Department of Electronic and Telecommunication Engineering University of Moratuwa

EN3150 - Pattern Recognition

Assignment 03

Simple convolutional neural network to perform classification

210141U	DISSANAYAKA D.M.S.P.
210341H	LIYANAARACHCHI L.A.S.
210303U	KULASINGHAM P.N.
210705E	WICKRAMASINGHA M.P.D.N.

Submission Date: 12.12.2024

Contents

1 CNN for Image Classification **3**
1.1 Prepare the Environment . . . 3
1.2 Add Dataset . . . 3
1.3 Split the dataset into training, validation, and testing . . . 3
1.4 Build the CNN Model . . . 4
1.4.1 Check Cuda Availability and Version . . . 5
1.5 Custom CNN Model Architecture . . . 6
1.6 Activation Function Selection . . . 7
1.6.1 ReLU (Rectified Linear Unit) . . . 7
1.6.2 Softmax (used via log_softmax) . . . 7
1.7 Train the Model . . . 7
1.8 Adam Optimizer over SGD . . . 10
1.9 Sparse Categorical Crossentropy as the Loss Function . . . 11
1.10 Testing the model . . . 11
1.11 Plot training and validation loss for 0.0001, 0.001, 0.01, and 0.1 . . . 13

2 Comapre the Network with state-of-the-art Networks **23**
2.1 Dataset Overview . . . 23
2.2 ResNet . . . 24
2.2.1 Loading the pretrained model . . . 24
2.2.2 Fine-tuning the model . . . 24
2.2.3 Evaluating the model . . . 26
2.3 DenseNet . . . 30
2.3.1 Loading the pretrained model . . . 30
2.3.2 Fine tuning the model . . . 30
2.3.3 Evaluating the model . . . 32
2.4 Selecting pre-trained model or architecture . . . 35
2.5 Fine-tuning the pre-trained model . . . 35
2.6 Train the fine-tuned model . . . 36
2.7 Training and validation loss values . . . 36
2.8 Evaluate the fine-tuned model . . . 36
2.9 Compare the test accuracy . . . 36
2.10 Discussion trade-offs, advantages, and limitations . . . 36
2.10.1 Custom Model . . . 36
2.10.2 Pretrained Model . . . 37

1 CNN for Image Classification

1.1 Prepare the Environment

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

Listing 1: Python Code for Data Loading and Transformation

1.2 Add Dataset

```python
dataset_path = './realwaste/realwaste-main/RealWaste'
```

Listing 2: Dataset Path Example

1.3 Split the dataset into training, validation, and testing

```python
# Define transformations
transform = transforms.Compose([
    transforms.Resize((128, 128)),  # Resize images to 128x128
    transforms.ToTensor(),  # Convert images to PyTorch tensors
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # Normalize (mean, std for RGB)
])

# Load the dataset
dataset = datasets.ImageFolder(root=dataset_path, transform=transform)

# Split into training and validation datasets
train_size = int(0.6 * len(dataset))
test_size = int(0.2 * len(dataset))
val_size = len(dataset) - train_size - test_size
train_dataset, test_dataset, val_dataset = torch.utils.data.random_split(dataset, [train_size, test_size, val_size])

# Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)

# Print class names
print("Classes:", dataset.classes)
```

Listing 3: Dataset Loading and Transformation with PyTorch

Classes: `['Cardboard', 'Food Organics', 'Glass', 'Metal', 'Miscellaneous Trash', 'Paper', 'Plastic', 'Textile Trash', 'Vegetation']`

1.4 Build the CNN Model

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class CustomCNN(nn.Module):
    def __init__(self, x1, m1, x2, m2, x3, d, K):
        super(CustomCNN, self).__init__()

        # First Convolutional Layer
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=x1, kernel_size=m1, stride=1, padding=m1 // 2)
        self.activation1 = nn.ReLU()

        # MaxPooling Layer 1
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Second Convolutional Layer
        self.conv2 = nn.Conv2d(in_channels=x1, out_channels=x2, kernel_size=m2, stride=1, padding=m2 // 2)
        self.activation2 = nn.ReLU()

        # MaxPooling Layer 2
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully Connected Layer
        self.fc1 = nn.Linear(x2 * (128 // 4) * (128 // 4), x3)  # Adjust dimensions based on input size and pooling
        self.activation3 = nn.ReLU()
        self.dropout = nn.Dropout(d)  # Dropout layer

        # Output Layer
        self.fc2 = nn.Linear(x3, K)

    def forward(self, x):
        x = self.conv1(x)
        x = self.activation1(x)
        x = self.pool1(x)

        x = self.conv2(x)
        x = self.activation2(x)
        x = self.pool2(x)

        x = torch.flatten(x, 1)  # Flatten the output for the fully connected layer

        x = self.fc1(x)
        x = self.activation3(x)
        x = self.dropout(x)

        x = self.fc2(x)
        return F.log_softmax(x, dim=1)  # Softmax activation

# Define the model parameters
x1, m1 = 32, 3  # Filters and kernel size for the first convolutional layer
```

```python
x2, m2 = 64, 3  # Filters and kernel size for the second convolutional layer
x3 = 128         # Number of units in the fully connected layer
d = 0.5          # Dropout rate
K = len(dataset.classes)  # Number of output classes

# Instantiate the model
model = CustomCNN(x1, m1, x2, m2, x3, d, K)

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

Listing 4: Custom CNN Model in PyTorch

CustomCNN Model Architecture:

(conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(activation1): ReLU()
(pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(activation2): ReLU()
(pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(fc1): Linear(in_features=65536, out_features=128, bias=True)
(activation3): ReLU()
(dropout): Dropout(p=0.5, inplace=False)
(fc2): Linear(in_features=128, out_features=9, bias=True)

1.4.1 Check Cuda Availability and Version

```python
# Check if CUDA is available
cuda_available = torch.cuda.is_available()
print(f"CUDA available: {cuda_available}")

# If CUDA is available, check the PyTorch CUDA version
if cuda_available:
    print(f"CUDA version supported by PyTorch: {torch.version.cuda}")
    print(f"Number of GPUs available: {torch.cuda.device_count()}")
    print(f"Current GPU: {torch.cuda.get_device_name(0)}")
else:
    print("CUDA is not available in this PyTorch installation.")
```

Listing 5: Check CUDA Availability in PyTorch

CUDA available: True
CUDA version supported by PyTorch: 11.8
Number of GPUs available: 1
Current GPU: NVIDIA GeForce GTX 1650

1.5 Custom CNN Model Architecture

Conv1 Layer:
Input Channels: 3 (RGB)
Output Channels: 32
Kernel Size: 3x3
Stride: 1
Padding: 1 (to maintain spatial dimensions)
Activation Function: ReLU
Output Shape after Pooling: (32, 64, 64)

Conv2 Layer:
Input Channels: 32 (from Conv1)
Output Channels: 64
Kernel Size: 3x3
Stride: 1
Padding: 1 (to maintain spatial dimensions)
Activation Function: ReLU
Output Shape after Pooling: (64, 32, 32)

Flatten Layer:
Input Shape: (64, 32, 32)
Output Shape: 65536 features

Fully Connected Layer (fc1):
Input Features: 65536
Output Units: 128
Activation Function: ReLU

Dropout Layer:
Dropout Rate: 0.5

Fully Connected Layer (fc2):
Input Features: 128
Output Classes: K (number of output classes)

Softmax Output Layer:
Output: Log probabilities over K classes

Number of Parameters Calculation:

Conv1 Parameters:
Formula: $(3 \times 3 \times 3 + 1) \times 32$
Calculation: $(27 + 1) \times 32 = 896$

Conv2 Parameters:
Formula: $(32 \times 3 \times 3 + 1) \times 64$
Calculation: $(288 + 1) \times 64 = 18496$

Fully Connected Layer 1 Parameters:

Formula: $(64 \times 32 \times 32 + 1) \times 128$
Calculation: $(65536 + 1) \times 128 = 8388608$

Fully Connected Layer 2 Parameters:
Formula: $(128 + 1) \times K$
Calculation: $(129) \times K$

1.6 Activation Function Selection

1.6.1 ReLU (Rectified Linear Unit)

ReLU is chosen as the activation function for intermediate layers due to its efficiency and simplicity. It helps mitigate the vanishing gradient problem by maintaining non-zero gradients for positive values, enabling faster and more stable training. Additionally, ReLU introduces sparsity in the activations by setting negative values to zero, which:

- Enhances model interpretability.
- Helps reduce overfitting.

From a computational perspective, ReLU is simple and more efficient compared to other nonlinear activation functions like sigmoid or tanh, making it an ideal choice for deep learning models.

1.6.2 Softmax (used via log_softmax)

Softmax is applied in the output layer as it converts raw outputs (logits) into a probability distribution over K classes. This is essential for multi-class classification tasks where:

- Each output class must have a probability between 0 and 1.
- The predicted probabilities across all classes must sum to 1.

Softmax ensures compatibility with the CrossEntropyLoss function, which:

- Directly optimizes the log probabilities generated by `log_softmax`.
- Ensures proper gradient updates during backpropagation.

By using Softmax, the model can effectively handle classification problems with K classes.

1.7 Train the Model

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Model, loss function, and optimizer
model = CustomCNN(x1, m1, x2, m2, x3, d, K)  # Define your
    model
```

```python
device = torch.device("cuda" if torch.cuda.is_available() else
    "cpu")
model.to(device)

criterion = nn.CrossEntropyLoss()  # Loss function for
    classification
optimizer = optim.Adam(model.parameters(), lr=0.001)  # Adam
    optimizer

# Training parameters
num_epochs = 20  # Number of epochs
train_losses = []
val_losses = []

# Training and validation loop
for epoch in range(num_epochs):
    # Training phase
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()  # Clear gradients
        outputs = model(images)  # Forward pass
        loss = criterion(outputs, labels)  # Compute loss
        loss.backward()  # Backward pass
        optimizer.step()  # Update weights
        running_loss += loss.item() * images.size(0)


        # Train Accuracy
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    epoch_train_loss = running_loss / len(train_loader.dataset
        )
    train_losses.append(epoch_train_loss)

    # Calculate Train Accuracy as percentage
    train_accuracy = 100 * correct / total

    # Validation phase
    model.eval()
    val_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(
                device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item() * images.size(0)

            # Accuracy
```

```
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    epoch_val_loss = val_loss / len(val_loader.dataset)
    val_losses.append(epoch_val_loss)
    val_accuracy = 100 * correct / total

    print(f"Epoch {epoch+1}/{num_epochs}, "
          f"Train Loss: {epoch_train_loss:.4f}, "
          f"Train Accuracy: {train_accuracy:.2f}%",
          f"Validation Loss: {epoch_val_loss:.4f}, "
          f"Validation Accuracy: {val_accuracy:.2f}%")
```

Training and Validation Results

Epoch 1/20, Train Loss: 2.0285, Train Accuracy: 25.29% Validation Loss: 1.6530, Validation Accuracy: 48.05%
Epoch 2/20, Train Loss: 1.6442, Train Accuracy: 41.46% Validation Loss: 1.3348, Validation Accuracy: 55.21%
Epoch 3/20, Train Loss: 1.4046, Train Accuracy: 50.23% Validation Loss: 1.1384, Validation Accuracy: 60.36%
Epoch 4/20, Train Loss: 1.2585, Train Accuracy: 55.95% Validation Loss: 1.1247, Validation Accuracy: 58.36%
Epoch 5/20, Train Loss: 1.1186, Train Accuracy: 60.22% Validation Loss: 1.0748, Validation Accuracy: 61.93%
Epoch 6/20, Train Loss: 0.9956, Train Accuracy: 64.57% Validation Loss: 0.9856, Validation Accuracy: 63.83%
Epoch 7/20, Train Loss: 0.9213, Train Accuracy: 66.64% Validation Loss: 1.0158, Validation Accuracy: 62.46%
Epoch 8/20, Train Loss: 0.7986, Train Accuracy: 70.36% Validation Loss: 0.9215, Validation Accuracy: 67.72%
Epoch 9/20, Train Loss: 0.7113, Train Accuracy: 73.41% Validation Loss: 0.9926, Validation Accuracy: 63.72%
Epoch 10/20, Train Loss: 0.6452, Train Accuracy: 76.53% Validation Loss: 0.9566, Validation Accuracy: 67.82%
Epoch 11/20, Train Loss: 0.5382, Train Accuracy: 79.62% Validation Loss: 0.9583, Validation Accuracy: 66.46%
Epoch 12/20, Train Loss: 0.4669, Train Accuracy: 82.67% Validation Loss: 1.0385, Validation Accuracy: 67.30%
Epoch 13/20, Train Loss: 0.4692, Train Accuracy: 83.27% Validation Loss: 1.0108, Validation Accuracy: 69.19%
Epoch 14/20, Train Loss: 0.4084, Train Accuracy: 84.92% Validation Loss: 1.0913, Validation Accuracy: 66.67%
Epoch 15/20, Train Loss: 0.3217, Train Accuracy: 87.65% Validation Loss: 1.0809, Validation Accuracy: 67.40%
Epoch 16/20, Train Loss: 0.3394, Train Accuracy: 88.71% Validation Loss: 1.1911, Validation Accuracy: 65.83%
Epoch 17/20, Train Loss: 0.2878, Train Accuracy: 89.55% Validation Loss: 1.2849, Validation Accuracy: 65.30%

Epoch 18/20, Train Loss: 0.2712, Train Accuracy: 90.04% Validation Loss: 1.1358, Validation Accuracy: 68.03%
Epoch 19/20, Train Loss: 0.2313, Train Accuracy: 91.72% Validation Loss: 1.2599, Validation Accuracy: 67.09%
Epoch 20/20, Train Loss: 0.2782, Train Accuracy: 89.93% Validation Loss: 1.3685, Validation Accuracy: 66.25%

```
import matplotlib.pyplot as plt

plt.plot(range(1, num_epochs+1), train_losses, label='Training Loss')
plt.plot(range(1, num_epochs+1), val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title(f'Training and Validation Loss')
plt.show()
```

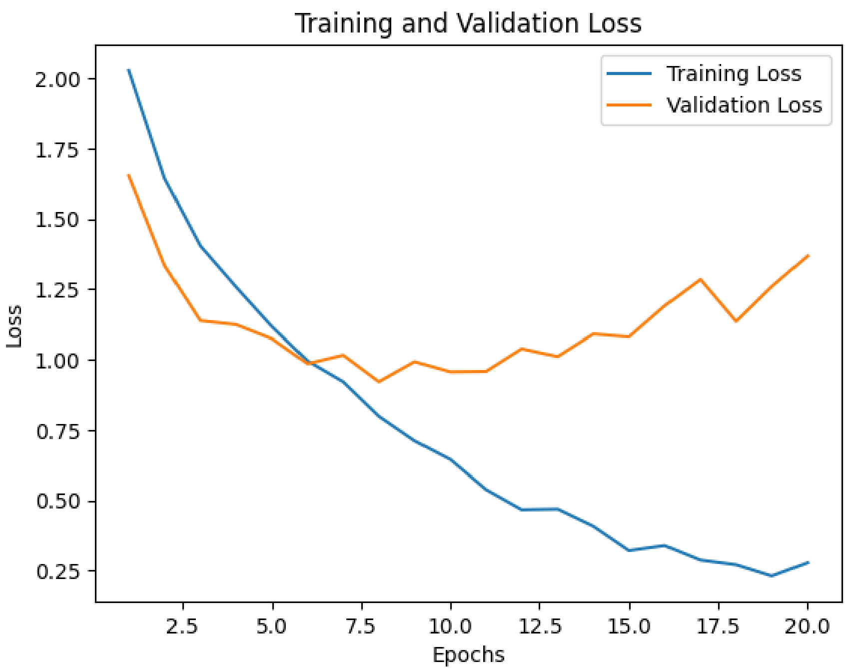

Figure 1: Training and Validation Loss

1.8 Adam Optimizer over SGD

The Adam optimizer is chosen because it combines the benefits of Momentum and RMSProp, making it well-suited for deep learning tasks. It dynamically adjusts the learning rate for each parameter using estimates of the first and second moments of gradients. This enables:

- Faster convergence.
- Better handling of sparse gradients.

In contrast, SGD (Stochastic Gradient Descent) with a fixed learning rate may require more manual tuning and tends to converge slower without additional techniques like momentum.

For tasks like classification, where the dataset may have varying complexity, Adam provides better efficiency and performance out of the box.

1.9 Sparse Categorical Crossentropy as the Loss Function

CrossEntropyLoss is specifically designed for classification problems with mutually exclusive classes. It works by comparing the predicted probability distribution over classes with the true class labels, penalizing incorrect predictions. This makes it highly suitable for multi-class classification tasks like this one.

The loss function's mathematical formulation inherently aligns with softmax outputs, ensuring proper gradient updates for learning discriminative features.

1.10 Testing the model

```python
from sklearn.metrics import confusion_matrix, precision_score,
    recall_score, accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns

# Evaluate the model on the testing dataset
model.eval()
test_loss = 0.0
correct = 0
total = 0
all_labels = []
all_preds = []

# Iterate over the test set
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss = criterion(outputs, labels)
        test_loss += loss.item() * images.size(0)

        # Get predictions
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        # Store all labels and predictions for confusion
            matrix
        all_labels.extend(labels.cpu().numpy())
        all_preds.extend(predicted.cpu().numpy())

# Calculate Test Accuracy
```

```
test_accuracy = 100 * correct / total
test_loss /= len(test_loader.dataset)

# Print test accuracy and loss
print(f"Test Loss: {test_loss:.4f}")
print(f"Test Accuracy: {test_accuracy:.2f}%")

# Confusion Matrix
cm = confusion_matrix(all_labels, all_preds)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels
    =range(K), yticklabels=range(K))
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()

# Precision and Recall
precision = precision_score(all_labels, all_preds, average='
    weighted')
recall = recall_score(all_labels, all_preds, average='weighted
    ')

print(f"Precision (Weighted): {precision:.4f}")
print(f"Recall (Weighted): {recall:.4f}")

# Evaluate train accuracy
model.train()
train_loss = 0.0
correct = 0
total = 0
with torch.no_grad():
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss = criterion(outputs, labels)
        train_loss += loss.item() * images.size(0)

        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

train_accuracy = 100 * correct / total
print(f"Train Accuracy: {train_accuracy:.2f}%")
```

Listing 6: Model Evaluation and Metrics

Test Loss	1.4439
Test Accuracy	65.47%
Precision (Weighted)	0.6611
Recall (Weighted)	0.6547
Train Accuracy	91.79%

Table 1: Test and Train Performance Metrics

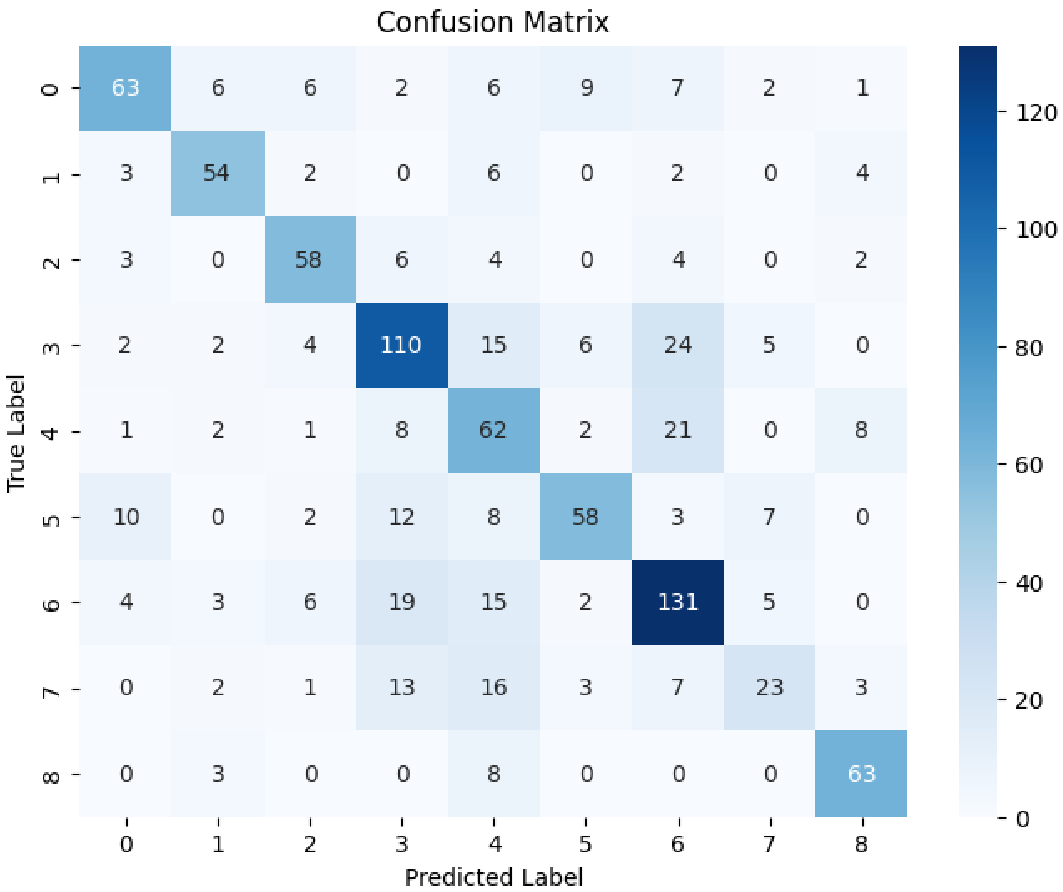

Figure 2: Confusion matrix

1.11 Plot training and validation loss for 0.0001, 0.001, 0.01, and 0.1

```python
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# Model, loss function, and optimizer
model = CustomCNN(x1, m1, x2, m2, x3, d, K)  # Define your
    model
device = torch.device("cuda" if torch.cuda.is_available() else
     "cpu")
model.to(device)

LR = [0.0001, 0.001, 0.01, 0.1]

for lr in LR:
    print("--------------------------------")
    print(f"Learning Rate: {lr}")
    criterion = nn.CrossEntropyLoss()  # Loss function for
        classification
```

```
    optimizer = optim.Adam(model.parameters(), lr=lr)  # Adam optimizer

    # Training parameters
    num_epochs = 20  # Number of epochs
    train_losses = []
    val_losses = []

    # Training and validation loop
    for epoch in range(num_epochs):
        # Training phase
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)

            optimizer.zero_grad()  # Clear gradients
            outputs = model(images)  # Forward pass
            loss = criterion(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update weights

            running_loss += loss.item() * images.size(0)

            # Train Accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_train_loss = running_loss / len(train_loader.dataset)
        train_losses.append(epoch_train_loss)

        # Calculate Train Accuracy as percentage
        train_accuracy = 100 * correct / total

        # Validation phase
        model.eval()
        val_loss = 0.0
        correct = 0
        total = 0
        with torch.no_grad():
            for images, labels in val_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item() * images.size(0)

                # Accuracy
                _, predicted = torch.max(outputs, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        epoch_val_loss = val_loss / len(val_loader.dataset)
```

```python
        val_losses.append(epoch_val_loss)
        val_accuracy = 100 * correct / total

        print(f"Epoch {epoch+1}/{num_epochs}, "
              f"Train Loss: {epoch_train_loss:.4f}, "
              f"Train Accuracy: {train_accuracy:.2f}% ",
              f"Validation Loss: {epoch_val_loss:.4f}, "
              f"Validation Accuracy: {val_accuracy:.2f}%")

    print(f"Testing for Learning Rate: {lr}")
    model.eval()
    test_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            test_loss += loss.item() * images.size(0)

            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    test_loss /= len(test_loader.dataset)
    test_accuracy = 100 * correct / total

    print(f"Learning Rate: {lr} | Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.2f}%")

    plt.plot(range(1, num_epochs+1), train_losses, label='Training Loss')
    plt.plot(range(1, num_epochs+1), val_losses, label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.title(f'Training and Validation Loss for Learning Rate: {lr}')
    plt.show()
```

Listing 7: Training and Testing with Different Learning Rates

---

Learning Rate: 0.0001
Epoch 1/20, Train Loss: 2.0171, Train Accuracy: 27.46% Validation Loss: 1.7636, Validation Accuracy: 44.58%
Epoch 2/20, Train Loss: 1.7602, Train Accuracy: 37.60% Validation Loss: 1.5222, Validation Accuracy: 49.42%
Epoch 3/20, Train Loss: 1.6039, Train Accuracy: 44.16% Validation Loss: 1.4079, Validation Accuracy: 54.47%
Epoch 4/20, Train Loss: 1.5052, Train Accuracy: 47.14% Validation Loss: 1.3314, Validation Accuracy: 56.57%
Epoch 5/20, Train Loss: 1.4239, Train Accuracy: 50.68% Validation Loss:

1.2581, Validation Accuracy: 56.47%
Epoch 6/20, Train Loss: 1.3516, Train Accuracy: 52.79% Validation Loss: 1.2153, Validation Accuracy: 57.73%
Epoch 7/20, Train Loss: 1.2865, Train Accuracy: 55.98% Validation Loss: 1.1827, Validation Accuracy: 58.99%
Epoch 8/20, Train Loss: 1.2529, Train Accuracy: 56.09% Validation Loss: 1.1136, Validation Accuracy: 62.46%
Epoch 9/20, Train Loss: 1.1713, Train Accuracy: 59.42% Validation Loss: 1.0845, Validation Accuracy: 62.04%
Epoch 10/20, Train Loss: 1.1410, Train Accuracy: 60.19% Validation Loss: 1.0591, Validation Accuracy: 63.30%
Epoch 11/20, Train Loss: 1.0907, Train Accuracy: 60.96% Validation Loss: 1.0313, Validation Accuracy: 64.56%
Epoch 12/20, Train Loss: 1.0682, Train Accuracy: 62.93% Validation Loss: 1.0005, Validation Accuracy: 64.98%
Epoch 13/20, Train Loss: 1.0176, Train Accuracy: 63.77% Validation Loss: 1.0000, Validation Accuracy: 64.67%
Epoch 14/20, Train Loss: 0.9746, Train Accuracy: 66.26% Validation Loss: 0.9547, Validation Accuracy: 65.93%
Epoch 15/20, Train Loss: 0.9785, Train Accuracy: 64.22% Validation Loss: 0.9440, Validation Accuracy: 66.35%
Epoch 16/20, Train Loss: 0.9388, Train Accuracy: 67.98% Validation Loss: 0.9565, Validation Accuracy: 66.46%
Epoch 17/20, Train Loss: 0.8845, Train Accuracy: 69.87% Validation Loss: 0.9258, Validation Accuracy: 66.25%
Epoch 18/20, Train Loss: 0.8450, Train Accuracy: 70.57% Validation Loss: 0.9274, Validation Accuracy: 67.51%
Epoch 19/20, Train Loss: 0.8313, Train Accuracy: 69.69% Validation Loss: 0.9170, Validation Accuracy: 68.56%
Epoch 20/20, Train Loss: 0.8069, Train Accuracy: 70.99% Validation Loss: 0.9138, Validation Accuracy: 69.30%

Testing for Learning Rate: 0.0001
Learning Rate: 0.0001 — Test Loss: 0.9721, Test Accuracy: 65.16%

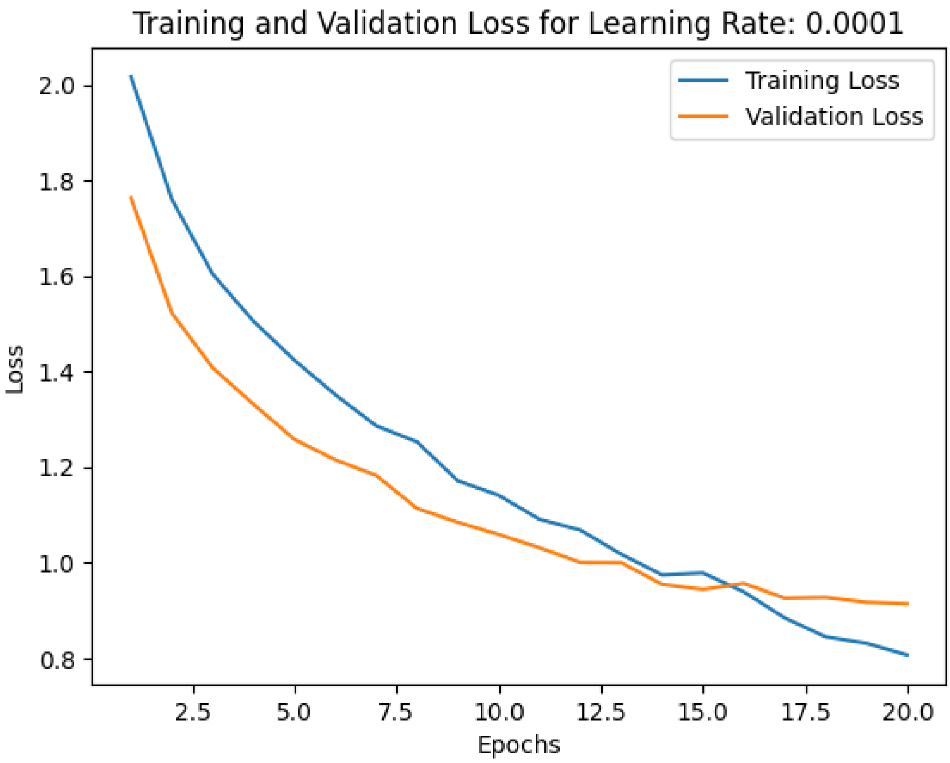

Figure 3: Learning Rate: 0.0001

Learning Rate: 0.001
Epoch 1/20, Train Loss: 1.3445, Train Accuracy: 51.56% Validation Loss: 1.0834, Validation Accuracy: 62.04%
Epoch 2/20, Train Loss: 1.1354, Train Accuracy: 58.82% Validation Loss: 1.0473, Validation Accuracy: 63.41%
Epoch 3/20, Train Loss: 1.0066, Train Accuracy: 64.43% Validation Loss: 1.0251, Validation Accuracy: 63.51%
Epoch 4/20, Train Loss: 0.9036, Train Accuracy: 67.66% Validation Loss: 1.0008, Validation Accuracy: 65.09%
Epoch 5/20, Train Loss: 0.8048, Train Accuracy: 70.26% Validation Loss: 0.9781, Validation Accuracy: 66.35%
Epoch 6/20, Train Loss: 0.7116, Train Accuracy: 74.39% Validation Loss: 0.9636, Validation Accuracy: 66.98%
Epoch 7/20, Train Loss: 0.6423, Train Accuracy: 76.01% Validation Loss: 1.0547, Validation Accuracy: 65.93%
Epoch 8/20, Train Loss: 0.5508, Train Accuracy: 79.97% Validation Loss: 1.0815, Validation Accuracy: 65.51%
Epoch 9/20, Train Loss: 0.5248, Train Accuracy: 80.88% Validation Loss: 1.0207, Validation Accuracy: 67.51%
Epoch 10/20, Train Loss: 0.4397, Train Accuracy: 83.58% Validation Loss: 1.1289, Validation Accuracy: 66.46%
Epoch 11/20, Train Loss: 0.4277, Train Accuracy: 83.30% Validation Loss:

1.1988, Validation Accuracy: 67.51%
Epoch 12/20, Train Loss: 0.4054, Train Accuracy: 84.95% Validation Loss: 1.1854, Validation Accuracy: 66.88%
Epoch 13/20, Train Loss: 0.3514, Train Accuracy: 86.92% Validation Loss: 1.1455, Validation Accuracy: 66.67%
Epoch 14/20, Train Loss: 0.3059, Train Accuracy: 88.53% Validation Loss: 1.2260, Validation Accuracy: 67.09%
Epoch 15/20, Train Loss: 0.2670, Train Accuracy: 89.55% Validation Loss: 1.4207, Validation Accuracy: 66.04%
Epoch 16/20, Train Loss: 0.2769, Train Accuracy: 89.41% Validation Loss: 1.2646, Validation Accuracy: 68.03%
Epoch 17/20, Train Loss: 0.2358, Train Accuracy: 91.93% Validation Loss: 1.3290, Validation Accuracy: 66.98%
Epoch 18/20, Train Loss: 0.2464, Train Accuracy: 90.60% Validation Loss: 1.3245, Validation Accuracy: 66.56%
Epoch 19/20, Train Loss: 0.2297, Train Accuracy: 90.88% Validation Loss: 1.3893, Validation Accuracy: 66.25%
Epoch 20/20, Train Loss: 0.2012, Train Accuracy: 92.00% Validation Loss: 1.4655, Validation Accuracy: 68.98%

Testing for Learning Rate: 0.001
Learning Rate: 0.001 — Test Loss: 1.4845, Test Accuracy: 66.63%

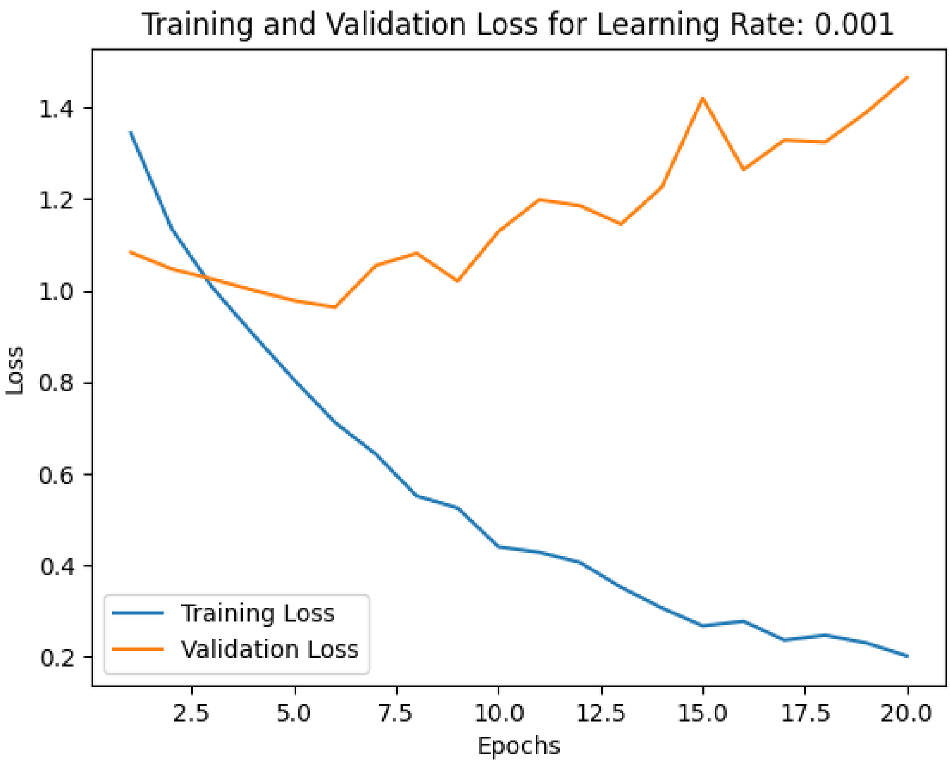

Figure 4: Learning Rate: 0.001

Learning Rate: 0.01
Epoch 1/20, Train Loss: 1.9594, Train Accuracy: 39.25% Validation Loss: 1.8079, Validation Accuracy: 33.96%
Epoch 2/20, Train Loss: 1.4480, Train Accuracy: 51.84% Validation Loss: 1.5077, Validation Accuracy: 45.95%
Epoch 3/20, Train Loss: 1.0856, Train Accuracy: 63.42% Validation Loss: 1.7044, Validation Accuracy: 48.90%
Epoch 4/20, Train Loss: 0.8039, Train Accuracy: 73.73% Validation Loss: 1.7050, Validation Accuracy: 49.42%
Epoch 5/20, Train Loss: 0.6031, Train Accuracy: 79.38% Validation Loss: 2.1100, Validation Accuracy: 41.11%
Epoch 6/20, Train Loss: 0.5099, Train Accuracy: 83.30% Validation Loss: 2.4433, Validation Accuracy: 45.85%
Epoch 7/20, Train Loss: 0.6892, Train Accuracy: 79.73% Validation Loss: 2.5181, Validation Accuracy: 41.22%
Epoch 8/20, Train Loss: 0.5362, Train Accuracy: 83.48% Validation Loss: 2.4963, Validation Accuracy: 40.38%
Epoch 9/20, Train Loss: 0.5140, Train Accuracy: 85.44% Validation Loss: 3.2499, Validation Accuracy: 37.54%
Epoch 10/20, Train Loss: 0.6119, Train Accuracy: 82.32% Validation Loss: 3.2854, Validation Accuracy: 43.11%
Epoch 11/20, Train Loss: 0.5154, Train Accuracy: 85.90% Validation Loss: 3.7873, Validation Accuracy: 42.27%
Epoch 12/20, Train Loss: 0.5454, Train Accuracy: 86.46% Validation Loss: 2.8227, Validation Accuracy: 38.49%
Epoch 13/20, Train Loss: 0.4634, Train Accuracy: 86.11% Validation Loss: 2.7984, Validation Accuracy: 43.11%
Epoch 14/20, Train Loss: 0.3785, Train Accuracy: 88.74% Validation Loss: 3.0848, Validation Accuracy: 43.11%
Epoch 15/20, Train Loss: 0.3061, Train Accuracy: 90.11% Validation Loss: 4.1324, Validation Accuracy: 40.38%
Epoch 16/20, Train Loss: 0.4191, Train Accuracy: 89.37% Validation Loss: 3.9586, Validation Accuracy: 42.27%
Epoch 17/20, Train Loss: 0.4112, Train Accuracy: 88.64% Validation Loss: 3.1025, Validation Accuracy: 40.80%
Epoch 18/20, Train Loss: 0.3348, Train Accuracy: 90.56% Validation Loss: 3.5894, Validation Accuracy: 43.53%
Epoch 19/20, Train Loss: 0.4569, Train Accuracy: 89.30% Validation Loss: 4.0916, Validation Accuracy: 40.17%
Epoch 20/20, Train Loss: 0.4883, Train Accuracy: 86.43% Validation Loss: 3.4627, Validation Accuracy: 39.01%

Testing for Learning Rate: 0.01
Learning Rate: 0.01 — Test Loss: 3.2796, Test Accuracy: 41.79%

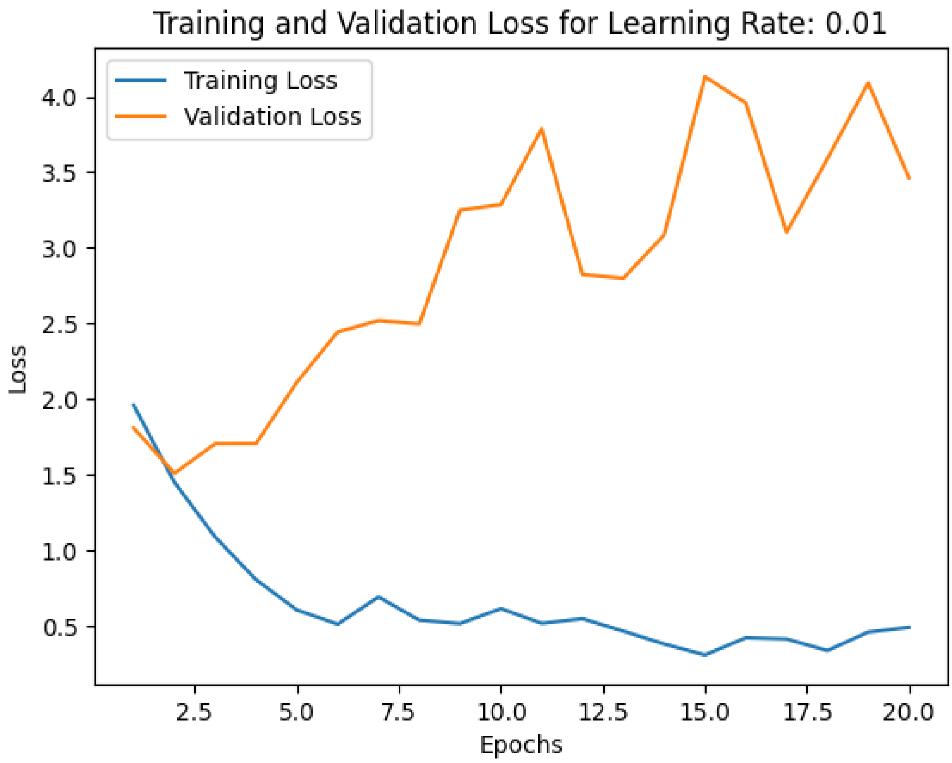

Figure 5: Learning Rate: 0.01

Learning Rate: 0.1
Epoch 1/20, Train Loss: 17.2835, Train Accuracy: 20.52% Validation Loss: 2.1676, Validation Accuracy: 17.14%
Epoch 2/20, Train Loss: 2.1520, Train Accuracy: 19.33% Validation Loss: 2.1776, Validation Accuracy: 17.14%
Epoch 3/20, Train Loss: 2.1473, Train Accuracy: 19.64% Validation Loss: 2.1865, Validation Accuracy: 17.14%
Epoch 4/20, Train Loss: 2.1508, Train Accuracy: 19.64% Validation Loss: 2.1825, Validation Accuracy: 17.14%
Epoch 5/20, Train Loss: 2.1469, Train Accuracy: 19.05% Validation Loss: 2.1679, Validation Accuracy: 17.14%
Epoch 6/20, Train Loss: 2.150
6, Train Accuracy: 18.77% Validation Loss: 2.1830, Validation Accuracy: 17.14% Epoch 7/20, Train Loss: 2.1479, Train Accuracy: 18.84% Validation Loss: 2.1785, Validation Accuracy: 17.14%
Epoch 8/20, Train Loss: 2.1466, Train Accuracy: 19.71% Validation Loss: 2.1761, Validation Accuracy: 16.09%
Epoch 9/20, Train Loss: 2.1510, Train Accuracy: 18.91% Validation Loss: 2.1902, Validation Accuracy: 17.14%
Epoch 10/20, Train Loss: 2.1473, Train Accuracy: 19.40% Validation Loss: 2.1805, Validation Accuracy: 17.14%
Epoch 11/20, Train Loss: 2.1508, Train Accuracy: 19.57% Validation Loss:

2.1717, Validation Accuracy: 17.14%
Epoch 12/20, Train Loss: 2.1516, Train Accuracy: 19.08% Validation Loss: 2.1907, Validation Accuracy: 17.14%
Epoch 13/20, Train Loss: 2.1457, Train Accuracy: 20.24% Validation Loss: 2.2028, Validation Accuracy: 17.14%
Epoch 14/20, Train Loss: 2.1535, Train Accuracy: 18.73% Validation Loss: 2.1788, Validation Accuracy: 17.14%
Epoch 15/20, Train Loss: 2.1487, Train Accuracy: 19.96% Validation Loss: 2.1807, Validation Accuracy: 17.14%
Epoch 16/20, Train Loss: 2.1483, Train Accuracy: 18.84% Validation Loss: 2.1727, Validation Accuracy: 17.14%
Epoch 17/20, Train Loss: 2.1495, Train Accuracy: 19.36% Validation Loss: 2.1702, Validation Accuracy: 17.14%
Epoch 18/20, Train Loss: 2.1499, Train Accuracy: 19.47% Validation Loss: 2.1704, Validation Accuracy: 17.14%
Epoch 19/20, Train Loss: 2.1468, Train Accuracy: 20.10% Validation Loss: 2.1739, Validation Accuracy: 17.14%
Epoch 20/20, Train Loss: 2.1489, Train Accuracy: 19.78% Validation Loss: 2.1723, Validation Accuracy: 17.14%

Testing for Learning Rate: 0.1
Learning Rate: 0.1 — Test Loss: 2.1360, Test Accuracy: 19.47%

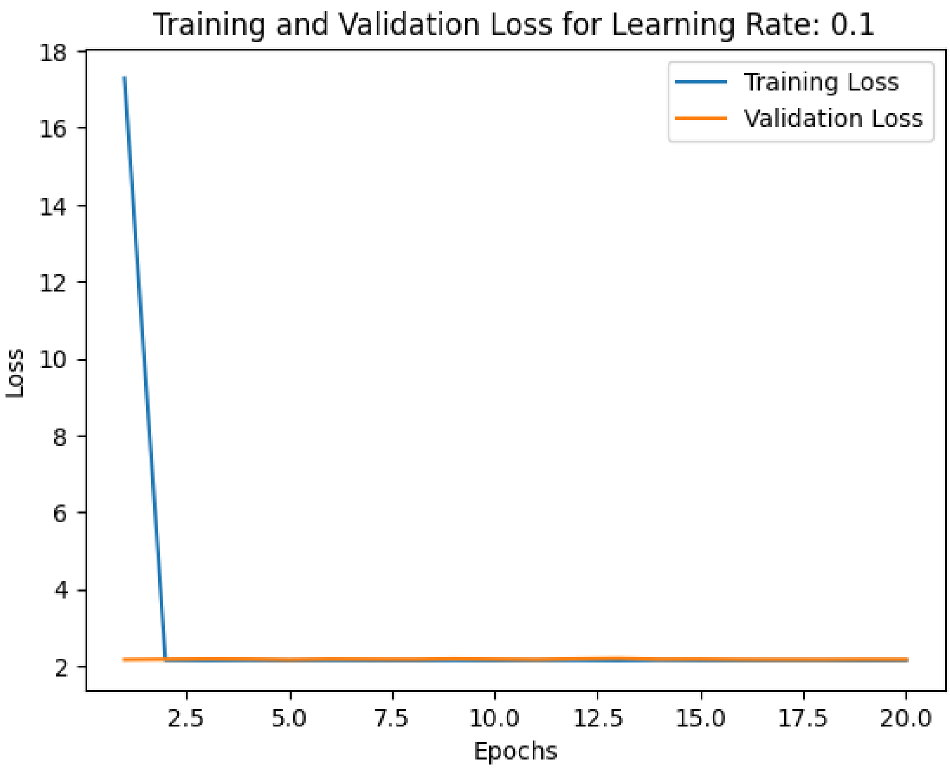

Figure 6: Learning Rate: 0.1

Comments on Results

- **Learning Rate = 0.0001**: Likely too slow; the loss decreases very gradually.
- **Learning Rate = 0.001**: Typically a good balance between speed and stability.
- **Learning Rate = 0.01**: Faster convergence but may start to oscillate.
- **Learning Rate = 0.1**: May diverge or show instability.

Based on the observations, **0.001** is selected as the optimal learning rate as it provides steady convergence of the validation loss without overfitting or oscillations.

2 Comapre the Network with state-of-the-art Networks

2.1 Dataset Overview

```
dataset_path = './realwaste/realwaste-main/RealWaste'

import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import DataLoader

num_classes = 9
batch_size = 32
learning_rate = 0.001
num_epochs = 20

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

transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

dataset = datasets.ImageFolder(root=dataset_path, transform=transform)
dataset_size = len(dataset)
indices = torch.randperm(dataset_size).tolist()

train_size = int(0.6 * dataset_size)
val_size = int(0.2 * dataset_size)
test_size = dataset_size - train_size - val_size

train_indices, val_indices, test_indices = indices[:train_size], indices[train_size:train_size+val_size], indices[train_size+val_size:]

train_dataset = torch.utils.data.Subset(dataset, train_indices)
val_dataset = torch.utils.data.Subset(dataset, val_indices)
test_dataset = torch.utils.data.Subset(dataset, test_indices)

# Print dataset information
print("Classes:", dataset.classes)
print(f"Total images: {len(dataset)}")
print(f"Training set size: {len(train_dataset)}")
print(f"Validation set size: {len(val_dataset)}")
print(f"Test set size: {len(test_dataset)}")
```

- **Classes:** Cardboard, Food Organics, Glass, Metal, Miscellaneous Trash, Paper, Plastic, Textile Trash, Vegetation

- **Total images:** 4752
- **Training set size:** 2851
- **Validation set size:** 950
- **Test set size:** 951

2.2 ResNet

2.2.1 Loading the pretrained model

```
model = models.resnet50(weights=models.ResNet50_Weights.IMAGENET1K_V1)

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

model.fc = nn.Linear(model.fc.in_features, num_classes)

nn.init.xavier_uniform_(model.fc.weight)
nn.init.zeros_(model.fc.bias)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=learning_rate)

print(f"Model is running on {device}")
```

Model is running on cuda

2.2.2 Fine-tuning the model

```
# Training and Validation
num_epochs = 20
train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []

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

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
```

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

            train_loss += loss.item() * inputs.size(0)
            _, predicted = outputs.max(1)
            correct_train += (predicted == labels).sum().item()
            total_train += labels.size(0)

        train_accuracy = correct_train / total_train
        train_loss /= total_train
        train_losses.append(train_loss)
        train_accuracies.append(train_accuracy)


        # Validation
        model.eval()
        val_loss = 0.0
        correct_val = 0
        total_val = 0

        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(
                    device)

                outputs = model(inputs)
                loss = criterion(outputs, labels)

                val_loss += loss.item() * inputs.size(0)
                _, predicted = outputs.max(1)
                correct_val += (predicted == labels).sum().item()
                total_val += labels.size(0)

        val_accuracy = correct_val / total_val
        val_loss /= total_val
        val_losses.append(val_loss)
        val_accuracies.append(val_accuracy)

        print(f"Epoch {epoch+1}/{num_epochs}, "
            f"Train Loss: {train_loss:.4f}, "
            f"Train Accuracy: {train_accuracy:.2f}%",
            f"Validation Loss: {val_loss:.4f}, "
            f"Validation Accuracy: {val_accuracy:.2f}%")
```

Results

Epoch 1/20, Train Loss: 1.4362, Train Accuracy: 0.49% Validation Loss: 1.0811, Validation Accuracy: 0.61%
Epoch 2/20, Train Loss: 0.9367, Train Accuracy: 0.68% Validation Loss: 1.0090, Validation Accuracy: 0.64%
Epoch 3/20, Train Loss: 0.8264, Train Accuracy: 0.71% Validation Loss: 0.8685, Validation Accuracy: 0.70%
Epoch 4/20, Train Loss: 0.7337, Train Accuracy: 0.75% Validation Loss: 0.8363, Validation Accuracy: 0.71%

Epoch 5/20, Train Loss: 0.6686, Train Accuracy: 0.76% Validation Loss: 0.7935, Validation Accuracy: 0.73%
Epoch 6/20, Train Loss: 0.6560, Train Accuracy: 0.77% Validation Loss: 0.7883, Validation Accuracy: 0.73%
Epoch 7/20, Train Loss: 0.6099, Train Accuracy: 0.79% Validation Loss: 0.7984, Validation Accuracy: 0.73%
Epoch 8/20, Train Loss: 0.5968, Train Accuracy: 0.79% Validation Loss: 0.8045, Validation Accuracy: 0.73%
Epoch 9/20, Train Loss: 0.5437, Train Accuracy: 0.82% Validation Loss: 0.8088, Validation Accuracy: 0.73%
Epoch 10/20, Train Loss: 0.5159, Train Accuracy: 0.83% Validation Loss: 0.8302, Validation Accuracy: 0.72%
Epoch 11/20, Train Loss: 0.5146, Train Accuracy: 0.83% Validation Loss: 0.7458, Validation Accuracy: 0.75%
Epoch 12/20, Train Loss: 0.5088, Train Accuracy: 0.82% Validation Loss: 0.8226, Validation Accuracy: 0.74%
Epoch 13/20, Train Loss: 0.4967, Train Accuracy: 0.82% Validation Loss: 0.8047, Validation Accuracy: 0.74%
Epoch 14/20, Train Loss: 0.4706, Train Accuracy: 0.84% Validation Loss: 0.7765, Validation Accuracy: 0.75%
Epoch 15/20, Train Loss: 0.4416, Train Accuracy: 0.85% Validation Loss: 0.8384, Validation Accuracy: 0.73%
Epoch 16/20, Train Loss: 0.4602, Train Accuracy: 0.85% Validation Loss: 0.7870, Validation Accuracy: 0.73%
Epoch 17/20, Train Loss: 0.4167, Train Accuracy: 0.85% Validation Loss: 0.7979, Validation Accuracy: 0.74%
Epoch 18/20, Train Loss: 0.4397, Train Accuracy: 0.85% Validation Loss: 0.8045, Validation Accuracy: 0.73%
Epoch 19/20, Train Loss: 0.4051, Train Accuracy: 0.86% Validation Loss: 0.7393, Validation Accuracy: 0.74%
Epoch 20/20, Train Loss: 0.3960, Train Accuracy: 0.87% Validation Loss: 0.7958, Validation Accuracy: 0.75%

2.2.3 Evaluating the model

```
from sklearn.metrics import confusion_matrix,
    ConfusionMatrixDisplay

all_labels = []
all_predictions = []

# Testing
model.eval()
test_loss = 0.0
correct_test = 0
total_test = 0

model.eval()
with torch.no_grad():
    for inputs, labels in test_loader:
```

```
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            test_loss += loss.item() * inputs.size(0)
            _, predicted = outputs.max(1)
            correct_test += (predicted == labels).sum().item()
            total_test += labels.size(0)
            all_labels.extend(labels.cpu().numpy())
            all_predictions.extend(predicted.cpu().numpy())
test_accuracy = correct_test / total_test
test_loss /= total_test
print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {
    test_accuracy:.4f}")
```

Test Loss: 0.8071, Test Accuracy: 0.7274

```
import matplotlib.pyplot as plt

# Plot training and validation loss
plt.figure(figsize=(10, 5))
plt.plot(range(1, num_epochs + 1), train_losses, label="
    Training Loss")
plt.plot(range(1, num_epochs + 1), val_losses, label="
    Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training and Validation Loss")
plt.legend()
plt.show()

# Plot training and validation accuracy
plt.figure(figsize=(10, 5))
plt.plot(range(1, num_epochs + 1), train_accuracies, label="
    Training Accuracy")
plt.plot(range(1, num_epochs + 1), val_accuracies, label="
    Validation Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Training and Validation Accuracy")
plt.legend()
plt.show()
```

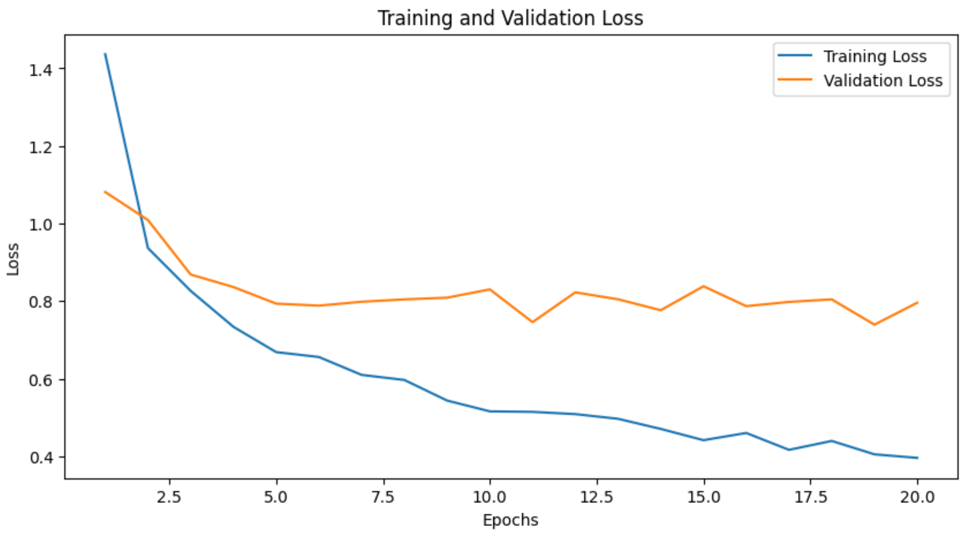

Figure 7: Traning and Validation Loss

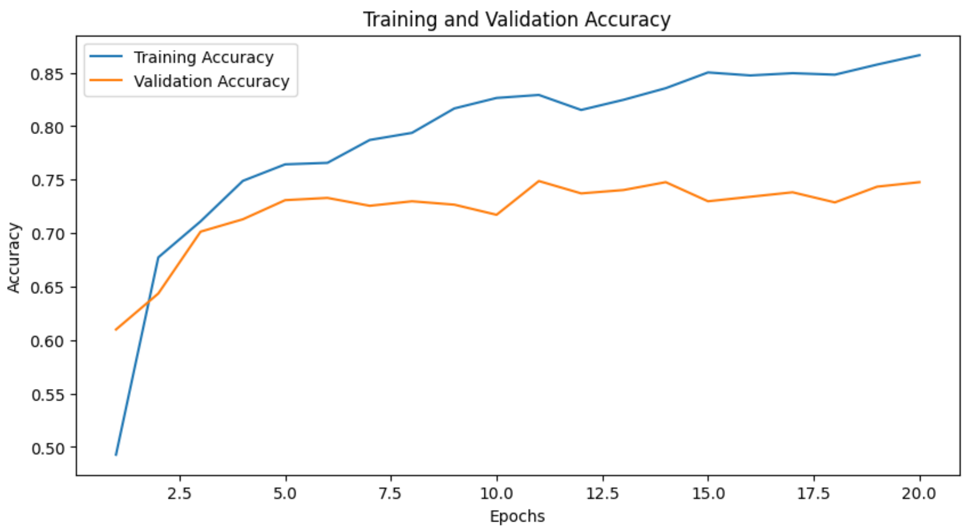

Figure 8: Training and Validation Accuracy

```python
from sklearn.metrics import confusion_matrix,
    ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Generate confusion matrix
cm = confusion_matrix(all_labels, all_predictions)

# Create and plot the confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
    display_labels=dataset.classes)
disp.plot(cmap='Blues', xticks_rotation=45)
plt.title('Confusion Matrix')
plt.show()
```

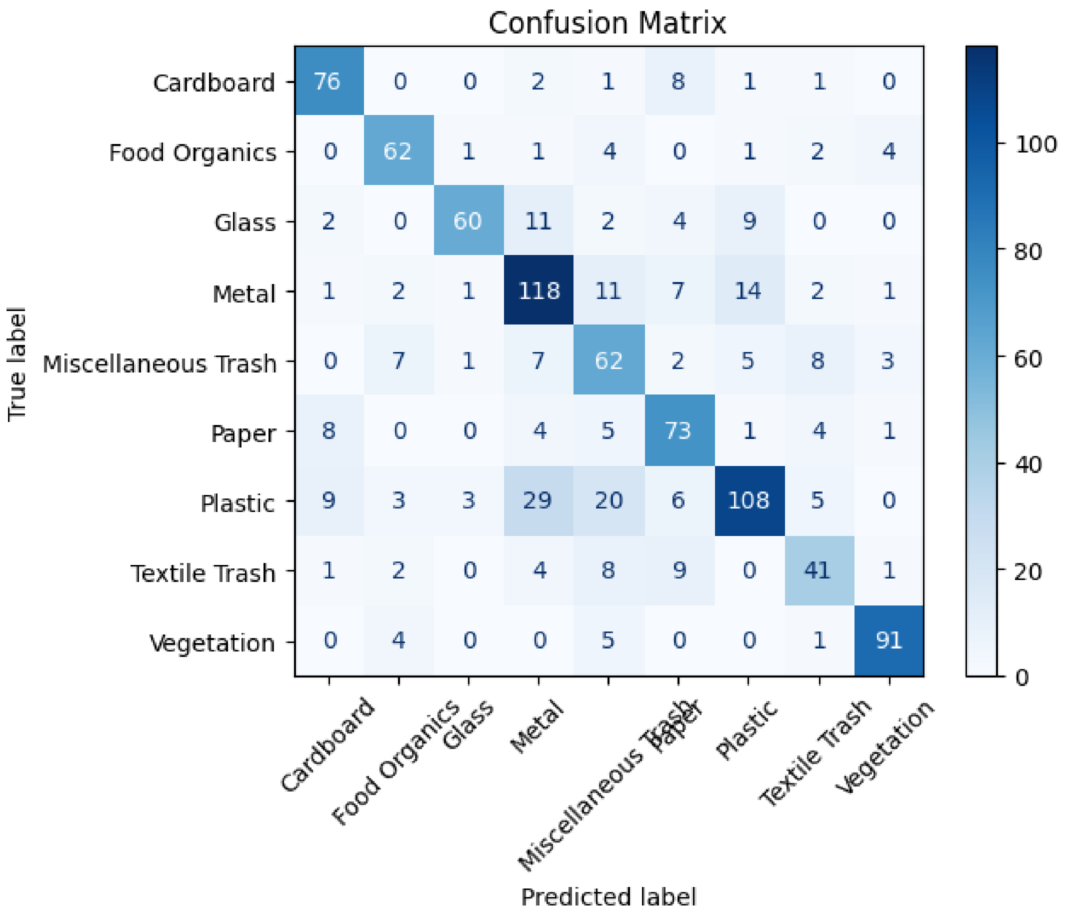

Figure 9: confusion matrix

2.3 DenseNet

2.3.1 Loading the pretrained model

```
model = models.densenet121(weights=models.DenseNet121_Weights.
    IMAGENET1K_V1)

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

num_features = model.classifier.in_features
model.classifier = nn.Linear(num_features, num_classes)

nn.init.xavier_uniform_(model.classifier.weight)
nn.init.zeros_(model.classifier.bias)

device = torch.device("cuda" if torch.cuda.is_available() else
     "cpu")
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.classifier.parameters(), lr=
    learning_rate)

print(f"Model is running on {device}")
```

Downloading: "https://download.pytorch.org/models/densenet121-a639ec97.pth" to /home/pasindupnk/.cache/torch/hub/checkpoints/densenet121-a639ec97.pth 100%—— 30.8M/30.8M [00:03¡00:00, 9.23MB/s] Model is running on cuda

2.3.2 Fine tuning the model

```
model = models.densenet121(weights=models.DenseNet121_Weights.
    IMAGENET1K_V1)

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

num_features = model.classifier.in_features
model.classifier = nn.Linear(num_features, num_classes)

nn.init.xavier_uniform_(model.classifier.weight)
nn.init.zeros_(model.classifier.bias)

device = torch.device("cuda" if torch.cuda.is_available() else
     "cpu")
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.classifier.parameters(), lr=
    learning_rate)

print(f"Model is running on {device}")
```

Epoch 1/20, Train Loss: 1.9297, Train Accuracy: 0.34, Validation Loss: 1.4091, Validation Accuracy: 0.49
Epoch 2/20, Train Loss: 1.1414, Train Accuracy: 0.59, Validation Loss: 1.0602, Validation Accuracy: 0.62
Epoch 3/20, Train Loss: 0.9154, Train Accuracy: 0.68, Validation Loss: 0.9550, Validation Accuracy: 0.67
Epoch 4/20, Train Loss: 0.7823, Train Accuracy: 0.73, Validation Loss: 0.8637, Validation Accuracy: 0.69
Epoch 5/20, Train Loss: 0.7062, Train Accuracy: 0.75, Validation Loss: 0.8227, Validation Accuracy: 0.71
Epoch 6/20, Train Loss: 0.6511, Train Accuracy: 0.77, Validation Loss: 0.8039, Validation Accuracy: 0.71
Epoch 7/20, Train Loss: 0.6012, Train Accuracy: 0.80, Validation Loss: 0.7795, Validation Accuracy: 0.72
Epoch 8/20, Train Loss: 0.5862, Train Accuracy: 0.80, Validation Loss: 0.8148, Validation Accuracy: 0.72
Epoch 9/20, Train Loss: 0.5587, Train Accuracy: 0.81, Validation Loss: 0.7501, Validation Accuracy: 0.75
Epoch 10/20, Train Loss: 0.5173, Train Accuracy: 0.82, Validation Loss: 0.7492, Validation Accuracy: 0.73
Epoch 11/20, Train Loss: 0.5029, Train Accuracy: 0.83, Validation Loss: 0.7497, Validation Accuracy: 0.73
Epoch 12/20, Train Loss: 0.4957, Train Accuracy: 0.84, Validation Loss: 0.7355, Validation Accuracy: 0.73
Epoch 13/20, Train Loss: 0.4605, Train Accuracy: 0.85, Validation Loss: 0.7160, Validation Accuracy: 0.75
Epoch 14/20, Train Loss: 0.4649, Train Accuracy: 0.84, Validation Loss: 0.7437, Validation Accuracy: 0.74
Epoch 15/20, Train Loss: 0.4401, Train Accuracy: 0.85, Validation Loss: 0.7675, Validation Accuracy: 0.73
Epoch 16/20, Train Loss: 0.4303, Train Accuracy: 0.85, Validation Loss: 0.7416, Validation Accuracy: 0.74
Epoch 17/20, Train Loss: 0.4217, Train Accuracy: 0.86, Validation Loss: 0.7365, Validation Accuracy: 0.74
Epoch 18/20, Train Loss: 0.4163, Train Accuracy: 0.86, Validation Loss: 0.7271, Validation Accuracy: 0.74
Epoch 19/20, Train Loss: 0.4251, Train Accuracy: 0.85, Validation Loss: 0.7258, Validation Accuracy: 0.75
Epoch 20/20, Train Loss: 0.4052, Train Accuracy: 0.86, Validation Loss: 0.7356, Validation Accuracy: 0.74

2.3.3 Evaluating the model

```
# Testing
all_labels, all_predictions = [], []
model.eval()
test_loss, correct_test, total_test = 0.0, 0, 0

with torch.no_grad():
    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        test_loss += loss.item() * inputs.size(0)
        _, predicted = outputs.max(1)
        correct_test += (predicted == labels).sum().item()
        total_test += labels.size(0)
        all_labels.extend(labels.cpu().numpy())
        all_predictions.extend(predicted.cpu().numpy())

test_accuracy = correct_test / total_test
test_loss /= total_test
print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {
    test_accuracy:.4f}")
```

Test Loss: 0.7478, Test Accuracy: 0.7442

```
model = models.resnet50(weights=models.ResNet50_Weights.
    IMAGENET1K_V1)

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

model.fc = nn.Linear(model.fc.in_features, num_classes)

nn.init.xavier_uniform_(model.fc.weight)
nn.init.zeros_(model.fc.bias)

device = torch.device("cuda" if torch.cuda.is_available() else
     "cpu")
model = model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.fc.parameters(), lr=learning_rate
    )

print(f"Model is running on {device}")
```

Model is running on cuda

```
# Plot training and validation loss
plt.figure(figsize=(10, 5))
plt.plot(range(1, num_epochs + 1), train_losses, label="
    Training Loss")
plt.plot(range(1, num_epochs + 1), val_losses, label="
    Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training and Validation Loss")
plt.legend()
plt.show()

# Plot training and validation accuracy
plt.figure(figsize=(10, 5))
plt.plot(range(1, num_epochs + 1), train_accuracies, label="
    Training Accuracy")
plt.plot(range(1, num_epochs + 1), val_accuracies, label="
    Validation Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Training and Validation Accuracy")
plt.legend()
plt.show()
```

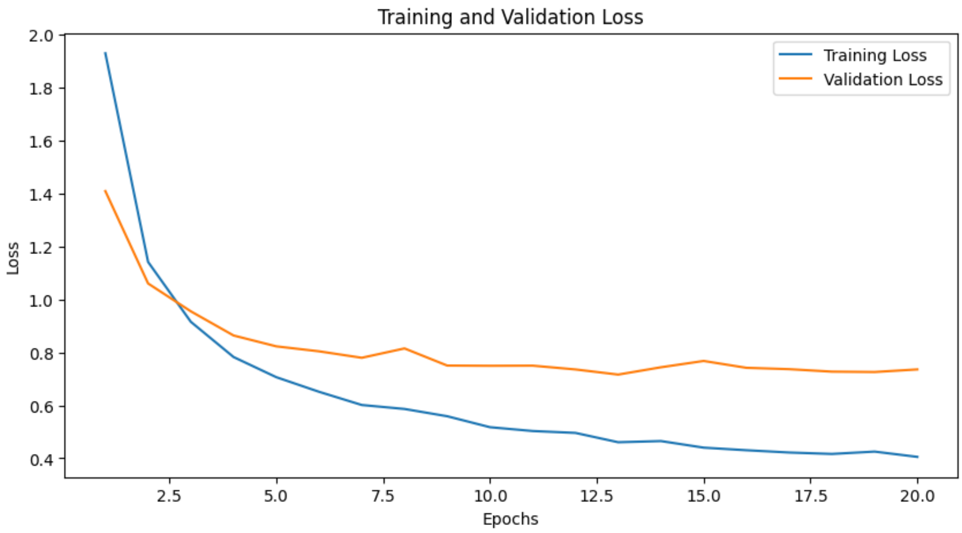

Figure 10: Training and Validation Loss

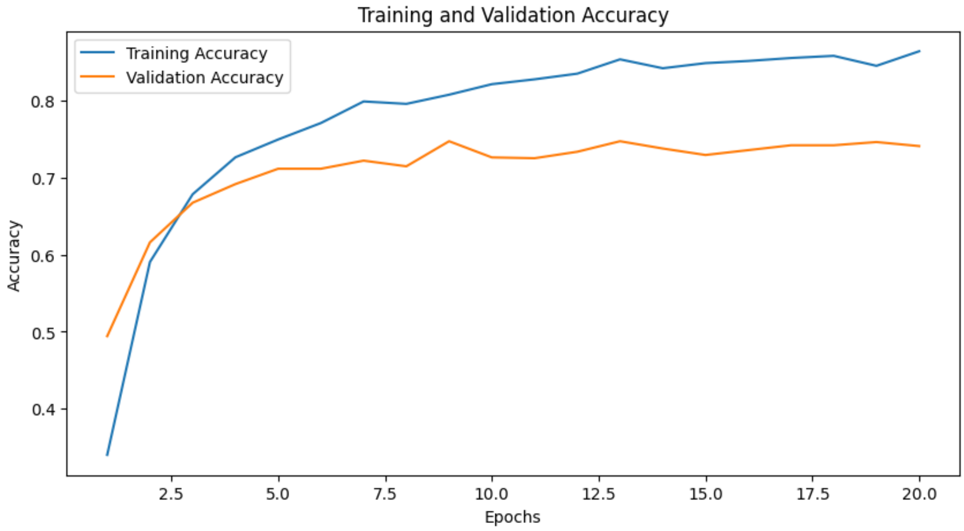

Figure 11: Training and Validation Accuracy

```
cm = confusion_matrix(all_labels, all_predictions)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
    display_labels=dataset.classes)
disp.plot(cmap='Blues', xticks_rotation=45)
plt.title('Confusion Matrix')
plt.show()
```

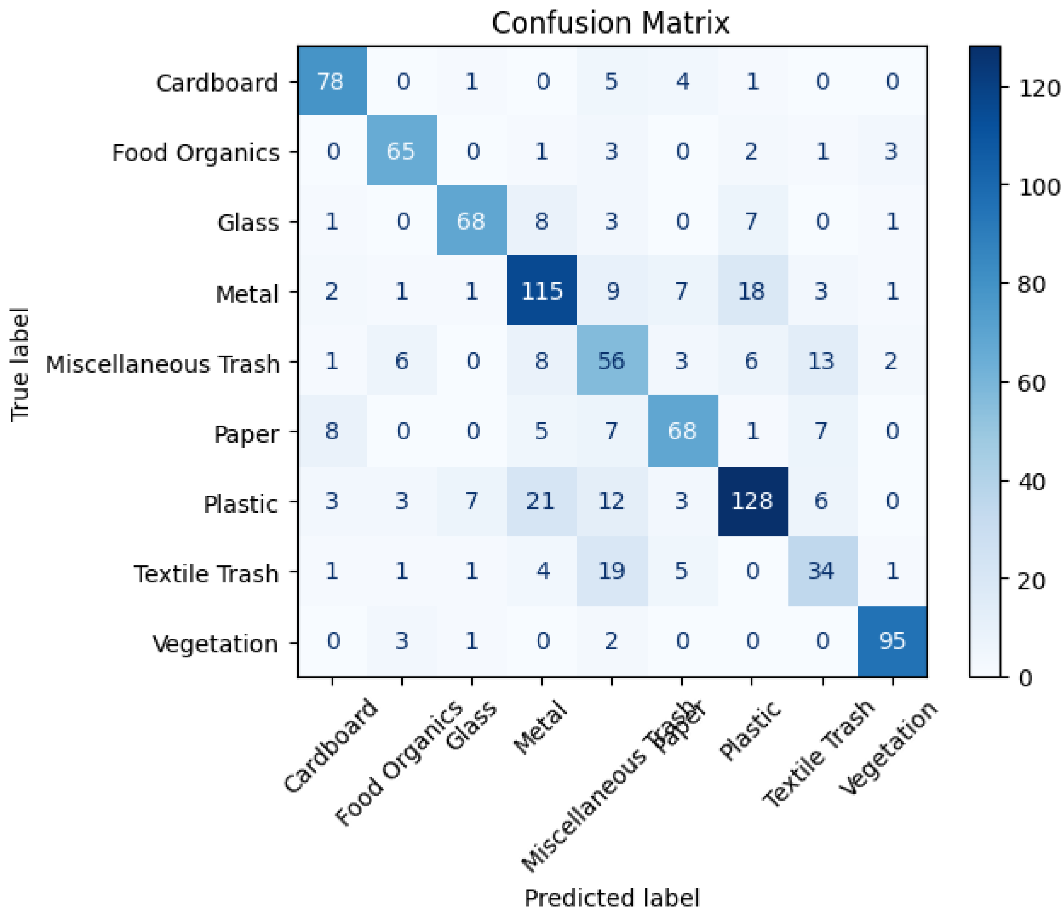

Figure 12: Confusion Matrix

2.4 Selecting pre-trained model or architecture

We have selected ResNet50 and DenseNet121, both well-known architectures pre-trained on ImageNet.

2.5 Fine-tuning the pre-trained model

The models are loaded using PyTorch's `torchvision.models` library, and their architectures are adapted using the `fine_tune_model` function to suit the Flowers-102 dataset. Specifically:

- **ResNet50**: The `fc` layer is modified to output the number of classes in the dataset.
- **DenseNet121**: The `classifier` layer is replaced to accommodate the new number of classes.

2.6 Train the fine-tuned model

The models are trained using the `train_model` function:

- **Optimizer**: Adam optimizer with a learning rate of 0.0001.
- **Learning Rate Scheduler**: Adjusts the learning rate every 7 epochs by a factor of 0.1.
- **Epochs**: 20.

Training splits (`train`, `val`, `test`) are organized into separate directories, and datasets are loaded using PyTorch's `ImageFolder` and `DataLoader` classes.

2.7 Training and validation loss values

Loss values are recorded for both training and validation splits and plotted using the plot_loss function.

2.8 Evaluate the fine-tuned model

The evaluate_model function computes the test accuracy using the testing split. It utilizes a simple loop to accumulate correct predictions and calculate the percentage accuracy.

2.9 Compare the test accuracy

To evaluate and compare the effectiveness of pre-trained models like ResNet or DenseNet with a custom CNN, several metrics should be considered, including accuracy, loss, confusion matrices, inference time, and generalization capabilities. Pre-trained models often excel due to their sophisticated architectures and transfer learning advantages, leveraging features learned from large datasets like ImageNet. If they outperform the custom CNN, it highlights their ability to adapt well to new tasks with limited data. Conversely, if the custom CNN performs comparably or better, it suggests that the model is well-designed and fine-tuned for the specific dataset and task. Visualization through accuracy and loss curves, along with confusion matrices, helps analyze classification performance and identify misclassification patterns. Ultimately, discussing factors like dataset size, complexity, and model parameters provides insights into why one approach outperforms the other, offering a deeper understanding of model behaviour.

2.10 Discussion trade-offs, advantages, and limitations

2.10.1 Custom Model

Advantages

- **Lightweight**: Custom models are smaller in size, which makes them faster to train and easier to deploy on resource-constrained devices.

- **Full Customizability**: They can be tailored to the specific requirements of the dataset and task, allowing for optimized performance when designed appropriately.
- **Simpler Architecture**: Easier to understand, debug, and modify compared to complex pre-trained models.

Limitations

- **Limited Generalization**: Custom models may underperform on complex datasets, particularly when data is scarce, as they lack prior knowledge from extensive pretraining.
- **Requires More Training Data**: Without access to pre-trained weights, custom models need significantly larger datasets to learn high-level features effectively.
- **Shallow Learning**: May struggle with extracting intricate patterns due to limited depth and absence of pretraining.

2.10.2 Pretrained Model

Advantages

- **Faster Convergence**: pretrained models start with learned weights, enabling them to recognize general features like edges and shapes, reducing training time.
- **Better Generalization**: Leveraging knowledge from large datasets (e.g., ImageNet) improves performance on smaller datasets, particularly when data is limited.
- **Robustness**: pretrained weights act as a form of regularization, mitigating overfitting in cases of limited data.
- **Complex Pattern Recognition**: Handles intricate patterns and domain-specific features more effectively than custom models.

Limitations

- **Computational Cost**: Fine-tuning pretrained models is computationally intensive and requires more resources for both training and inference.
- **Domain Gap Challenges**: When the target dataset differs significantly from the source dataset (e.g., ImageNet vs. medical imaging), adaptation becomes challenging.
- **Hyperparameter Sensitivity**: Fine-tuning requires careful adjustment of hyperparameters, such as learning rate, to avoid overfitting or underfitting.
- **Larger Model Size**: pre-trained models consume more memory and storage, making them less suitable for lightweight applications.

Trade-offs and Comparison

- **Performance vs. Complexity**: pre-trained models generally outperform custom models due to their depth and pretraining, but they come with increased computational costs and complexity.
- **Flexibility vs. Generalization**: Custom models are easier to customize and may be ideal for specific, simple tasks, while pre-trained models are better suited for complex or high-stakes tasks requiring robust generalization.
- **Training Data Requirements**: Custom models need substantial labelled data to perform well, whereas pre-trained models thrive even with limited data through transfer learning.

Pretrained models are typically the better choice for tasks with limited data or complex patterns, as they offer faster convergence, higher accuracy, and better generalization. However, custom models are advantageous when computational resources are constrained, the task is straightforward, or there's a need for a lightweight and tailored solution. The choice depends on the dataset size, task complexity, and resource availability.

Appendix

You can find the project repository on GitHub:
Image Classification using CNN