Department of Electronic & Telecommunication Engineering

University of Moratuwa

EN3150 - Pattern Recognition

Assignment 03: Simple convolutional neural network to perform classification

Group: Perceptrons

HERATH B.H.M.K.S.B.	210212N
KURUPPU M.P.	210325M
PEIRIS D.L.C.J.	210454G
WEERASINGHE M.G.I.U.	210689F

Date - 2024.12.11

Contents

Visit Our GitHub Repository Here:
simple convolutional neural network to perform classification

1 CNN for image classification

Here we have used real waste dataset from UCI Machine Learning repository.

1.1 Analyzing the Dataset

Categories in the dataset

```python
import os
def count_images_in_folders(root_path):
    """
    Count the number of image files in each subfolder under the given root directory.
    Parameters:
    - root_path: The directory containing class subfolders.
    Returns:
    - A dictionary with folder names as keys and image counts as values.
    """
    folder_counts = {}
    # Iterate through each subfolder
    for folder in os.listdir(root_path):
        folder_path = os.path.join(root_path, folder)
        # Check if it's a directory
        if os.path.isdir(folder_path):
            # Count files with image extensions
            count = sum(
                1 for file in os.listdir(folder_path)
                if file.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.gif'))
            )
            folder_counts[folder] = count
    return folder_counts
if __name__ == "__main__":
    root_path = "/content/dataset/realwaste-main/RealWaste"
    counts = count_images_in_folders(root_path)
    for folder, count in counts.items():
        print(f"{folder}: {count} images")
```

```
Plastic: 921 images
Metal: 790 images
Miscellaneous Trash: 495 images
Vegetation: 436 images
Food Organics: 411 images
Paper: 500 images
Cardboard: 461 images
Glass: 420 images
Textile Trash: 318 images
```

Figure 1: Categories of the dataset

Dataset Summary

```python
import os
from PIL import Image
def inspect_dataset(dataset_path):
    stats = {
        "total_images": 0,
        "image_sizes": set(),  # Unique image dimensions (width, height)
        "file_types": set(),   # Unique file extensions
        "color_modes": set(),  # Unique color modes (e.g., RGB, Grayscale)
    }
    # Traverse dataset directory
    for root, dirs, files in os.walk(dataset_path):
        for file_name in files:
            file_path = os.path.join(root, file_name)
            try:
                with Image.open(file_path) as img:
                    stats["total_images"] += 1
                    stats["image_sizes"].add(img.size)  # Add image dimensions
                    stats["file_types"].add(file_name.split('.')[-1].lower())  # Add file extension
                    stats["color_modes"].add(img.mode)  # Add color mode (e.g., RGB, L)
            except Exception as e:
                print(f"Error loading image: {file_path}, {e}")
    return stats
dataset_path = "/content/dataset/realwaste-main/RealWaste"  # Base directory
stats = inspect_dataset(dataset_path)
print("Total Images:", stats["total_images"])
print("Unique Image Sizes:", stats["image_sizes"])
print("File Types:", stats["file_types"])
print("Color Modes:", stats["color_modes"])
```

```
Total Images: 4752
Unique Image Sizes: {(524, 524)}
File Types: {'jpg'}
Color Modes: {'RGB'}
```

Figure 2: Dataset summary

The dataset consists of 4,752 images, all of which share a consistent size of 524x524 pixels. The images are in JPG format and use the RGB color mode.

Mean and Standard deviation of Images from the Dataset

```python
import os
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
def compute_mean_std(dataset_path):
    means = np.zeros(3)  # For RGB channels (R, G, B)
    stds = np.zeros(3)
    total_pixels = 0
    for root, dirs, files in os.walk(dataset_path):
        for file_name in files:
            file_path = os.path.join(root, file_name)
            try:
                with Image.open(file_path) as img:
                    img = img.convert("RGB")  # Ensure all images are RGB
                    img_array = np.array(img)  # Convert to NumPy array
                    img_array = img_array / 255.0  # Normalize to [0, 1]

                    # Compute stats for the current image
                    pixels = img_array.shape[0] * img_array.shape[1]
                    total_pixels += pixels

                    means += img_array.reshape(-1, 3).sum(axis=0)  # Sum over color channels
                    stds += (img_array.reshape(-1, 3) ** 2).sum(axis=0)
            except Exception as e:
                print(f"Error processing image {file_path}: {e}")
    # Compute final means and standard deviations
    means /= total_pixels
    stds = np.sqrt(stds / total_pixels - means**2)  # Variance = E[X^2] - (E[X])^2
    return means, stds
def plot_gaussian_distributions(mean, std):
```

```
    mean = list(mean)
    std = list(std)
    colors = ['red', 'green', 'blue']  # Colors for the RGB distributions
    labels = ['Red Channel', 'Green Channel', 'Blue Channel']
    # Generate x values for the Gaussian distributions
    x = np.linspace(0, 255, 500)  # Pixel intensity range
    plt.figure(figsize=(10, 6))
    # Plot each Gaussian distribution
    for mean_val, std_val, color, label in zip(mean, std, colors, labels):
        y = (1 / (std_val * np.sqrt(2 * np.pi))) * np.exp(-0.5 * ((x - mean_val * 255) / (std_val * 255)) ** 2)
        plt.plot(x, y, color=color, label=label)
    # Add plot details
    plt.title('Gaussian Distributions for RGB Channels', fontsize=14)
    plt.xlabel('Pixel Intensity', fontsize=12)
    plt.ylabel('Density', fontsize=12)
    plt.legend()
    plt.grid(alpha=0.3)
    plt.tight_layout()
    plt.show()
# Example usage
dataset_path = "/content/dataset/realwaste-main/RealWaste"  # Replace with your dataset path
mean, std = compute_mean_std(dataset_path)
print("Mean (R, G, B):", mean)
print("Standard Deviation (R, G, B):", std)
# Plot the Gaussian distributions
plot_gaussian_distributions(mean, std)
```

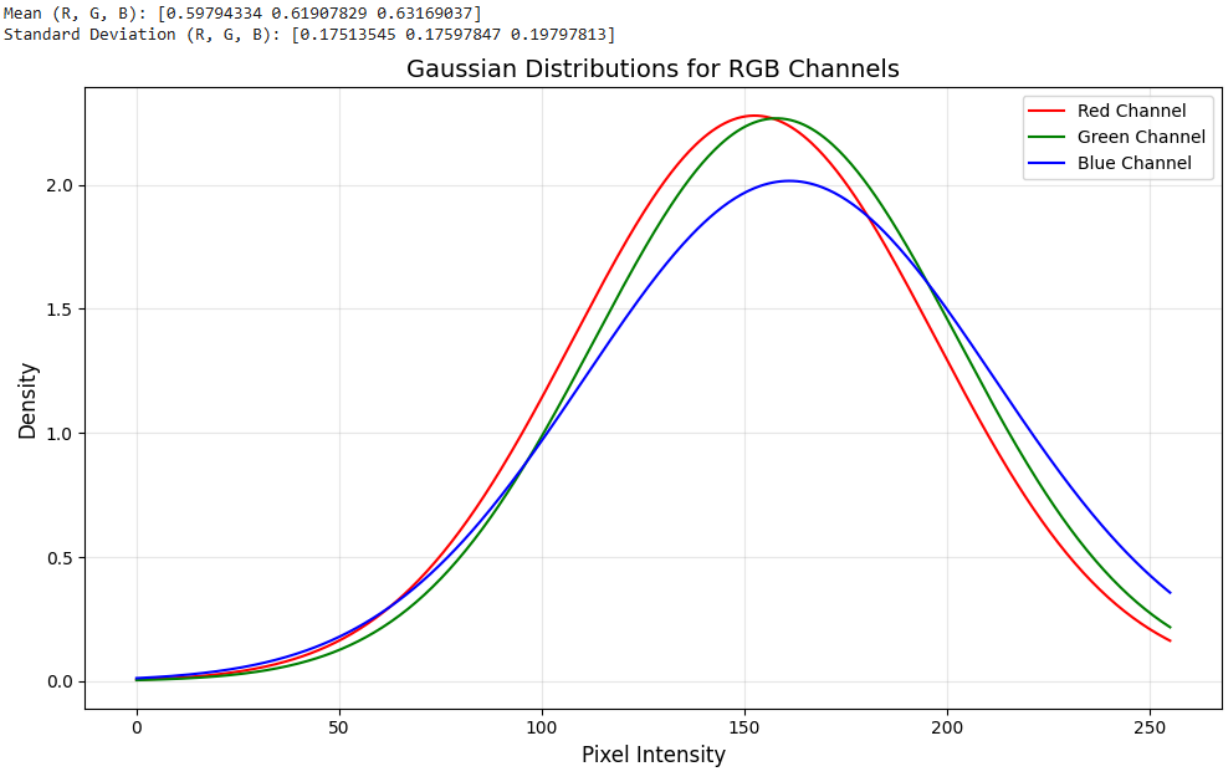

Figure 3: Mean and Std of the dataset

1.2 Preprocessing: Image Normalization

We have normalised the image tensor using the calculated mean and standard deviation for each channel (R, G, B). The normalised pixel value is computed as follows:

$$\text{Normalized Pixel Value} = \frac{\text{Pixel Value} - \text{Mean}}{\text{Standard Deviation}}$$

This normalisation achieves the following benefits:

1. **Centering the Data:** Ensures pixel values are centered around zero, reducing bias due to variations in brightness and contrast.
2. **Improving Convergence:** Scales input data consistently, facilitating faster and more stable model training.
3. **Reducing Sensitivity to Illumination Variations:** Makes the model robust to differences in lighting conditions across the dataset.
4. **Improving Numerical Stability:** Prevents extreme values in activation layers, addressing gradient vanishing or exploding issues.
5. **Aligning with Pretrained Models:** Matches the normalization scheme used during the training of pretrained models like ResNet or VGG for better compatibility.

```
# Function to display original and preprocessed images
def visualize_random_images(dataset, num_images=4):
    fig, axes = plt.subplots(num_images, 2, figsize=(10,
       num_images * 5))
    axes = axes.flatten()
    # Select random indices from the dataset
    random_indices = random.sample(range(len(dataset)),
       num_images)
    for i, idx in enumerate(random_indices):
        img, label = dataset[idx]
        # Get the original image path and load the original image
        original_img_path = dataset.image_paths[idx]
        original_img = Image.open(original_img_path).convert("RGB
           ")
        # Plot original image
        axes[i * 2].imshow(original_img)
        axes[i * 2].set_title(f"Original: {dataset.classes[label
           ]}")
        axes[i * 2].axis('off')
        # Plot preprocessed image (transformed)
        axes[i * 2 + 1].imshow(img.permute(1, 2, 0))  # Convert
           from (C, H, W) to (H, W, C)
        axes[i * 2 + 1].set_title(f"Preprocessed: {dataset.
           classes[label]}")
        axes[i * 2 + 1].axis('off')
    plt.tight_layout()
    plt.show()
# Example usage
visualize_random_images(dataset)
```

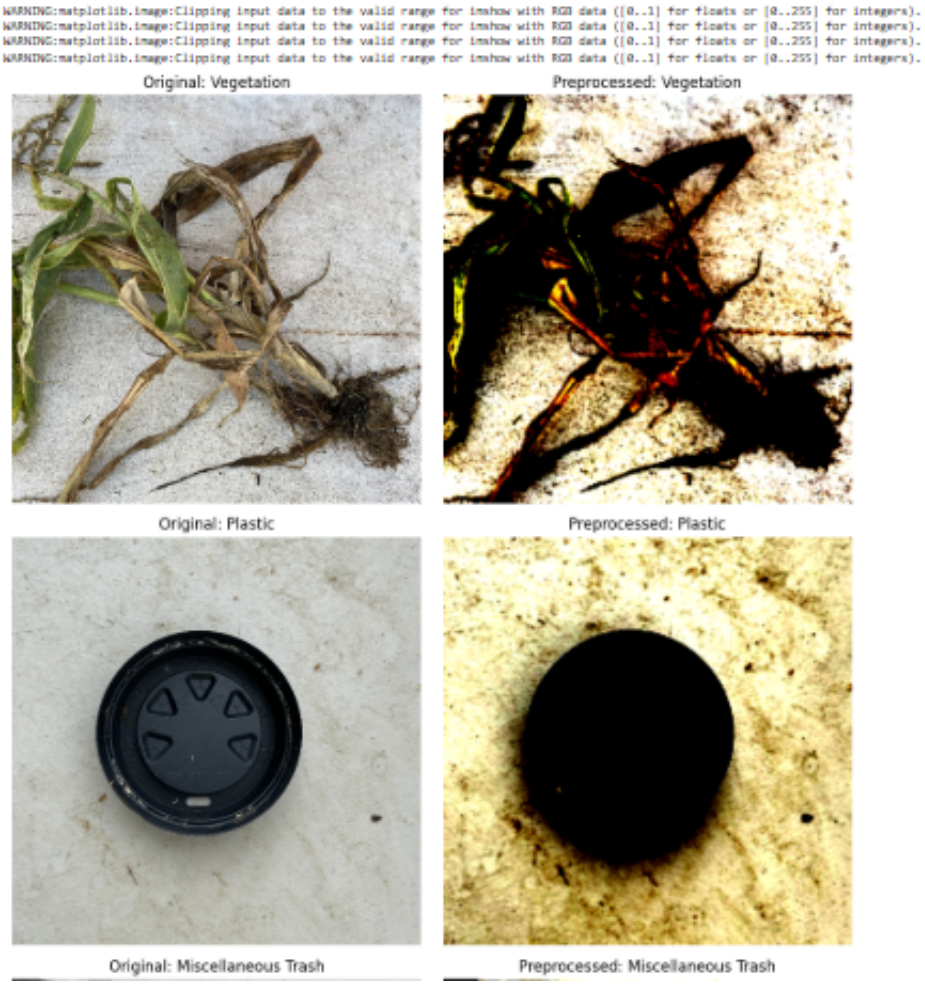

Figure 4: Mean and Std of the dataset

1.3 Train Validation Test split of the Dataset

```python
import os
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import transforms
from PIL import Image
class WasteDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        """
        Args:
            root_dir (str): Directory with all the images,
               organized by subfolders for each class.
            transform (callable, optional): Optional transform to
                be applied on an image.
        """
        self.root_dir = root_dir
        self.transform = transform
```

```
        self.classes = sorted(os.listdir(root_dir))
        self.image_paths = []
        self.labels = []
        # Collect image paths and labels
        for label_idx, class_name in enumerate(self.classes):
            class_dir = os.path.join(root_dir, class_name)
            for img_file in os.listdir(class_dir):
                self.image_paths.append(os.path.join(class_dir,
                    img_file))
                self.labels.append(label_idx)
    def __len__(self):
        return len(self.image_paths)
    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        label = self.labels[idx]
        with Image.open(img_path) as img:
            img = img.convert("RGB")  # Ensure RGB format
            if self.transform:
                img = self.transform(img)
        return img, label
# Define dataset paths
dataset_path = "/content/dataset/realwaste-main/RealWaste"
# Define transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize images to a fixed
        size
    transforms.ToTensor(),          # Convert image to PyTorch
        tensor and normalize to [0, 1]
    transforms.Normalize(mean, std) # Normalize using calculated
        mean and std ])
# Create the dataset
dataset = WasteDataset(dataset_path, transform=transform)
# Calculate split sizes
dataset_size = len(dataset)
train_size = int(0.6 * dataset_size)
val_size = int(0.2 * dataset_size)
test_size = dataset_size - train_size - val_size
# Split the dataset
train_dataset, val_dataset, test_dataset = random_split(dataset,
    [train_size, val_size, test_size])
# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=
    True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False
    )
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=
    False)
```

```
# Print dataset sizes
print(f"Total dataset size: {dataset_size}")
print(f"Training set size: {len(train_dataset)}")
print(f"Validation set size: {len(val_dataset)}")
print(f"Test set size: {len(test_dataset)}")
```

```
Total dataset size: 4752
Training set size: 2851
Validation set size: 950
Test set size: 951
```

Figure 5: Training, validation, and testing subsets

1.4 Building the CNN model

EnhancedCNN model

```
import torch.nn as nn
import torch.nn.functional as F
class EnhancedCNN(nn.Module):
    def __init__(self, num_classes):
        super(EnhancedCNN, self).__init__()
        # Convolutional layers
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
           # Input: (3, 224, 224) -> Output: (32, 224, 224)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
           # Input: (32, 224, 224) -> Output: (64, 112, 112)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
           # Pooling: -> Output: (64, 112, 112)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
            # Input: (64, 112, 112) -> Output: (128, 112, 112)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding
           =1)# Input: (128, 112, 112) -> Output: (256, 56, 56)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
                  # Pooling: -> Output: (256, 56, 56)
        self.conv5 = nn.Conv2d(256, 512, kernel_size=3, padding
           =1) # Input: (256, 56, 56) -> Output: (512, 56, 56)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)
                   # Pooling: -> Output: (512, 28, 28)
        # Flatten layer
        self.flatten = nn.Flatten() # Output: 512 * 28 * 28 =
           401408
        # Fully connected layers
        self.fc1 = nn.Linear(512 * 28 * 28, 1024)
        self.fc2 = nn.Linear(1024, 512)
        self.fc3 = nn.Linear(512, 256)
        self.dropout = nn.Dropout(0.5)  # Dropout for
           regularization
        self.fc4 = nn.Linear(256, num_classes)  # Output layer
    def forward(self, x):
        # Convolutional and pooling layers
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = self.pool1(x)
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = self.pool2(x)
        x = F.relu(self.conv5(x))
        x = self.pool3(x)
        # Fully connected layers
        x = self.flatten(x)
```

```
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        return x
# Initialize model
num_classes = 9  # Number of classes
model = EnhancedCNN(num_classes)
print(model)
```

Enhanced CNN Improved model

To improve accuracy, we enhanced the previous model by introducing the following key features:

1. **Batch Normalization:** Incorporated after each convolutional layer to stabilize learning and accelerate convergence.
2. **Global Average Pooling (GAP):** Replaced the flattening operation, reducing parameters while retaining spatial information.
3. **Simplified Fully Connected Layers:** Reduced the number and complexity of fully connected layers, enhancing generalization and minimizing overfitting.

These modifications resulted in a more efficient and accurate model.

```
class EnhancedCNNImproved(nn.Module):
    def __init__(self, num_classes):
        super(EnhancedCNNImproved, self).__init__()

        # Convolutional layers with BatchNorm
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.conv4 = nn.Conv2d(128, 256, kernel_size=3, padding
           =1)
        self.bn4 = nn.BatchNorm2d(256)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv5 = nn.Conv2d(256, 512, kernel_size=3, padding
           =1)
        self.bn5 = nn.BatchNorm2d(512)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Global Average Pooling
        self.global_pool = nn.AdaptiveAvgPool2d(1)

        # Fully connected layers
        self.fc1 = nn.Linear(512, 256)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(256, num_classes)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
```

```
        x = F.relu(self.bn2(self.conv2(x)))
        x = self.pool1(x)

        x = F.relu(self.bn3(self.conv3(x)))
        x = F.relu(self.bn4(self.conv4(x)))
        x = self.pool2(x)

        x = F.relu(self.bn5(self.conv5(x)))
        x = self.pool3(x)

        x = self.global_pool(x)
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return x

num_classes = 9  # Number of classes
model = EnhancedCNNImproved(num_classes)
print(model)
```

```
EnhancedCNNImproved(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv5): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn5): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (global_pool): AdaptiveAvgPool2d(output_size=1)
  (fc1): Linear(in_features=512, out_features=256, bias=True)
  (dropout): Dropout(p=0.5, inplace=False)
  (fc2): Linear(in_features=256, out_features=9, bias=True)
)
```

Figure 6: CNN Architecture

1.5 Parameters of the above CNN Model

Convolutional Layers

conv1: **Input Channels:** 3
Output Channels: 32
Kernel Size: 3×3
Stride: 1
Padding: 1

conv2: **Input Channels:** 32
Output Channels: 64
Kernel Size: 3×3
Stride: 1
Padding: 1

conv3: **Input Channels:** 64
Output Channels: 128
Kernel Size: 3×3
Stride: 1
Padding: 1

conv4: **Input Channels:** 128
Output Channels: 256
Kernel Size: 3×3
Stride: 1
Padding: 1

conv5: **Input Channels:** 256
Output Channels: 512
Kernel Size: 3×3
Stride: 1
Padding: 1

Batch Normalization Layers

- **bn1:** Applied after **conv1** with 32 channels.
- **bn2:** Applied after **conv2** with 64 channels.
- **bn3:** Applied after **conv3** with 128 channels.
- **bn4:** Applied after **conv4** with 256 channels.
- **bn5:** Applied after **conv5** with 512 channels.

Pooling Layers

pool1: **Type:** MaxPooling
Kernel Size: 2×2
Stride: 2
Padding: 0

pool2: **Type:** MaxPooling
Kernel Size: 2×2
Stride: 2
Padding: 0

pool3: **Type:** MaxPooling
Kernel Size: 2×2
Stride: 2
Padding: 0

Global Average Pooling

- **global_pool:** Adaptive Average Pooling with Output Size 1×1.

Fully Connected Layers

1. **fc1:**
   Input Features: 512
   Output Features: 256

2. **fc2:**
   Input Features: 256
   Output Features: 9

Activation Functions

- **ReLU:** Used after every convolutional and fully connected layer.
- **Final Layer:** The output layer produces logits, and a softmax activation is applied externally during training.

No need to manually apply softmax:

nn.CrossEntropyLoss expects raw logits as inputs (i.e., outputs from the model before any activation function like softmax is applied).

Softmax is applied internally:

During training, CrossEntropyLoss applies the log-softmax and calculates the negative log-likelihood loss in one step, which is more efficient than applying softmax manually.

Dropout Layer

- **dropout:**
  Dropout Rate: 0.5

1.6 Activation Functions

Activation Functions Explanation

1. ReLU Activation in Middle Layers

Purpose: ReLU (Rectified Linear Unit) is widely used in the middle layers of neural networks because it helps with non-linearity while being computationally efficient.

Why ReLU?

- **Non-linearity:** ReLU introduces non-linearity into the network, which allows it to learn complex relationships in the data. Without non-linear activation functions, the network would behave like a linear model, limiting its ability to approximate complex functions.
- **Efficiency:** ReLU is computationally simple, as it only involves a thresholding operation (outputting 0 for negative values and leaving positive values unchanged). This simplicity helps speed up training compared to other activation functions like Sigmoid or Tanh.
- **Avoiding Vanishing Gradient Problem:** Compared to Sigmoid or Tanh, ReLU does not saturate for positive input values, which reduces the risk of vanishing gradients during backpropagation and allows the network to learn more effectively.

2. Softmax Activation in the Final Layer

Purpose: Softmax is used in the final layer when the network is performing a classification task, especially for multi-class classification problems.

Why Softmax?

- **Probability Distribution:** Softmax converts the raw output scores (logits) from the final layer into a probability distribution. Each class will have a probability between 0 and 1, and the sum of all probabilities will be 1. This makes it suitable for classification tasks where we want to interpret the outputs as probabilities of different classes.
- **Multi-class Classification:** Softmax is particularly useful for multi-class classification because it normalizes the output into probabilities, ensuring that the predicted class has the highest probability. This makes it easy to select the most likely class for a given input.
- **Interpretability:** By applying Softmax, the network's output becomes more interpretable because it gives the likelihood of each class being the correct one, which is useful in decision-making, especially when confidence levels are important.

1.7 Training the Model

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from tqdm import tqdm
# Assuming the DataLoader and model setup is already complete
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Hyperparameters
num_epochs = 20
learning_rate = 0.001
# Define model, criterion, and optimizer
model = EnhancedCNNImproved(num_classes=len(dataset.classes)).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
# Initialize variables to track training
best_val_loss = float('inf')
best_model_path = "best_model.pth"
train_losses = []
val_losses = []
# Training loop
for epoch in range(num_epochs):
    # Training phase
    model.train()
    train_loss = 0.0
    for images, labels in tqdm(train_loader, desc=f"Epoch {epoch+1}/{num_epochs} [Training]"):
        images, labels = images.to(device), labels.to(device)
        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)
        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
    train_loss /= len(train_loader)
    train_losses.append(train_loss)
    # Validation phase
    model.eval()
    val_loss = 0.0
    with torch.no_grad():
        for images, labels in tqdm(val_loader, desc=f"Epoch {epoch+1}/{num_epochs} [Validation]"):
            images, labels = images.to(device), labels.to(device)
```

```
            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
    val_loss /= len(val_loader)
    val_losses.append(val_loss)
    # Save the best model
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        torch.save(model.state_dict(), best_model_path)
        print(f"Epoch {epoch+1}: Best model saved with validation
            loss: {val_loss:.4f}")
    # Print losses
    print(f"Epoch [{epoch+1}/{num_epochs}], Train Loss: {
       train_loss:.4f}, Val Loss: {val_loss:.4f}")
# After training, load the best model
model.load_state_dict(torch.load(best_model_path))
print("Best model loaded with validation loss:", best_val_loss)
# Plot training and validation loss
plt.figure(figsize=(10, 6))
plt.plot(range(1, num_epochs + 1), train_losses, label="Train
   Loss", marker='o')
plt.plot(range(1, num_epochs + 1), val_losses, label="Validation
   Loss", marker='o')
plt.title("Training and Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()
```

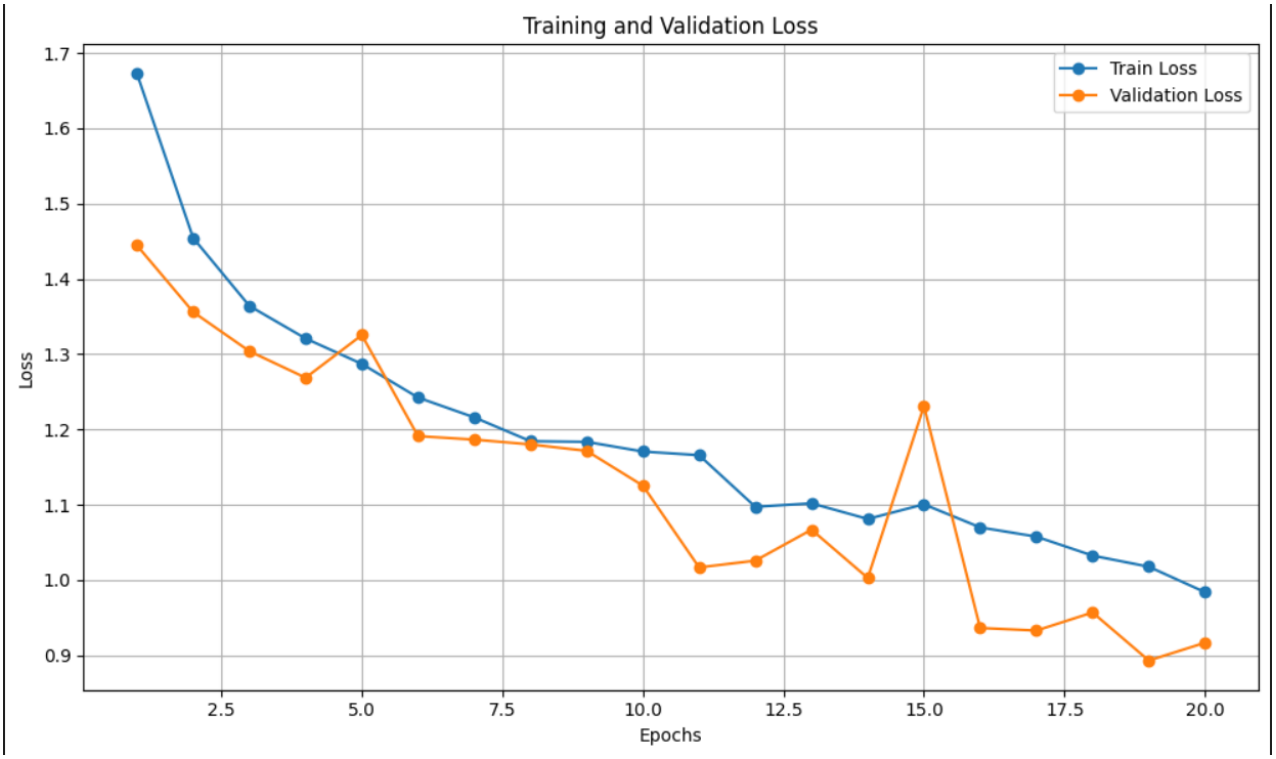

Figure 7: Training and Validation Loss

1.8 Why Adam Optimizer over SGD

1. Adaptive Learning Rate

Adam (Adaptive Moment Estimation) adjusts the learning rate for each parameter individually based on the first and second moments (mean and variance) of the gradients. This means it can adapt the learning rate throughout training, whereas **SGD** uses a fixed learning rate for all parameters, which may require manual tuning or result in slow convergence.

2. Handling Sparse Gradients

Adam is particularly effective when working with sparse gradients, as it adjusts the learning rate based on the magnitude of the gradients. **SGD** can struggle in scenarios where gradients are sparse, especially for complex models like deep neural networks.

3. Faster Convergence

Due to the adaptive learning rates and moment estimation, **Adam** often leads to faster convergence compared to **SGD**. It requires less manual tuning of the learning rate and can adaptively make progress during training. **SGD** may need momentum or learning rate schedules to achieve similar results.

4. Robustness to Hyperparameter Choices

Adam is more robust to the choice of hyperparameters, particularly the learning rate. **SGD** is more sensitive to the learning rate and often requires fine-tuning to find the best value, which can make training more time-consuming.

5. Momentum and Adaptive Updates

Adam combines the advantages of both **Momentum** and **RMSProp**:

- **Momentum:** Helps accelerate gradients in the correct direction, which speeds up convergence.
- **RMSProp:** Adjusts the learning rate based on the moving average of the squared gradients, which stabilizes updates and prevents large updates when gradients are large.

In contrast, **SGD** (even with momentum) does not adapt the learning rate based on the second moment of the gradients, which can sometimes slow down learning in certain situations.

6. Better Generalization

While Adam is generally better for fast convergence, it is observed that **SGD** often performs better for generalization when used with learning rate schedules and when fine-tuned properly. This makes **SGD** a better choice in certain settings, particularly for large datasets or when the goal is to achieve high performance on unseen data.

1.9 Why Sparse Categorical Crossentropy as the Loss function

1. Multi-class Classification with Integer Labels

Sparse Categorical Crossentropy is typically used in multi-class classification tasks where the labels are represented as integers (e.g., class 0, class 1, class 2, etc.) rather than one-hot encoded vectors. This loss function is ideal when the output classes are mutually exclusive and can be represented by a single integer label.

2. Simplicity and Efficiency

When the labels are integers (as opposed to one-hot encoded vectors), using **Sparse Categorical Crossentropy** eliminates the need to manually convert them into one-hot encoding, saving both memory and computational resources. This makes it more efficient when working with large datasets or models where encoding labels might be cumbersome.

3. Direct Comparison with Logits

Sparse Categorical Crossentropy directly compares the predicted logits (raw model outputs before applying softmax) with the integer labels. This allows for the calculation of the negative log-likelihood loss, which is particularly useful for classification problems where each class corresponds to a unique label.

4. Avoiding One-Hot Encoding Overhead

One-hot encoding requires creating a vector as long as the number of classes for each sample, which can lead to unnecessary computational overhead, especially with a large number of classes. Sparse Categorical Crossentropy avoids this by treating the labels as integers, directly comparing them to the predicted logits.

5. Use Case in Neural Networks

Sparse Categorical Crossentropy is widely used in classification tasks for neural networks where each input is assigned to one class from a set of mutually exclusive classes. The loss function ensures that the model is penalized based on the probability assigned to the correct class, which improves the model's performance in predicting the right label.

1.10 Evaluating the Model

```
import torch
import numpy as np
from sklearn.metrics import confusion_matrix, precision_score,
    recall_score, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
# Check if GPU is available
device = torch.device("cuda" if torch.cuda.is_available() else "
    cpu")
print(f"Using device: {device}")
# Load the best model weights and move the model to the
    appropriate device
model = EnhancedCNNImproved(num_classes=len(dataset.classes))  #
    Ensure the same architecture as the saved model
model.load_state_dict(torch.load('best_model.pth', map_location=
    device))  # Map weights to the current device
model.to(device)  # Move model to the selected device
model.eval()  # Set the model to evaluation mode
# Initialize variables for accuracy calculation
correct = 0
total = 0
all_labels = []
all_predictions = []
# Disable gradient computation for evaluation
```

```
with torch.no_grad():
    for images, labels in test_loader:  # Iterate through the test DataLoader
        images, labels = images.to(device), labels.to(device)  # Move data to the same device as the model
        # Forward pass: Get predictions
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)  # Get the class with the highest score
        # Update counts
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        # Store labels and predictions for metrics
        all_labels.extend(labels.cpu().numpy())
        all_predictions.extend(predicted.cpu().numpy())
# Calculate accuracy
accuracy = 100 * correct / total
print(f"Test Accuracy of the model: {accuracy:.2f}%")
# Compute confusion matrix
conf_matrix = confusion_matrix(all_labels, all_predictions)
print("Confusion Matrix:")
print(conf_matrix)
# Plot confusion matrix
disp = ConfusionMatrixDisplay(conf_matrix, display_labels=range(num_classes))
disp.plot(cmap='Blues', xticks_rotation='vertical')
plt.title("Confusion Matrix")
plt.show()
# Calculate precision and recall
precision = precision_score(all_labels, all_predictions, average='weighted')
recall = recall_score(all_labels, all_predictions, average='weighted')
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
```

Test Accuracy: 70.24%
Precision: 0.71
Recall: 0.70

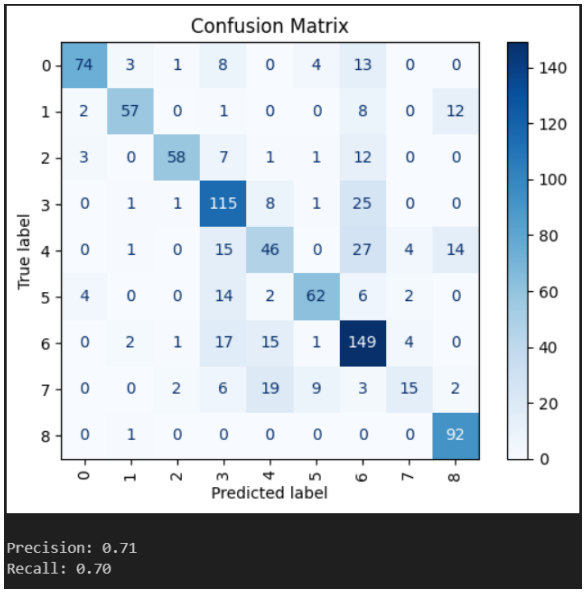

Figure 8: Model Evaluation

1.11 Plot training and validation loss for Different Learning Rates

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from tqdm import tqdm
# Assuming DataLoader and model setup is already complete
device = torch.device("cuda" if torch.cuda.is_available() else "
    cpu")
# Hyperparameters
num_epochs = 5
learning_rates = [0.0001, 0.001, 0.01, 0.1]
# Track losses for each learning rate
lr_results = {}
for lr in learning_rates:
    print(f"\nTraining with learning rate: {lr}")
    # Define model, criterion, and optimizer
    model = EnhancedCNNImproved(num_classes=len(dataset.classes))
        .to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
    # Initialize variables to track training
    train_losses = []
    val_losses = []
    for epoch in range(num_epochs):
        # Training phase
        model.train()
```

```
        train_loss = 0.0
        for images, labels in tqdm(train_loader, desc=f"Epoch {epoch+1}/{num_epochs} [Training] (LR: {lr})"):
            images, labels = images.to(device), labels.to(device)
            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)
            # Backward pass and optimization
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            train_loss += loss.item()
        train_loss /= len(train_loader)
        train_losses.append(train_loss)
        # Validation phase
        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for images, labels in tqdm(val_loader, desc=f"Epoch {epoch+1}/{num_epochs} [Validation] (LR: {lr})"):
                images, labels = images.to(device), labels.to(device)
                # Forward pass
                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
        val_loss /= len(val_loader)
        val_losses.append(val_loss)
        # Print losses
        print(f"Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss:.4f}, Val Loss: {val_loss:.4f}")
    # Save results for this learning rate
    lr_results[lr] = {
        "train_losses": train_losses,
        "val_losses": val_losses
    }
# Plot training and validation loss for all learning rates
plt.figure(figsize=(12, 8))
for lr, results in lr_results.items():
    plt.plot(range(1, num_epochs + 1), results["train_losses"], label=f"Train Loss (LR: {lr})", marker='o')
    plt.plot(range(1, num_epochs + 1), results["val_losses"], label=f"Val Loss (LR: {lr})", marker='x')
plt.title("Training and Validation Loss for Different Learning Rates")
plt.xlabel("Epochs")
plt.ylabel("Loss")
```

```
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()
# Comment on results
for lr, results in lr_results.items():
    print(f"Learning Rate: {lr}")
    print(f"  Final Train Loss: {results['train_losses'][-1]:.4f}
       ")
    print(f"  Final Val Loss: {results['val_losses'][-1]:.4f}")
```

To evaluate the impact of different learning rates on model performance, we plotted the training and validation loss curves over 10 epochs for learning rates of 0.0001, 0.001, 0.01, and 0.1. Due to computational constraints in Colab's free tier, we limited training to 10 epochs instead of the intended 20 epochs.

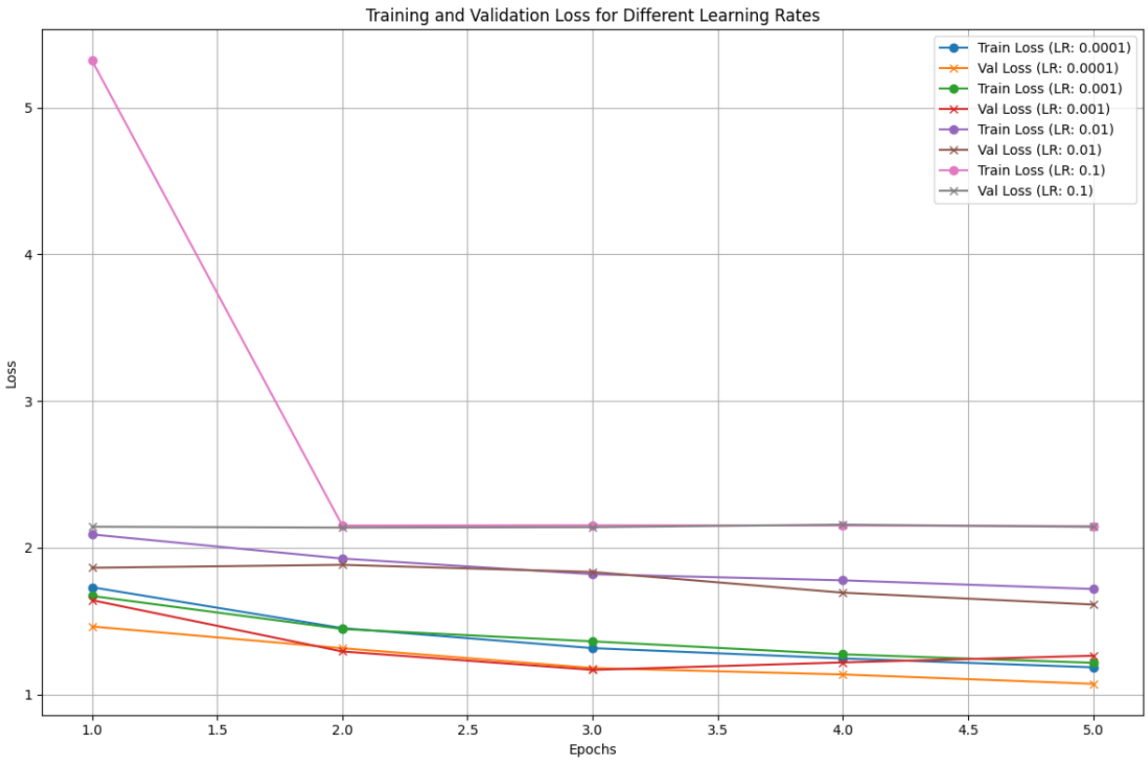

Figure 9: Training and Validation Loss for different Learining Rates

```
Learning Rate: 0.0001
  Final Train Loss: 1.1843
  Final Val Loss: 1.0720
Learning Rate: 0.001
  Final Train Loss: 1.2150
  Final Val Loss: 1.2634
Learning Rate: 0.01
  Final Train Loss: 1.7183
  Final Val Loss: 1.6124
Learning Rate: 0.1
  Final Train Loss: 2.1472
  Final Val Loss: 2.1418
```

Figure 10: Training and Validation Loss for different Learining Rates

Learning Rate: 0.0001 is the best choice based on the results:

- **Stable Convergence:** It shows a steady decrease in both training and validation loss, indicating that the model is learning effectively.
- **Less Overfitting:** Compared to the learning rate of 0.001, the model does not start to overfit, as the validation loss continues to decrease, indicating better generalization.
- **Slow Convergence:** While the convergence is slow, it is more stable compared to higher learning rates like 0.01 and 0.1, which caused the model to oscillate or diverge.

2 Compare our network with state-of-the-art networks

For the Comparison we have used Resnet18 and VGG16 pretrained models.

2.1 Finetune and training the pretrain Model on our Dataset

The below code snippet is used for importing the necessary libraries.
torchvision.models: Pretrained models like ResNet18 and VGG16.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import models
from torch.utils.data import DataLoader
```

The below code is used to load two pretrained models, ResNet18 and VGG16.

```

# Define the device
device = torch.device("cuda" if torch.cuda.is_available() else "
    cpu")

# Number of classes
num_classes = len(dataset.classes)

# Load pre-trained models
models_to_train = {
    "ResNet18": models.resnet18(pretrained=True),
    "VGG16": models.vgg16(pretrained=True)
}
```

The final layer of each model is updated to match the number of classes in the dataset.
ResNet18: Modifies model.fc (fully connected layer).
VGG16: Updates the last layer of model.classifier.

```
# Modify the last layer for the number of classes
for model_name, model in models_to_train.items():
    if model_name == "ResNet18":
        model.fc = nn.Linear(model.fc.in_features, num_classes)
    elif model_name == "VGG16":
        model.classifier[-1] = nn.Linear(model.classifier[-1].
            in_features, num_classes)
    model.to(device)
```

The model is trained in the following manner.

```

# Training parameters
```

```
num_epochs = 10
criterion = nn.CrossEntropyLoss()
batch_size = 32
learning_rate = 0.001


# Train and validate each model
results = {}
for model_name, model in models_to_train.items():
    print(f"\nTraining {model_name}...\n")

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

    # Record losses
    train_losses = []
    val_losses = []

    for epoch in range(num_epochs):
        # Training
        model.train()
        running_loss = 0.0
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        train_losses.append(running_loss / len(train_loader))

        # Validation
        model.eval()
        val_loss = 0.0
        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()
        val_losses.append(val_loss / len(val_loader))

        print(f"Epoch [{epoch + 1}/{num_epochs}], Train Loss: {
            train_losses[-1]:.4f}, Val Loss: {val_losses[-1]:.4f}"
            )
```

```
47
48      # Save the training and validation losses
49      results[model_name] = {"train_losses": train_losses, "
            val_losses": val_losses}
50
51      # Save the best model
52      torch.save(model.state_dict(), f"{model_name}_best.pth")
53
54  # Evaluate models on the test set
55  test_accuracies = {}
56  for model_name, model in models_to_train.items():
57      # Load the best model
58      model.load_state_dict(torch.load(f"{model_name}_best.pth"))
59      model.eval()
60
61      # Test accuracy
62      correct = 0
63      total = 0
64      with torch.no_grad():
65          for images, labels in test_loader:
66              images, labels = images.to(device), labels.to(device)
67              outputs = model(images)
68              _, predicted = torch.max(outputs, 1)
69              correct += (predicted == labels).sum().item()
70              total += labels.size(0)
71
72      accuracy = 100 * correct / total
73      test_accuracies[model_name] = accuracy
74      print(f"{model_name} Test Accuracy: {accuracy:.2f}%")
75
76  # Plot training and validation losses
77  import matplotlib.pyplot as plt
78
79  plt.figure(figsize=(12, 6))
80  for model_name, data in results.items():
81      plt.plot(data["train_losses"], label=f"{model_name} - Train
            Loss")
82      plt.plot(data["val_losses"], label=f"{model_name} - Val Loss"
            )
83  plt.xlabel("Epoch")
84  plt.ylabel("Loss")
85  plt.title("Training and Validation Losses")
86  plt.legend()
87  plt.grid()
88  plt.show()
89
90  # Print final test accuracies
```

```
print("\nFinal Test Accuracies:")
for model_name, accuracy in test_accuracies.items():
    print(f"{model_name}: {accuracy:.2f}%")
```

Training and validation loss values for each epoch

```
Training ResNet18...

Epoch [1/10], Train Loss: 1.1301, Val Loss: 1.2034
Epoch [2/10], Train Loss: 0.7336, Val Loss: 1.2255
Epoch [3/10], Train Loss: 0.6370, Val Loss: 1.1331
Epoch [4/10], Train Loss: 0.5567, Val Loss: 0.9247
Epoch [5/10], Train Loss: 0.3725, Val Loss: 0.9281
Epoch [6/10], Train Loss: 0.2607, Val Loss: 0.8451
Epoch [7/10], Train Loss: 0.2516, Val Loss: 0.8539
Epoch [8/10], Train Loss: 0.2417, Val Loss: 1.2445
Epoch [9/10], Train Loss: 0.2822, Val Loss: 0.9402
Epoch [10/10], Train Loss: 0.1641, Val Loss: 0.7132

Training VGG16...

Epoch [1/10], Train Loss: 2.1812, Val Loss: 2.8713
Epoch [2/10], Train Loss: 1.7872, Val Loss: 1.5261
Epoch [3/10], Train Loss: 1.4946, Val Loss: 1.4098
Epoch [4/10], Train Loss: 1.4231, Val Loss: 1.2742
Epoch [5/10], Train Loss: 1.3049, Val Loss: 1.3774
Epoch [6/10], Train Loss: 1.2828, Val Loss: 1.2734
Epoch [7/10], Train Loss: 1.1435, Val Loss: 1.2155
Epoch [8/10], Train Loss: 1.1371, Val Loss: 1.2153
Epoch [9/10], Train Loss: 1.0493, Val Loss: 1.1240
Epoch [10/10], Train Loss: 0.9743, Val Loss: 1.2663
```

Figure 11: Training and validation loss values for each epoch

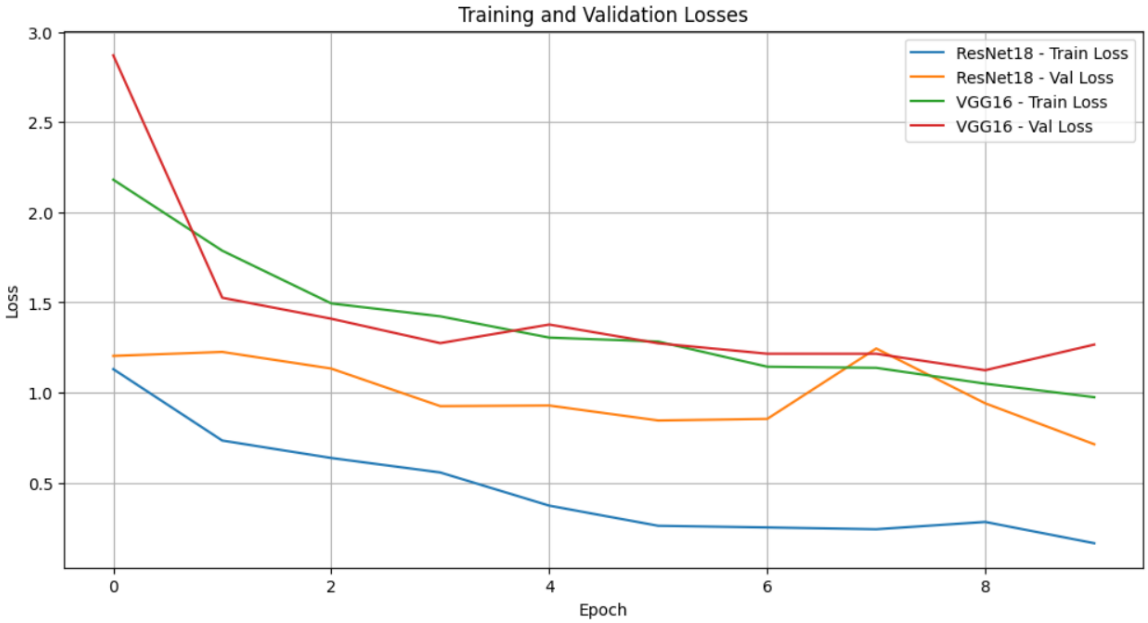

Figure 12: Training and validation loss values for each epoch plot

Test accuracies of the fine tuned models

```
Final Test Accuracies:
ResNet18: 79.07%
VGG16: 54.78%
```

Figure 13: Test accuracies of the fine tuned models

2.2 Comparison of the test accuracy our CNN model with fine tuned model

Our CNN model

In our model the training and validation losses show an overall decreasing trend, indicating that the model is learning effectively. The validation loss improves consistently, with the best validation loss (0.8930) achieved in Epoch 19. Checkpoints are saved at epochs where the validation loss improves, ensuring the best model is retained. At Epoch 15, the validation loss increases temporarily (1.2310), but the model regains performance in subsequent epochs. This may indicate the model is approaching overfitting.

ResNet18

Training Loss: Rapidly decreases from 1.1301 in epoch 1 to 0.1641 by epoch 10, indicating strong learning capacity.

Validation Loss: Shows improvement initially (reducing to 0.8451 at epoch 6) but exhibits variability afterward, including a spike at epoch 8. This suggests some overfitting as the training progresses.
Test Accuracy: 79.07%, which aligns with the better overall validation performance compared to VGG16. This demonstrates ResNet18's ability to generalize better on unseen data.

VGG16

Training Loss: Decreases steadily from 2.1812 in epoch 1 to 0.9743 by epoch 10, showing slower learning compared to ResNet18.
Validation Loss: Also improves initially, dropping to 1.2155 by epoch 7. However, the validation loss fluctuates toward the end.
Test Accuracy: 54.78%, significantly lower than ResNet18. This could be attributed to VGG16's relatively shallow structure (compared to ResNet18) struggling to capture the complexity of the dataset.

Training and Model Performance Observations

Both ResNet18 and VGG16 models were trained for 10 epoches. ResNet18 achieved a test accuracy of 79.07%, while VGG16 reached 54.78% within this training period. Notably, despite the relatively short training duration, ResNet18 demonstrated strong generalization performance, outperforming VGG16 significantly. But when comparing with our CNN model we could achieve only 70.24% accuracy eventhough we trained it for 20 epoches.

It is important to highlight that the current results were achieved with only 10 epochs of training. Based on the observed trends in the loss curves, it is evident that extending the training duration for both models, particularly ResNet18 and VGG16, to 20 epochs or more is likely to yield improved accuracy. ResNet18, with its deeper architecture and effective skip connections, is expected to benefit more from prolonged training, further refining its ability to generalize on unseen data. Similarly, VGG16, though slower in convergence, would also exhibit noticeable performance gains with extended training due to the continued optimization of its parameters.

2.3 Trade-offs, Advantages, and Limitations of using a custom model versus a pre-trained model

Custom Model

Advantages

- **Tailored to Specific Needs**: Custom models can be designed and trained for very specific tasks or datasets, leading to potentially better performance on niche problems.

- **Control Over Architecture**: You have full control over the model architecture, allowing you to experiment with different designs, such as custom layers, loss functions, or optimizations.
- **Data Privacy**: Since the model is trained on your own data, there is no need to rely on external data sources, which may be a concern for sensitive data applications.
- **Optimized for Domain**: Custom models can be designed to capture domain-specific features that a general pre-trained model might miss.

Limitations

- **Data and Resource Intensive**: Training a custom model from scratch requires a large amount of labeled data and significant computational resources (e.g., GPUs or TPUs).
- **Longer Development Time**: Training a custom model can take weeks or months, depending on the complexity of the task, whereas pre-trained models are ready to use immediately.
- **Expertise Required**: Designing and training a custom model effectively requires deep knowledge of machine learning concepts, model selection, and tuning.

Pre-trained Model

Advantages

- **Faster Deployment**: Pre-trained models are typically ready to use with minimal effort, significantly reducing the time required to deploy a model.
- **Reduced Training Time**: They have already been trained on large datasets (e.g., ImageNet, COCO), so fine-tuning or transfer learning can often yield excellent results with much less data and computation.
- **State-of-the-art Performance**: Pre-trained models often represent cutting-edge research and can outperform custom models on many general tasks (e.g., object detection, language modeling).
- **Less Data Needed**: You can often fine-tune pre-trained models with smaller datasets, reducing the need for a large amount of labeled data.

Limitations

- **Limited to Original Task**: Pre-trained models are generally optimized for tasks that are similar to the ones they were originally trained on, so their performance may suffer when applied to highly specific or different tasks.
- **Less Control Over Architecture**: You have limited ability to modify the internal architecture of a pre-trained model, which might not allow you to fully optimize it for your use case.

- **Overfitting Risk**: In some cases, fine-tuning a pre-trained model on a small dataset might lead to overfitting, as the model may not generalize well to new, unseen data.
- **Possible Licensing Issues**: Pre-trained models may have licensing restrictions, especially in commercial applications.

Trade-offs

- **Customization vs. Convenience**: Custom models offer flexibility but require significant effort in terms of data collection, model design, and training. Pre-trained models are quick and easy to deploy but may not be as adaptable to unique or highly specialized tasks.
- **Data Dependency**: Custom models require large labeled datasets to train effectively, while pre-trained models can work well even with small amounts of labeled data through fine-tuning.
- **Performance**: For general tasks, pre-trained models often offer state-of-the-art performance, but for highly specific tasks, a custom model may outperform pre-trained models once it has been sufficiently trained.

Custom Model	**Pre-trained Model**
Optimized for specific tasks, datasets	General-purpose, not specialized
Full control over model design and architecture	Limited control, predefined architecture
Requires large labeled datasets and extensive training	Works well with smaller datasets via fine-tuning
Longer, requires training from scratch	Shorter, can be deployed quickly after fine-tuning
High, needs significant computational resources	Low, pre-trained models are ready for use
Potentially better for specialized tasks	Often state-of-the-art for general tasks
Lower risk if properly regularized	Higher risk, especially with small datasets
No licensing issues (unless using external data sources)	Possible licensing restrictions, especially for commercial use
Can capture domain-specific features	May miss some domain-specific features
Highly adaptable, but requires more effort	Less adaptable to unique or highly specialized tasks

Table 1: Summary on Comparison of Custom Models and Pre-trained Models