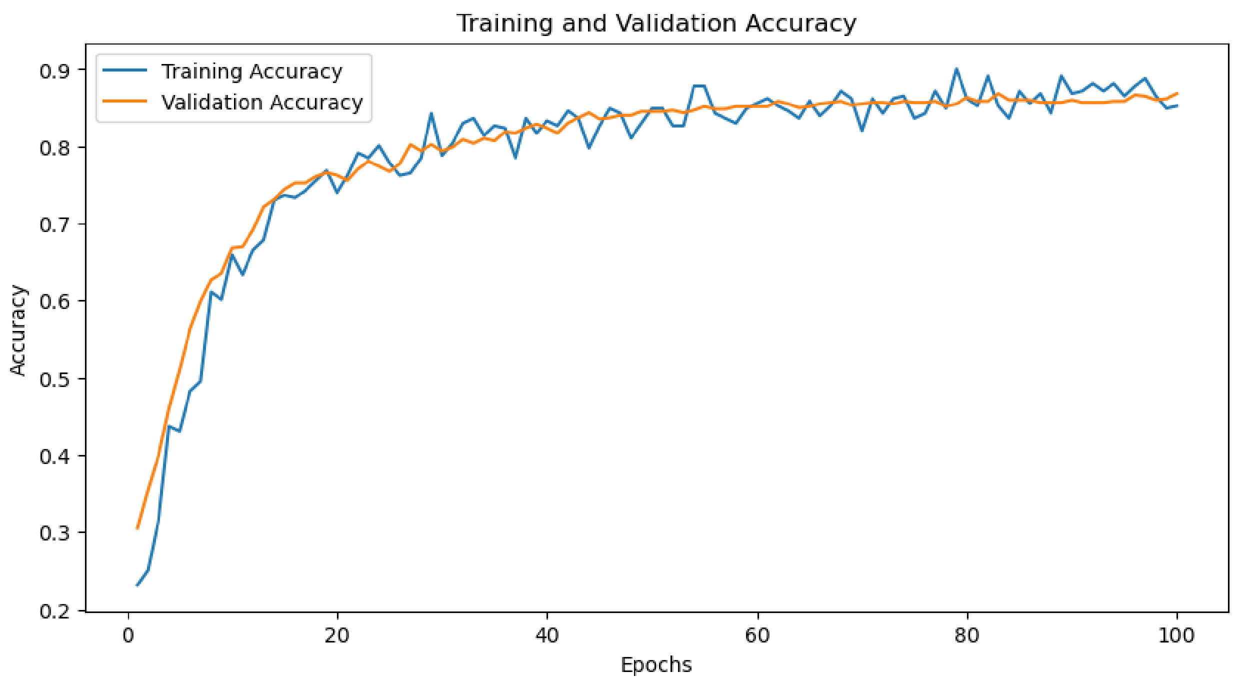
Without freezing:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import random_split, DataLoader
from torchvision import transforms, utils
from torchvision.datasets import ImageFolder
from torchvision.models import resnet34
import matplotlib.pyplot as plt
from sklearn.metrics import precision_recall_curve, auc
from sklearn.preprocessing import label_binarize
from PIL import Image
import torchvision.transforms.functional as F
import numpy as np
from itertools import cycle
from sklearn.metrics import confusion_matrix

# Define the loss function and optimizer with L2 regularization
criterion = nn.CrossEntropyLoss()
weight_decay = 1e-5  # You can experiment with different values
optimizer = optim.SGD(model.fc.parameters(), lr=0.0001, momentum=0.9,
weight_decay=weight_decay)

# Learning rate scheduler
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, patience=3,
factor=0.1, verbose=True)

# Early stopping
best_val_loss = float('inf')
patience = 5
counter = 0

# Lists to store training, validation, and testing metrics for each epoch
train_losses, train_accuracies = [], []
val_losses, val_accuracies = [], []

# Training loop
num_epochs = 100
for epoch in range(num_epochs):
    model.train()
    epoch_train_loss = 0.0
    correct_train = 0
    total_train = 0
```

```
for data, targets in train_loader:
    optimizer.zero_grad()
    outputs = model(data)
    loss = criterion(outputs, targets)
    loss.backward()
    optimizer.step()

    epoch_train_loss += loss.item()
    _, predicted = torch.max(outputs.data, 1)
    #total_train += labels.len
    correct_train += (predicted == targets).sum().item()

train_loss = epoch_train_loss / len(train_loader)
train_accuracy = (correct_train / len(train_loader.dataset))

# Validate the model after each epoch
model.eval()
epoch_val_loss = 0.0
correct_val = 0
total_val = 0

with torch.no_grad():
    for inputs, labels in valid_loader:
        outputs = model(inputs)
        val_loss = criterion(outputs, labels)
        epoch_val_loss += val_loss.item()

        _, predicted = torch.max(outputs.data, 1)
        #total_val += labels.size(0)
        correct_val += (predicted == labels).sum().item()

val_loss = epoch_val_loss / len(valid_loader)
val_accuracy = (correct_val / len(valid_loader.dataset))

# Append metrics to lists
train_losses.append(train_loss)
train_accuracies.append(train_accuracy)
val_losses.append(val_loss)
val_accuracies.append(val_accuracy)


# Learning rate scheduler step
scheduler.step(val_loss)
```

```
    # Early stopping check
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        counter = 0
    else:
        counter += 1
        if counter >= patience:
            break

    # Print metrics for each epoch
    print(f'Epoch {epoch+1}/{num_epochs}, '
          f'Training Loss: {train_loss:.4f}, Training Accuracy: {train_accuracy
* 100:.2f}%, '
          f'Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_accuracy
* 100:.2f}%, ')

# Plotting graphs
epochs = list(range(1, len(train_losses) + 1))

# Graph 1: Training and Validation Loss
plt.figure(figsize=(10, 5))
plt.plot(epochs, train_losses, label='Training Loss')
plt.plot(epochs, val_losses, label='Validation Loss')
vertical_lines = [19, 39, 59, 79]
for line in vertical_lines:
    plt.axvline(x=line, color='red')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.show()

# Graph 2: Training and Validation Accuracy
plt.figure(figsize=(10, 5))
plt.plot(epochs, train_accuracies, label='Training Accuracy')
plt.plot(epochs, val_accuracies, label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()
plt.show()
```

```
Epoch 100/100, Training Loss: 0.4205, Training Accuracy: 85.21%, Validation
Loss: 0.4369, Validation Accuracy: 86.80%,
```




```
import torch
import torch.nn.functional as F
import matplotlib.pyplot as plt
```

```
from sklearn.metrics import confusion_matrix, precision_recall_curve,
average_precision_score
from sklearn.preprocessing import label_binarize
from sklearn.utils import resample

# Lists to store testing metrics
test_losses, test_accuracies = [], []

# Testing loop with feature map visualization
model.eval()
epoch_test_loss = 0.0
correct_test = 0
total_test = 0
all_labels = []
all_predictions = []
all_probabilities = []

# Define a hook to capture feature maps
def visualize_hook(module, input, output):
    plt.figure(figsize=(15, 15))
    for i in range(output.size(1)):
        plt.subplot(8, 8, i + 1)
        plt.imshow(output[0, i].detach().cpu().numpy(), cmap="gray")
        plt.axis("off")
    plt.show()

# Choose a specific layer to visualize
layer_to_visualize = model.layer1[0].conv1  # Change this to the desired layer
in your model
hook = layer_to_visualize.register_forward_hook(visualize_hook)

image = r"C:\Users\Nissanth
NA\Downloads\Path1\test\Pentane\Pentane_Full_0474.JPG"
im = Image.open(image)
x = val_transform(im).unsqueeze(0)
_ = model(x)

with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        probabilities = F.softmax(outputs, dim=1)  # Apply softmax
        all_probabilities.append(probabilities.numpy())
        test_loss = criterion(outputs, labels)
        epoch_test_loss += test_loss.item()
```

```
        _, predicted = torch.max(probabilities, 1)
        total_test += labels.size(0)
        correct_test += (predicted == labels).sum().item()

        all_labels.append(labels.numpy())
        all_predictions.append(predicted.numpy())

# Calculate and print test metrics
test_loss = epoch_test_loss / len(test_loader)
test_accuracy = correct_test / total_test
print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy * 100:.2f}%")

# Remove the hook
hook.remove()

num_classes = 3

# Flatten the lists
all_labels = [item for sublist in all_labels for item in sublist]
all_predictions = [item for sublist in all_predictions for item in sublist]

# Generate confusion matrix
cm = confusion_matrix(all_labels, all_predictions)
print("Confusion Matrix:")
print(cm)

# Binarize the labels for precision-recall curve
binarized_labels = label_binarize(all_labels, classes=list(range(num_classes)))

precision = dict()
recall = dict()
average_precision = dict()
for i in range(num_classes):
    binarized_labels_class_i = binarized_labels[:, i]
    predicted_probabilities_class_i = np.array(all_probabilities[i])[:, i]
    print(f"Class {i}: Ground Truth Shape = {binarized_labels_class_i.shape},
Predicted Shape = {predicted_probabilities_class_i.shape}")

    # Resample predicted probabilities to match the length of the ground truth
    predicted_probabilities_class_i_resampled =
resample(predicted_probabilities_class_i,
n_samples=len(binarized_labels_class_i), random_state=42)
```

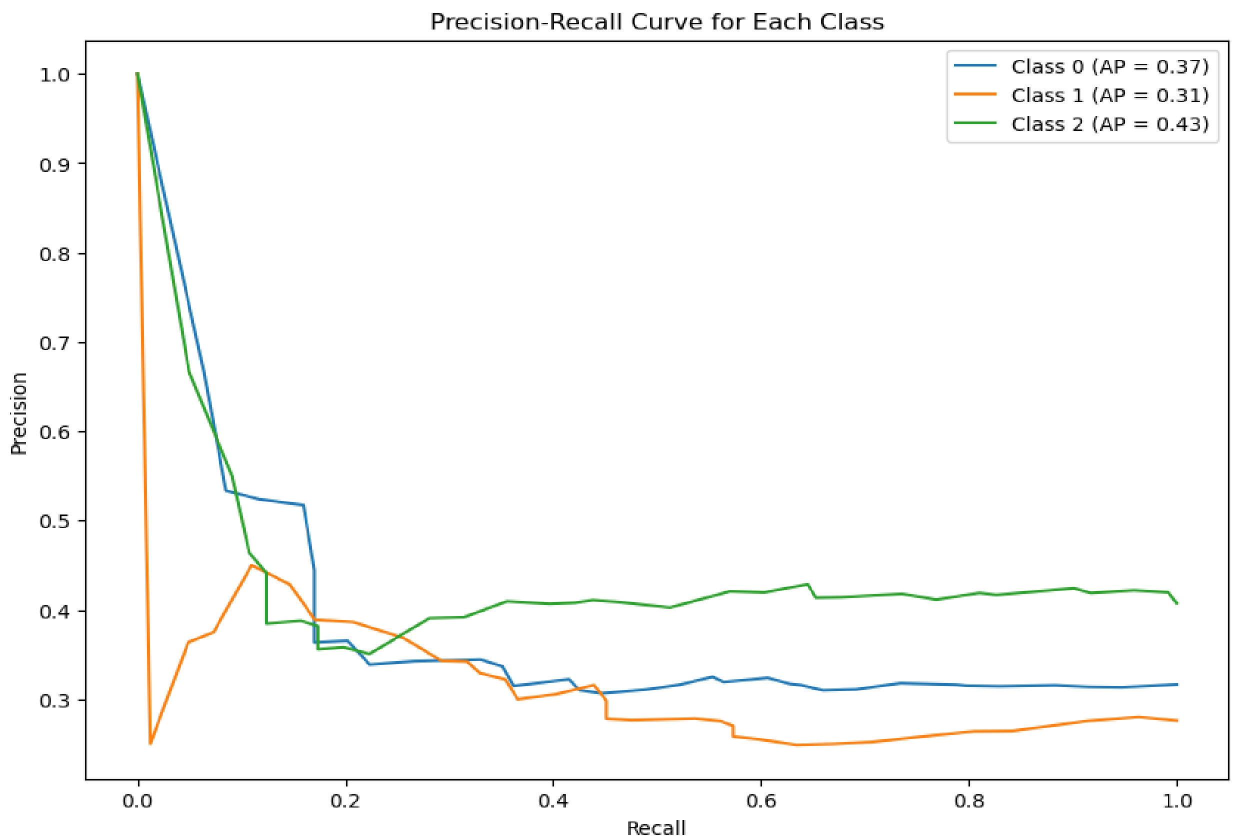
```
    precision[i], recall[i], _ =
precision_recall_curve(binarized_labels_class_i,
predicted_probabilities_class_i_resampled)
    average_precision[i] = average_precision_score(binarized_labels_class_i,
predicted_probabilities_class_i_resampled)

# Plot Precision-Recall curves
plt.figure(figsize=(10, 7))
for i in range(num_classes):
    plt.plot(recall[i], precision[i], label=f'Class {i} (AP =
{average_precision[i]:.2f})')

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve for Each Class')
plt.legend(loc='best')
plt.show()
```

```
Test Loss: 0.4367, Test Accuracy: 84.85%
```

Confusion Matrix:

```
[[ 78   0  16]
 [  1  60  21]
 [  2   5 114]]
```

```
Class 0: Ground Truth Shape = (297,), Predicted Shape = (32,)
Class 1: Ground Truth Shape = (297,), Predicted Shape = (32,)
Class 2: Ground Truth Shape = (297,), Predicted Shape = (32,)
```



Confusion Matrix Visualization:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns

conf_matrix_data = np.array([[78, 0, 16],
                             [1, 60, 21],
                             [2, 5, 114]])



# Plot the confusion matrix using seaborn and matplotlib
plt.figure(figsize=(8, 6))
sns.set(font_scale=1.2)
sns.heatmap(conf_matrix_data, annot=True, fmt="d", cmap="Blues",
            xticklabels=["Class 0", "Class 1", "Class 2"],
            yticklabels=["Class 0", "Class 1", "Class 2"])
plt.title("Confusion Matrix")
```

```
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```