Department of Electronic & Telecommunication Engineering
University of Moratuwa

EN3150 – Pattern Recognition

Assignment 03: Simple convolutional neural network to perform classification.

Group Assignment

Dilshan N.L.	210129P
Dodangoda D.K.S.J.	210150V
Gunawardana W. N. M	210199D
Wathudura T.R	210682D

Date - 2024.12.12

Contents

1 CNN for image classification

1.1 Set up your environment.

```
pip install tensorflow matplotlib numpy pandas scikit-learn

import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt
from torchinfo import summary
from torch.utils.data import DataLoader
from torch.utils.data import Dataset
from torchvision.datasets import ImageFolder
import random
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
BATCH_SIZE = 32
print(device)
```

Listing 1: Set up the environment

1.2 Prepare your dataset.

Selected dataset: Real Waste dataset

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

# Define the dataset path
dataset_path = 'realwaste-main/RealWaste'

# Transformations for PyTorch
transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor()
])

# Load dataset with ImageFolder and transformations
full_dataset = ImageFolder(root=dataset_path, transform=transform)
```

Listing 2: Prepare the dataset

1.3 Split the dataset into training, validation, and testing subsets.

```
# Load dataset with ImageFolder and transformations
full_dataset = ImageFolder(root=dataset_path, transform=transform)

# Calculate the correct split lengths
train_len = int(0.6 * len(full_dataset))
valid_len = int(0.2 * len(full_dataset))
test_len = len(full_dataset) - (train_len + valid_len)

# Split dataset into train, validation, and test
train_data, valid_data, test_data = random_split(full_dataset, [train_len, valid_len,
    test_len])

# Load data using DataLoader
train_dataloader = DataLoader(train_data, batch_size=32, shuffle=True)
valid_dataloader = DataLoader(valid_data, batch_size=32, shuffle=False)
test_dataloader = DataLoader(test_data, batch_size=32, shuffle=False)
```

Listing 3: Split the dataset

1.4 Build the CNN model.

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

class CNNModel(nn.Module):
    def __init__(self):
        super(CNNModel, self).__init__()
        self.convblock1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1),  # Added padding for better output size
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.convblock2 = nn.Sequential(
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1),  # Added padding
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.convblock3 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),  # Added padding
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.flatten = nn.Flatten()
        # The size after 3 convolution blocks with max pooling (each reduces size by half)
        # Input size: 128x128 -> After convblock1: 64x64 -> After convblock2: 32x32 -> After convblock3: 16x16
        self.classifier = nn.Sequential(
            nn.Linear(64 * 16 * 16, 128),  # Adjusted to match flattened output size
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(128, 32),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(32, 9)  # 9 classes as per the dataset
        )

    def forward(self, x):
        x = self.convblock1(x)
        x = self.convblock2(x)
        x = self.convblock3(x)
        x = self.flatten(x)
        x = self.classifier(x)
        return x
```

Listing 4: CNN model

1.5 Determine the parameters of the above network.

```python
model = CNNModel().to(device)
print(summary(CNNModel(), input_size=(BATCH_SIZE, 3, 131, 131)))
```

Listing 5: Get the parameters of the network

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
CNNModel                                 [32, 9]                   --
    Sequential    : 1-1                      [32, 16, 65, 65]          --
           Conv2d    : 2-1                       [32, 16, 131, 131]        448
           ReLU    : 2-2                       [32, 16, 131, 131]        --
           MaxPool2d    : 2-3                  [32, 16, 65, 65]          --
    Sequential    : 1-2                      [32, 32, 32, 32]          --
           Conv2d    : 2-4                       [32, 32, 65, 65]          4,640
           ReLU    : 2-5                       [32, 32, 65, 65]          --
```

```
            MaxPool2d    : 2-6                       [32, 32, 32, 32]          --
    Sequential    : 1-3                             [32, 64, 16, 16]          --
            Conv2d    : 2-7                         [32, 64, 32, 32]          18,496
            ReLU    : 2-8                           [32, 64, 32, 32]          --
            MaxPool2d    : 2-9                      [32, 64, 16, 16]          --
    Flatten    : 1-4                                [32, 16384]               --
    Sequential    : 1-5                             [32, 9]                   --
            Linear    : 2-10                        [32, 128]                 2,097,280
            ReLU    : 2-11                          [32, 128]                 --
            Dropout    : 2-12                       [32, 128]                 --
            Linear    : 2-13                        [32, 32]                  4,128
            ReLU    : 2-14                          [32, 32]                  --
            Dropout    : 2-15                       [32, 32]                  --
            Linear    : 2-16                        [32, 9]                   297
==========================================================================================

Total params: 2,125,289
Trainable params: 2,125,289
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 1.55
==========================================================================================

Input size (MB): 6.59
Forward/backward pass size (MB): 121.72
Params size (MB): 8.50
Estimated Total Size (MB): 136.81
==========================================================================================
```

Listing 6: Parameters of the network

The architecture of the Convolutional Neural Network (CNN) is designed as follows:

- **Convolutional Layer** : The first layer consists of 16 filters, each with a 3×3 kernel, and uses the ReLU activation function to introduce non-linearity.
- **MaxPooling Layer** : A MaxPooling operation with a pool size of 2×2 is applied to reduce spatial dimensions and retain important features.
- **Convolutional Layer** : A second convolutional layer with 32 filters, a 3×3 kernel, and ReLU activation to capture higher-level features.
- **MaxPooling Layer** : Another MaxPooling operation (2×2) is used for further dimensionality reduction.
- **Convolutional Layer** : The third convolutional layer has 64 filters, a 3×3 kernel, and ReLU activation, deepening the model's ability to detect complex patterns.
- **Flattening Layer**: The output of the convolutional layers is flattened into a 1D vector, preparing it for the fully connected layers.
- **Fully Connected Layer**: A dense layer with 128 units and ReLU activation, enabling the network to learn complex representations.
- **Dropout Layer** : A dropout layer with a rate of 0.5 is applied to reduce overfitting by randomly deactivating half of the neurons during training.
- **Fully Connected Layer** : Another dense layer with 32 units and ReLU activation further refines the learned features.
- **Dropout Layer**: Another dropout layer with a rate of 0.5 is applied to mitigate overfitting.
- **Output Layer** : The final output layer consists of 9 units, corresponding to 9 possible classes, and uses the 'softmax' activation function to produce probability distributions over the classes.

This architecture is designed to effectively capture spatial hierarchies and high-level features while minimizing overfitting through dropout layers.

1.6 Provide justifications for activation functions selections.

The **Rectified Linear Unit (ReLU)** activation function is applied to all hidden layers of the neural network, while the final (output) layer uses the **Softmax** activation function.

ReLU is chosen because it is computationally efficient and accelerates convergence during training. Compared to other activation functions like sigmoid and tanh, ReLU involves a simple thresholding operation at zero, avoiding the complex computations required by sigmoid and tanh. This efficiency makes ReLU particularly effective for deep learning tasks.

Another advantage of ReLU is that it alleviates the issue of "vanishing gradients," which commonly occurs with sigmoid and tanh functions. These functions tend to squash input values into narrow ranges, producing small gradients that slow down the learning process in deep networks. ReLU, in contrast, does not saturate for positive inputs and passes them through directly, ensuring a more consistent gradient flow and faster learning.

ReLU is also less prone to "exploding gradients." Its output range (0 to positive infinity) stabilizes the training process by keeping gradients within a manageable range. This ensures smooth backpropagation and helps the network to train effectively, even as it grows deeper.

The output layer uses the **Softmax** activation function, which converts the raw logits into a probability distribution. Each class is assigned a probability between 0 and 1, and the probabilities sum to 1. This makes Softmax a natural choice for multi-class classification problems, as it provides clear and interpretable results.

1.7 Train the model.

```python
def train(model, train_loader, valid_loader, epochs, optimizer, criterion, device):
    """
    Train and validate the model.

    Args:
        model: The PyTorch model to train.
        train_loader: DataLoader for the training set.
        valid_loader: DataLoader for the validation set.
        epochs: Number of training epochs.
        optimizer: Optimizer for the model.
        criterion: Loss function.
        device: Device to run the training on (e.g., 'cuda' or 'cpu').

    Returns:
        train_accuracy_hist: List of training accuracies.
        test_accuracy_hist: List of validation accuracies.
        train_loss_hist: List of training losses.
        test_loss_hist: List of validation losses.
    """
    train_accuracy_hist = []
    test_accuracy_hist = []
    train_loss_hist = []
    test_loss_hist = []

    for epoch in range(epochs):
        model.train()  # Set model to training mode
        running_loss = 0.0
        correct_train = 0
        total_train = 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()
            _, predicted = outputs.max(1)
            correct_train += (predicted == labels).sum().item()
            total_train += labels.size(0)

        train_accuracy = 100 * correct_train / total_train
        train_loss = running_loss / len(train_loader)
        train_accuracy_hist.append(train_accuracy)
        train_loss_hist.append(train_loss)

        # Validation step
        model.eval()  # Set model to evaluation mode
        running_loss = 0.0
        correct_valid = 0
        total_valid = 0

        with torch.no_grad():
            for images, labels in valid_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)

                running_loss += loss.item()
                _, predicted = outputs.max(1)
                correct_valid += (predicted == labels).sum().item()
                total_valid += labels.size(0)

        valid_accuracy = 100 * correct_valid / total_valid
        valid_loss = running_loss / len(valid_loader)
        test_accuracy_hist.append(valid_accuracy)
        test_loss_hist.append(valid_loss)

        print(f"Epoch [{epoch+1}/{epochs}], "
              f"Train Accuracy: {train_accuracy:.2f}%, "
              f"Valid Accuracy: {valid_accuracy:.2f}%, "
              f"Train Loss: {train_loss:.4f}, "
              f"Valid Loss: {valid_loss:.4f}")

    return train_accuracy_hist, test_accuracy_hist, train_loss_hist, test_loss_hist
```

Listing 7: Train the model

1.8 Why have we chosen Adam optimizer over SGD?

The **Adam optimizer** was chosen instead of **Stochastic Gradient Descent (SGD)** due to its efficiency and adaptability. Key advantages include:

- **Adaptive Learning Rates:** Adam adjusts the learning rate for each parameter based on its historical gradients. This feature ensures faster convergence and makes Adam effective in scenarios with sparse gradients or complex datasets.
- **Momentum:** By maintaining a moving average of past gradients and their variances, Adam smooths updates and prevents oscillations. This momentum helps the optimizer move past saddle points and local minima efficiently.
- **Ease of Use:** Adam works well with default hyperparameters, reducing the need for manual tuning. This makes it accessible and reliable for a wide range of problems.
- **Robustness:** Adam handles noisy gradients and sparse data effectively, ensuring stability during training, even under challenging conditions.
- **Deep Learning Compatibility:** Adam performs exceptionally well with deep networks, navigating their complex loss landscapes more efficiently than SGD.

Overall, Adam's adaptive nature, momentum, and stability make it the preferred choice for this project.

1.9 Why have we chosen sparse categorical crossentropy as the loss function?

The **Sparse Categorical Crossentropy** loss function was selected for its compatibility with the problem and computational efficiency:

- **Softmax Compatibility:** This loss function works seamlessly with the Softmax output layer, enabling efficient comparison of predicted probabilities with true labels.
- **Integer-Encoded Labels:** Unlike traditional Categorical Crossentropy, Sparse Categorical Crossentropy works directly with integer labels, avoiding the overhead of one-hot encoding.
- **Designed for Multi-Class Problems:** It is well-suited for scenarios with multiple classes, ensuring accurate computation of loss and facilitating optimization.
- **Numerical Stability:** This loss function enhances computational stability by avoiding sparse vector calculations, reducing the risk of numerical issues.
- **Memory Efficiency:** By using integer labels, it reduces memory usage during training and inference, making it ideal for large datasets.

Overall, Sparse Categorical Crossentropy ensures accurate, efficient, and stable optimization for multi-class classification tasks.

1.10 Evaluate the model(for learning rate = 0.001).

```
Train Accuracy: 72.25%
Valid Accuracy: 64.95%
```

Listing 8: results

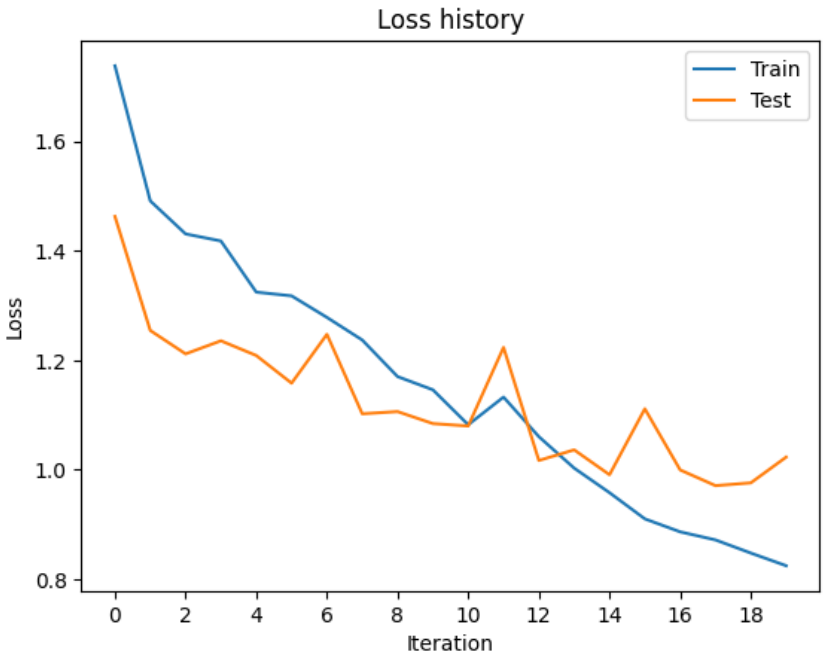

Figure 1: Loss history

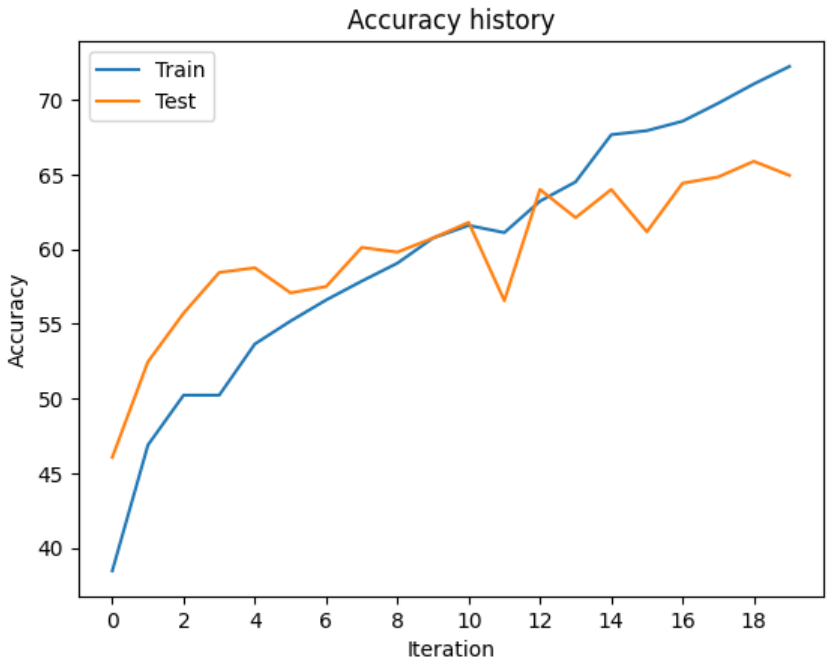

Figure 2: Accuracy

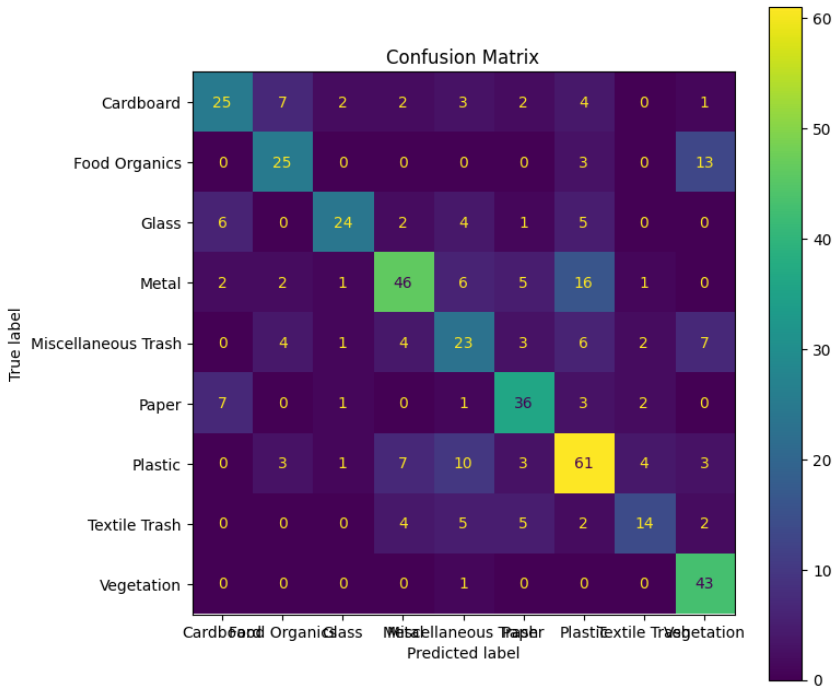

Figure 3: Confusion Matrix

1.11 Plot training and validation loss for with respect to epoch for different learning rates such as 0. 0001, 0. 001, 0. 01, and 0. 1. Comment on your results and select a learning rate with a justification.

Learning rate = 0.0001

```
Train Accuracy: 50.11%
```

```
Valid Accuracy: 55.40%
```

Listing 9: results

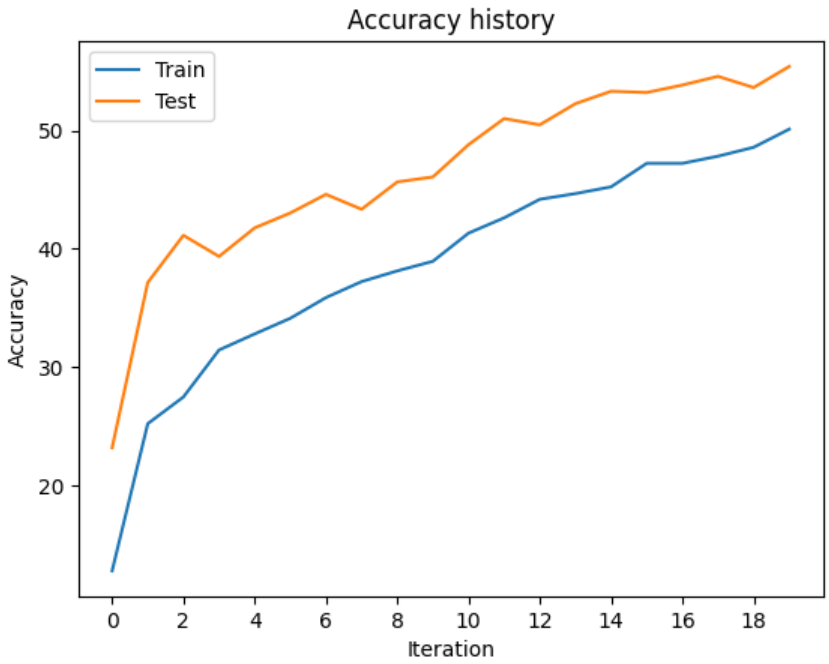

Figure 4: Accuracy

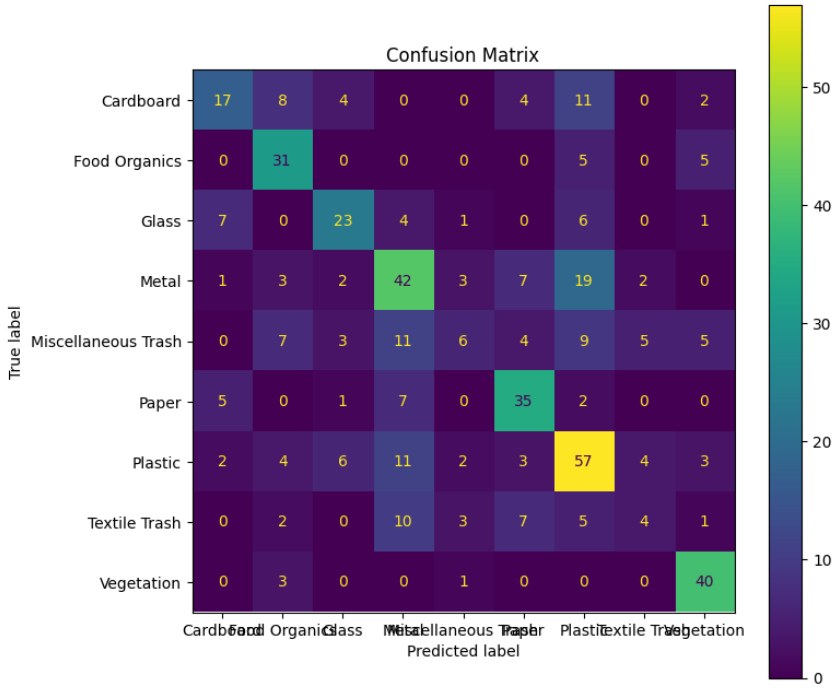

Figure 5: Confusion Matrix

Accuracy is increasing with the number of iterations. So, overall this learning rate is good. But the final accuracy after 20 iteration is not good.

Learning rate = 0.001

```
Train Accuracy: 72.25%
Valid Accuracy: 64.95%
```

Listing 10: results

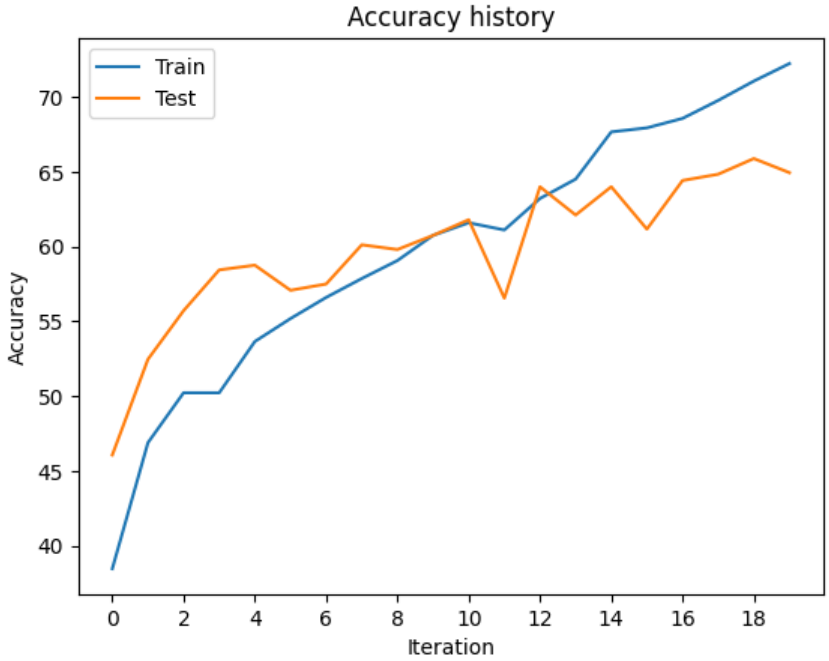

Figure 6: Accuracy

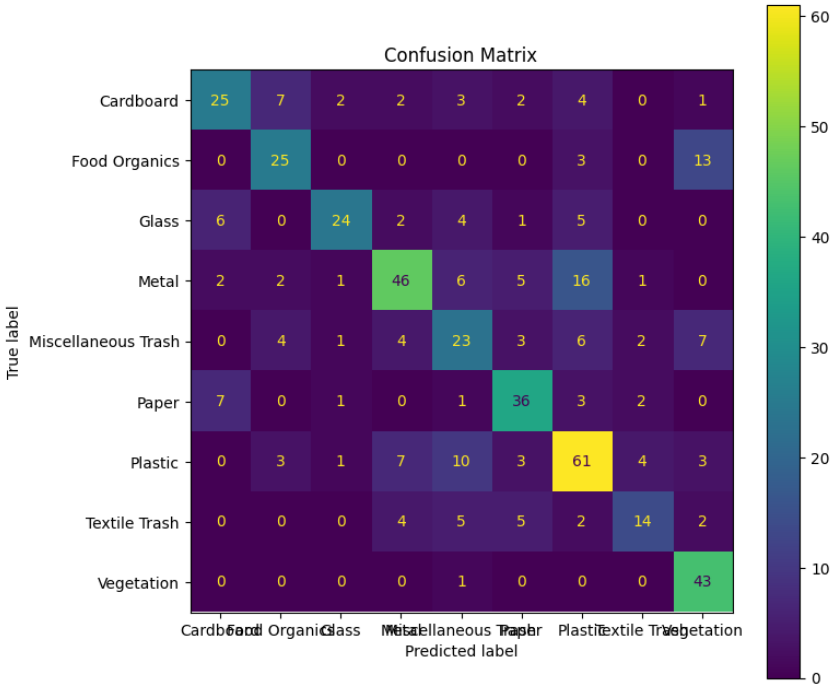

Figure 7: Confusion Matrix

Accuracy is increasing with the number of iterations. So, overall this learning rate is good. Also the final accuracy value is better than the previous one.

Learning rate = 0.01

```
Train Accuracy: 19.38%
Valid Accuracy: 19.41%
```

Listing 11: results

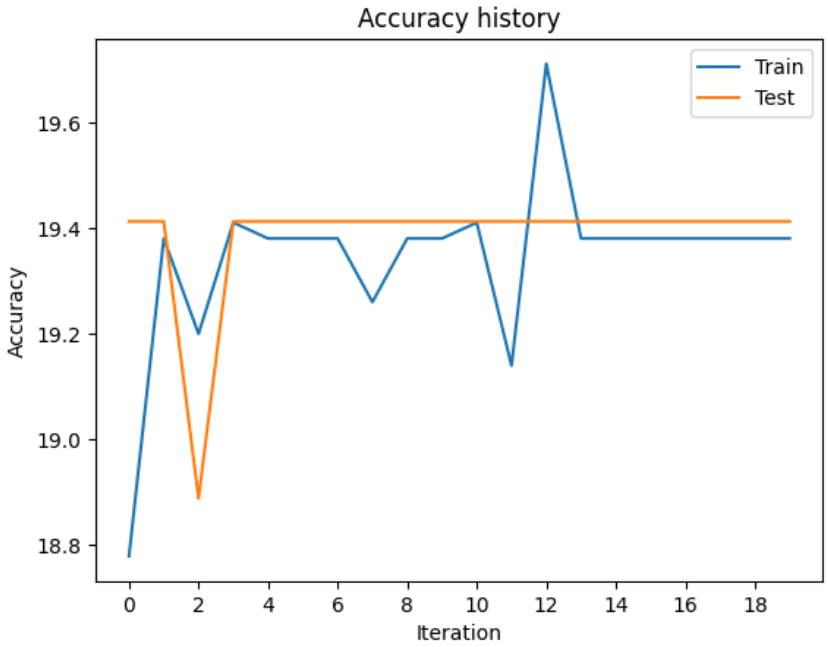

Figure 8: Accuracy

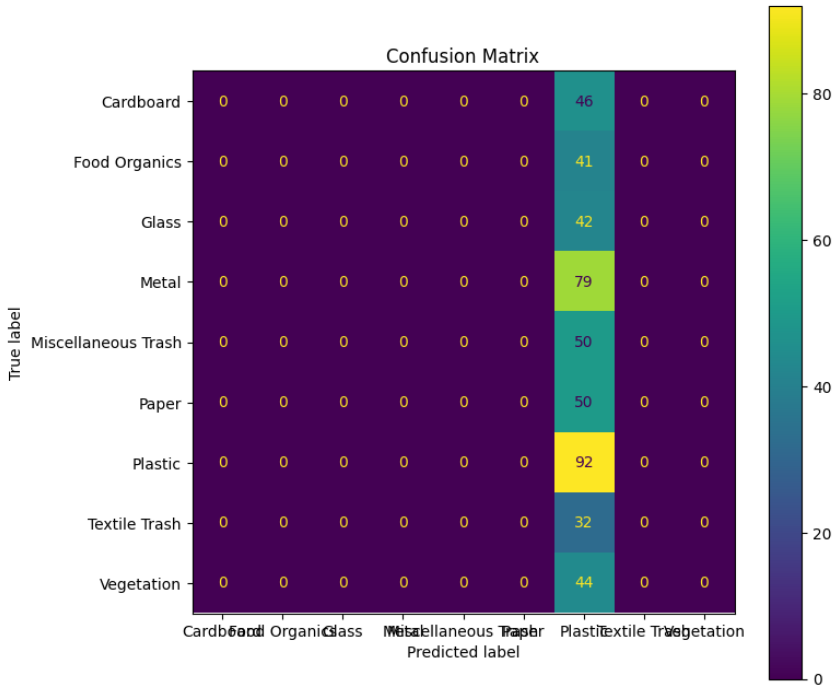

Figure 9: Confusion Matrix

The accuracy remains largely unchanged throughout the epochs, indicating that the model is not

learning effectively. Both train and validation losses fluctuate slightly, but there is no significant improvement over time.

Learning rate = 0.1

```
Train Accuracy: 18.54%
Valid Accuracy: 19.41%
```

Listing 12: results

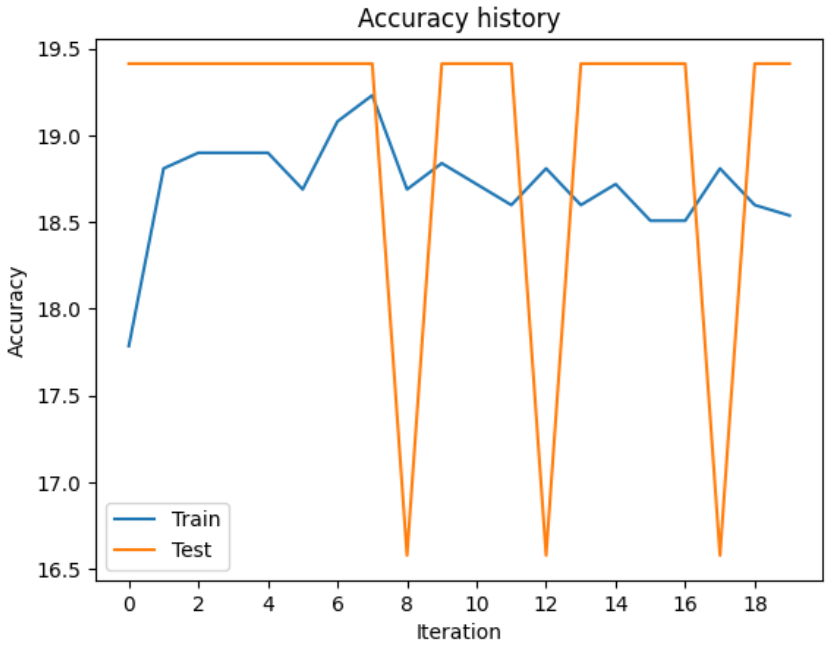

Figure 10: Accuracy

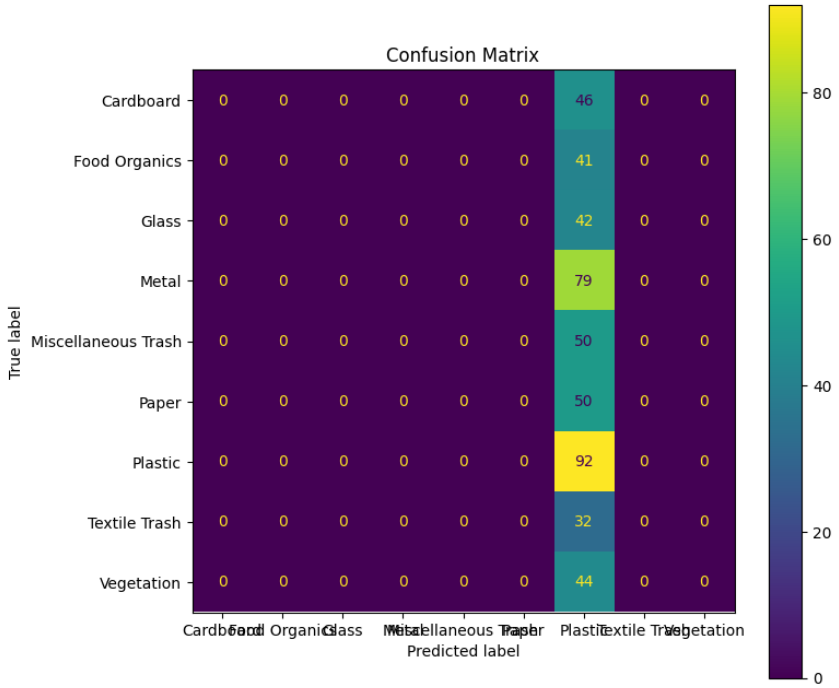

Figure 11: Confusion Matrix

The model performs poorly with no significant improvement in accuracy, showing fluctuations in both training and validation loss. This suggests that the model is not converging and requires further adjustments.

Observations

After training the model with different learning rates (0.0001, 0.001, 0.01, and 0.1), the model with a learning rate of 0.001 performed the best based on validation accuracy. A learning rate of 0.001 strikes a good balance between slow convergence (as seen with 0.0001) and instability (observed with higher rates like 0.01 or 0.1), leading to better accuracy in 20 epochs. Therefore, 0.001 is optimal for stable and efficient training, providing the best performance overall.

2 Compare your network with state-of-the-art networks.

2.12 Choose two state-of-the-art pre-trained models or architectures.

Selected models: Resnet, VGG

2.13 Resnet

2.13.1 Load the pre-trained model and fine-tune it for the your dataset.

```
# Function to fine-tune a model
def fine_tune_model(pretrained_model, num_classes):
    # Modify the final layer
    for param in pretrained_model.parameters():
        param.requires_grad = False  # Freeze all layers except the final layer

    # Replace the classifier layer to match the number of classes
    if isinstance(pretrained_model, models.ResNet):
        pretrained_model.fc = nn.Linear(pretrained_model.fc.in_features, num_classes)
    elif isinstance(pretrained_model, models.GoogLeNet):
        pretrained_model.fc = nn.Linear(pretrained_model.fc.in_features, num_classes)

    return pretrained_model.to(device)

# Load ResNet and GoogLeNet models
resnet_model = fine_tune_model(models.resnet18(pretrained=True), num_classes)
googlenet_model = fine_tune_model(models.googlenet(pretrained=True), num_classes)
```

Listing 13: Load the pre-trained model and fine-tune it

2.13.2 Train the fine-tuned model using the same training and testing data splits as your custom CNN model.

```
# Train ResNet
print("\nTraining ResNet:")
resnet_train_loss, resnet_valid_loss = train_model(resnet_model, train_loader,
    valid_loader, epochs=20, optimizer=resnet_optimizer)
```

Listing 14: Train the fine-tuned model

2.13.3 Record training and validation loss values for each epoch.

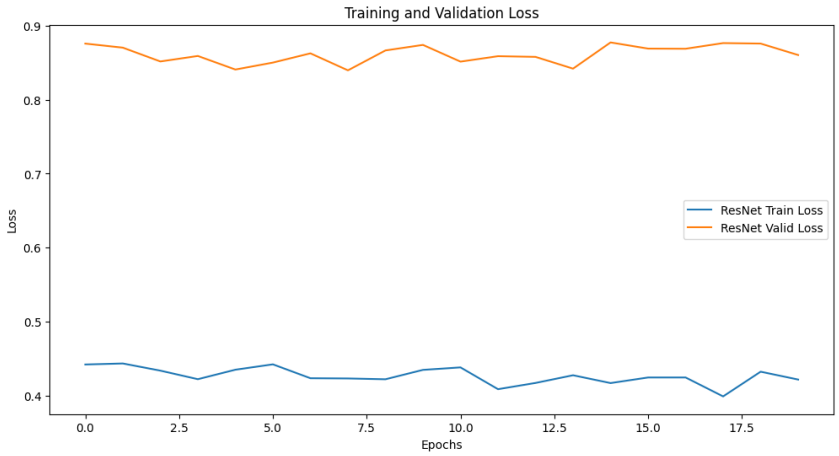

Figure 12: Loss values

2.13.4 Evaluate the fine-tuned model on the testing dataset and calculate the test accuracy.

```
ResNet Test Accuracy: 71.85%
```

Listing 15: Evaluate the fine-tuned model

2.14 VGG

2.14.1 Load the pre-trained model and fine-tune it for the your dataset.

```python
# Function to fine-tune a model
def fine_tune_vgg(pretrained_model, num_classes, device):

    # Freeze all convolutional layers
    for param in pretrained_model.features.parameters():
        param.requires_grad = False

    # Modify the classifier to match the number of classes
    in_features = pretrained_model.classifier[-1].in_features
    pretrained_model.classifier[-1] = nn.Linear(in_features, num_classes)

    # Move the model to the appropriate device
    return pretrained_model.to(device)

vgg_model = fine_tune_vgg(models.vgg16(pretrained=True), num_classes, device)
```

Listing 16: Load the pre-trained model and fine-tune it

2.14.2 Train the fine-tuned model using the same training and testing data splits as your custom CNN model.

```python
import torch.optim as optim

# Define the optimizer
learning_rate = 0.001
vgg_optimizer = optim.Adam(vgg_model.parameters(), lr=learning_rate)

# Train the model
print("\nTraining VGG:")
vgg_train_loss, vgg_valid_loss = train_model(vgg_model, train_loader, valid_loader,
    epochs=20, optimizer=vgg_optimizer)
```

Listing 17: Train the fine-tuned model

2.14.3 Record training and validation loss values for each epoch.

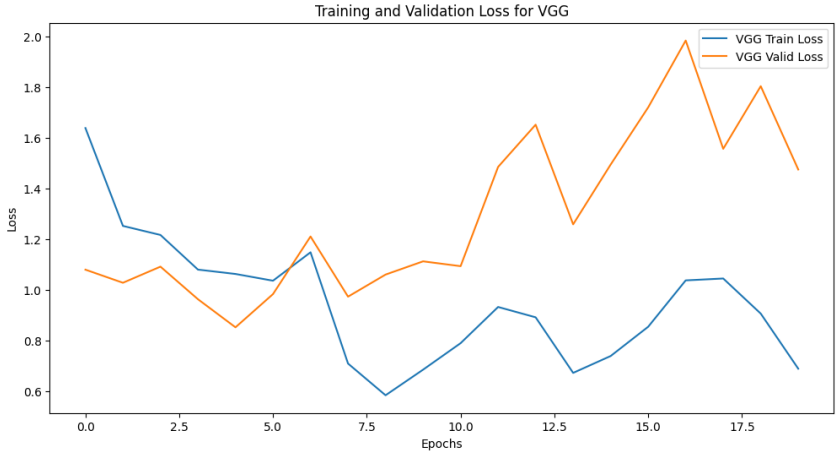

Figure 13: Loss values

2.14.4 Evaluate the fine-tuned model on the testing dataset and calculate the test accuracy.

```
VGG Test Accuracy: 79.20%
```

Listing 18: Evaluate the fine-tuned model

2.15 Compare the test accuracy of your custom CNN model with that of the fine-tuned state-of-the-art model.

- **Custom CNN**: Best validation accuracy of **64.95%**.
- **ResNet**: Test accuracy of **71.85%**.
- **VGG**: Test accuracy of **79.20%**.

From the results, we can conclude that the pre-trained models (ResNet and VGG) perform better than the custom CNN on the test data. While the custom CNN achieved a decent validation accuracy, it lags behind both ResNet and VGG in terms of test performance.

A notable observation is that for **ResNet**, the training loss is significantly higher than the test loss, which suggests that the model may be overfitting or having difficulty adapting to the specific dataset. In contrast, **VGG** performs more consistently across both training and testing phases, with a more stable and lower loss, making it the more reliable model in this case.

Furthermore, despite continuous training, the pre-trained models (ResNet and VGG) show minimal changes in the loss values. This is expected, as these models have already been trained on large and diverse datasets, allowing them to generalize well and perform effectively on a wide range of tasks.

2.16 Discuss trade-offs, advantages, and limitations of using a custom model versus a pre-trained model.

Custom Model

Advantages:

- Tailored to specific datasets, offering flexibility and control over architecture.
- No dependency on pre-trained features, potentially better for niche tasks.

Limitations:

- Requires large datasets to perform well.
- Long training times and extensive hyperparameter tuning are needed.

Pre-trained Model

Advantages:

- Faster training with minimal data due to learned features from large datasets.
- Strong generalization ability, leading to better performance on smaller datasets.

Limitations:

- Less flexibility; may not be ideal for very specific tasks.
- Risk of overfitting if not fine-tuned properly on domain-specific data.

Trade-offs

- **Training Time vs Accuracy:** Custom models take longer to train but can be optimized for specific tasks. Pre-trained models converge faster but may not always perform as well on specialized tasks.
- **Data Requirements:** Custom models need more data, while pre-trained models perform well with less.
- **Overfitting Risk:** Custom models are more prone to overfitting, while pre-trained models can generalize better.

Conclusion

Pre-trained models are ideal for smaller datasets or when fast deployment is needed. Custom models offer better optimization for specific tasks but require more data and training time.

GitHub Profile

Repository link: GitHub Repository