Department of Electronic & Telecommunication Engineering, University of Moratuwa, Sri Lanka.

Simple Convolutional Neural Network to Perform classification
Assignment 3

by:

220005R	AAZIR M.A.M
220146A	DULSARA G.M.L.
220423V	NETHKALANA W.K.S.
220735E	YASHODHARA.M.H.K.

Submitted in partial fulfillment of the requirements for the module
EN3150 Pattern Recognition

28 th October 2025

Contents

1 Introduction

This assignment focuses on developing a simple image classifier using a CNN model from scratch and comparing its performance with fine tuned state-of-the-art pre trained architectures. The task involves selecting a suitable dataset from the UCI Machine Learning Repository, preprocessing the data, splitting it into training, validation and testing subsets and constructing a CNN model with convolutional, pooling and fully connected layers.
Furthermore, the assignment explores the use of transfer learning where pre trained models are fine tuned for the target classification task. By comparing a custom CNN with these pre trained models, this study aims to highlight the trade offs between training a model from scratch and leveraging pre existing architectures.
Overall, this exercise provides a hands-on understanding of deep learning workflows for image classification, including model design, parameter tuning, optimizer selection and performance evaluation.

1.1 Dataset Used

The dataset used for this project is the Real Waste dataset which contains images of different types of waste collected under real world conditions. It has nine categories: **cardboard, food organics, glass, metal, miscellaneous trash, paper, plastic, textile trash and vegetation.** This dataset was chosen because it provides a variety of samples that represent common waste materials, making it suitable for training a waste classification model.
Before training, all images were preprocessed to have consistent size and format. The dataset was then split into training (70%), validation (15%) and testing (15%) sets to allow proper model training, tuning and evaluation.
Using this dataset, the model is trained to classify waste accurately, simulating a real world scenario important for recycling and waste management.

2 Custom CNN Model

2.1 CNN Architecture Used

The CNN model is built using a series of convolutional and max pooling layers that help the network learn different image features step by step. Batch normalization and ReLU activation functions are added to make training more stable and efficient. After the convolutional layers, a global average pooling layer reduces the data before passing it to the fully connected layers which classify the images into their respective categories.

2.1.1 Importing Libraries

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim.lr_scheduler import StepLR
from torchvision import datasets, models, transforms
import os
import shutil
import time
import cv2
import pandas as pd
from torch.utils.data import Dataset, DataLoader, WeightedRandomSampler, ConcatDataset
from PIL import Image
import random
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt
from collections import Counter, defaultdict
from sklearn.metrics import classification_report, confusion_matrix, precision_score, recall_score, f1_score
import seaborn as sns
import copy
import splitfolders
```

2.1.2 Dataset Splitting

```
base_dir = "/kaggle/input/realwaste/realwaste-main/RealWaste"
split_dir = "/kaggle/working/RealWaste_split"

# Split dataset into train (70%), val (15%), test (15%)
splitfolders.ratio(base_dir, output=split_dir, seed=42, ratio=(.7, .15, .15))
```

2.1.3 Data Augmentation

```
IMG_SIZE = (224, 224)

train_transforms = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.RandomRotation(45),
    transforms.RandomResizedCrop(size=IMG_SIZE[0], scale=(0.6, 0.9)),
    transforms.Resize(IMG_SIZE),
    transforms.ToTensor()
])

val_test_transforms = transforms.Compose([
    transforms.Resize(IMG_SIZE),
    transforms.ToTensor()
])

train_dataset = datasets.ImageFolder(os.path.join(split_dir, 'train'), transform=train_transforms)
val_dataset   = datasets.ImageFolder(os.path.join(split_dir, 'val'), transform=val_test_transforms)
test_dataset  = datasets.ImageFolder(os.path.join(split_dir, 'test'), transform=val_test_transforms)
```

2.1.4 Model Definition

```
class CNNModel(nn.Module):
    def __init__(self, num_classes=9):
        super(CNNModel, self).__init__()
        self.conv_block1 = self._create_conv_block(3, 64)
        self.conv_block2 = self._create_conv_block(64, 128)
        self.conv_block3 = self._create_conv_block(128, 256)
        self.conv_block4 = self._create_conv_block(256, 512)
        self.conv_block5 = self._create_conv_block(512, 512)
        self.global_avg_pool = nn.AdaptiveAvgPool2d(1)
        self.fc1 = nn.Linear(512, 512)
        self.fc2 = nn.Linear(512, num_classes)
        self.dropout = nn.Dropout(0.3)

    def _create_conv_block(self, in_channels, out_channels):
        return nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

    def forward(self, x):
        x = self.conv_block1(x)
        x = self.conv_block2(x)
        x = self.conv_block3(x)
        x = self.conv_block4(x)
        x = self.conv_block5(x)
        x = self.global_avg_pool(x)
        x = x.view(x.size(0), -1)
        x = nn.functional.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x

model = CNNModel(num_classes=9)
```

2.1.5 CNN Model Parameters

The CNN architecture consists of five convolutional blocks followed by a fully connected layer and an output layer. Each block uses ReLU activation and batch normalization to stabilize learning. The model also employs a dropout layer to prevent overfitting.

Table 1: Parameters of the CNN Model

Parameter	**Description**
Activation Function	ReLU (Rectified Linear Unit) after each convolution layer
Number of Convolution Blocks	5
Kernel Size	3×3 for all convolution layers
Padding	1 (to preserve spatial dimensions)
Filter Sizes	64, 128, 256, 512, 512 (for each successive block)
Pooling Type	Max Pooling (2×2) after each block
Normalization	Batch Normalization applied after each convolution layer
Fully Connected Layers	Two linear layers: $512 \rightarrow 512 \rightarrow 9$ (output classes)
Dropout Rate	0.3 (applied before the final fully connected layer)
Global Pooling	Adaptive Average Pooling (output size = 1)

2.1.6 Activation Function Justification

The ReLU (Rectified Linear Unit) activation function was chosen because it helps the model learn faster and avoids the vanishing gradient problem that often occurs with sigmoid or tanh activations. ReLU introduces non-linearity, allowing the network to learn complex patterns, while being computationally efficient. Its simplicity and effectiveness make it the most widely used activation function in modern CNN architectures.

2.2 Model Training

The CNN model was trained for 20 epochs using the Adam optimizer and cross entropy loss function. The training process aimed to minimize classification error while maintaining good generalization. Model performance was evaluated on the validation set after each epoch to monitor accuracy and loss, ensuring effective learning and preventing overfitting.

2.2.1 Training Loop

```
def train_model(model, train_loader, val_loader, criterion, optimizer, num_epochs=20):
    train_losses, val_losses, train_acc, val_acc = [], [], [], []

    for epoch in range(num_epochs):
        # --- Training ---
        model.train()
        running_loss, correct, total = 0, 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() * images.size(0)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_train_loss = running_loss / total
        epoch_train_acc = 100 * correct / total
        train_losses.append(epoch_train_loss)
        train_acc.append(epoch_train_acc)

        # --- Validation ---
        model.eval()
        val_running_loss, val_correct, val_total = 0, 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_running_loss += loss.item() * images.size(0)
                _, predicted = torch.max(outputs, 1)
                val_total += labels.size(0)
                val_correct += (predicted == labels).sum().item()

        epoch_val_loss = val_running_loss / val_total
        epoch_val_acc = 100 * val_correct / val_total
        val_losses.append(epoch_val_loss)
        val_acc.append(epoch_val_acc)

        print(f"Epoch {epoch+1}/{num_epochs}: "
              f"Train Loss: {epoch_train_loss:.4f}, Train Acc: {epoch_train_acc:.2f}% | "
              f"Val Loss: {epoch_val_loss:.4f}, Val Acc: {epoch_val_acc:.2f}%")

    return train_losses, val_losses, train_acc, val_acc
```

The figure below shows the loss and the validation accuracy across 20 epochs.

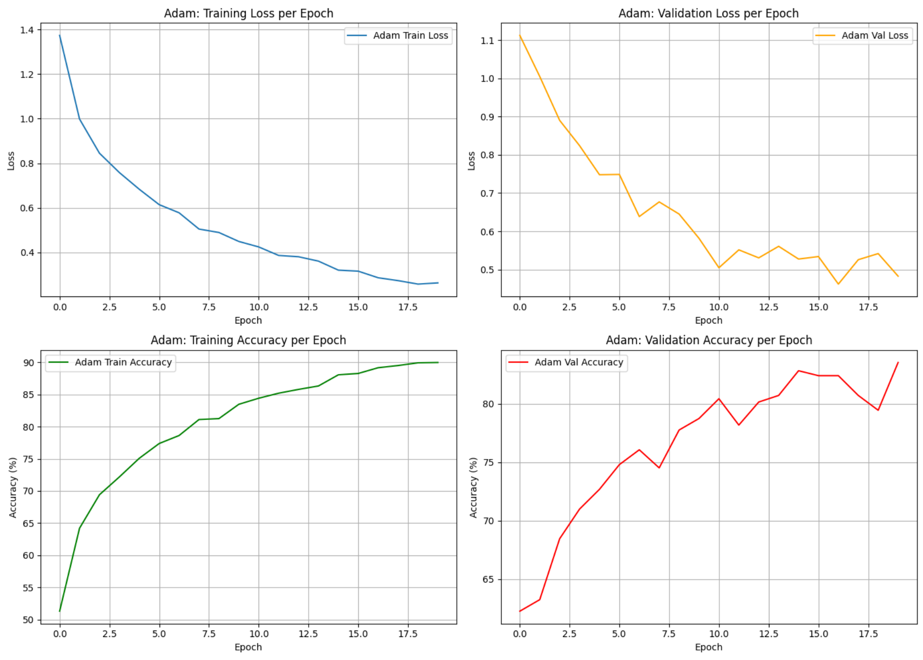


Figure 1: Training and validation accuracies across 20 epochs

Epoch	Train Loss	Train Accuracy (%)	Val Loss	Val Accuracy (%)
1	1.3738	51.30	1.1125	62.25
2	0.9985	64.20	1.0050	63.24
3	0.8452	69.41	0.8902	68.45
4	0.7583	72.17	0.8241	70.99
5	0.6834	75.07	0.7480	72.68
6	0.6143	77.37	0.7488	74.79
7	0.5778	78.61	0.6386	76.06
8	0.5048	81.10	0.6769	74.51
9	0.4893	81.24	0.6451	77.75
10	0.4491	83.47	0.5816	78.73
11	0.4245	84.41	0.5046	80.42
12	0.3864	85.19	0.5515	78.17
13	0.3804	85.79	0.5305	80.14
14	0.3608	86.32	0.5607	80.70
15	0.3204	88.05	0.5274	82.82
16	0.3157	88.26	0.5339	82.39
17	0.2862	89.16	0.4619	82.39
18	0.2733	89.50	0.5257	80.70
19	0.2577	89.92	0.5417	79.44
20	0.2631	89.97	0.4826	83.52

Table 2: Training and validation performance of the CNN model using Adam optimizer over 20 epochs.

2.3 Justifications for Optimizer and Learning Rate

2.3.1 Optimizer Selection

The Adam optimizer was used for training the model. Adam combines the advantages of both AdaGrad and RMSProp by adapting the learning rate for each parameter individually and using momentum to accelerate convergence. This makes it well suited for training deep networks efficiently and handling sparse gradients.

2.3.2 Learning Rate Selection

The learning rate was chosen through experimentation, starting with a small value (2×10^{-5}) to ensure stable training. A learning rate scheduler was also used to reduce the learning rate gradually during training, helping the model converge more effectively and avoid overshooting minima.

2.4 Optimizer Comparison

The CNN model was trained using three optimizers: standard Stochastic Gradient Descent (SGD), SGD with Momentum and Adam. The performance of each optimizer was evaluated using the following metrics:

- **Training Loss and Accuracy:** to monitor learning progress and convergence.
- **Validation Accuracy:** to check generalization and detect overfitting.
- **Test Accuracy and Loss:** to evaluate final model performance on unseen data.

Justification for Metrics: Accuracy and loss were chosen because they directly reflect how well the model predicts the correct class labels. Validation accuracy is particularly important to ensure the model generalizes beyond the training data while test accuracy confirms the final performance on unseen data.

2.4.1 Final Validation and Test Performance

Optimizer	**Final Train Accuracy (%)**	**Validation Accuracy (%)**	**Test Accuracy (%)**
Adam	89.97	83.52	84.42
SGD	91.47	80.85	83.73
SGD with Momentum	94.42	83.66	86.09

Table 3: Comparison of optimizers on the CNN model.

2.4.2 Analysis

- **SGD:** Achieved reasonable training and test accuracy but converged more slowly and had slightly lower validation performance.
- **SGD with Momentum:** Showed faster convergence and the highest test accuracy, thanks to momentum which helps accelerate learning in relevant directions and dampens oscillations.
- **Adam:** Provided stable and efficient convergence with strong validation performance, slightly lower than SGD with Momentum on test accuracy. Its adaptive learning rates make it robust for training deep networks.

The plots obtained are given below.

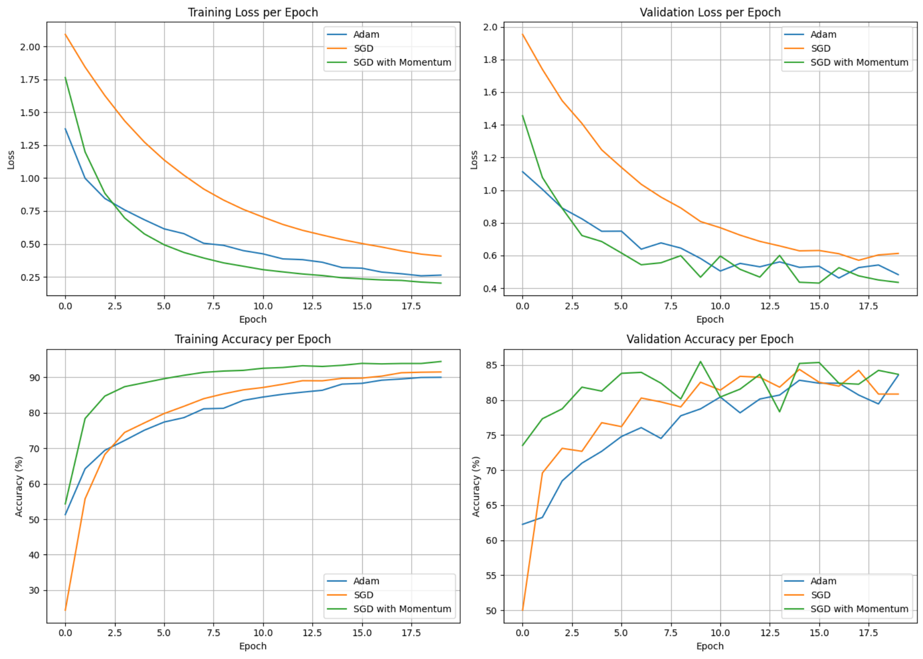


Figure 2: Training and validation accuracies across 20 epochs

2.5 Impact of the Momentum Parameter

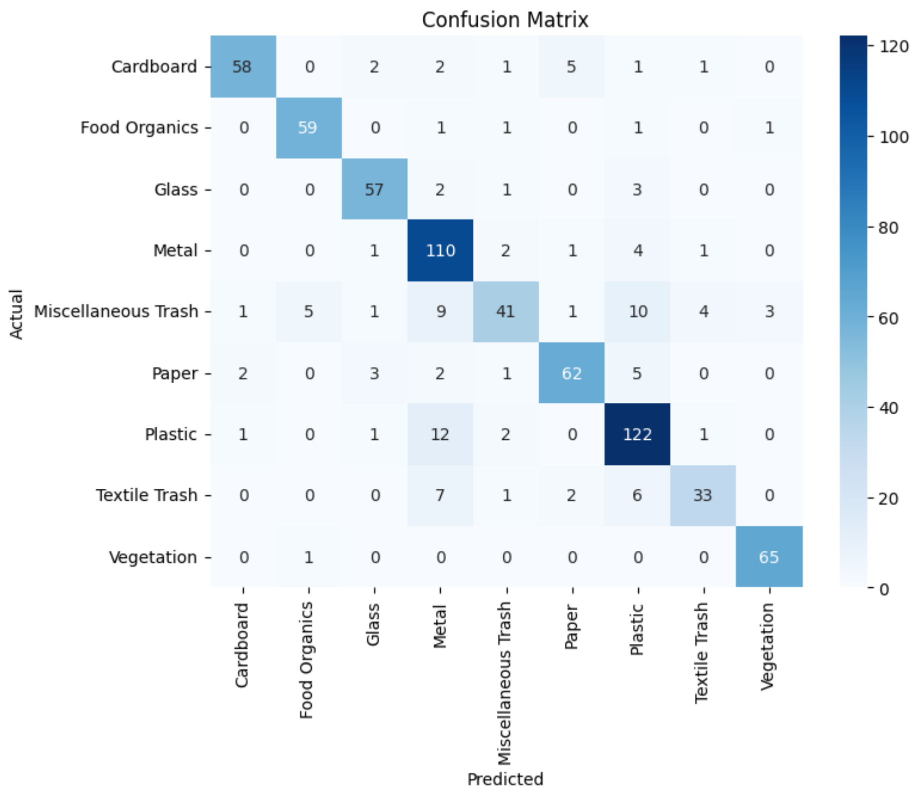
Momentum in optimizers helps the model move faster in the right direction by using past gradients. This reduces oscillations and improves convergence. In our experiments, SGD with momentum achieved higher validation and test accuracy than standard SGD, showing more stable and efficient training.

2.6 Model Evaluation

After training and validating the custom model, its performance was evaluated on the unseen testing dataset to measure generalization capability. The evaluation metrics considered include accuracy, precision, recall, F1-score, and the confusion matrix. These metrics provide a comprehensive understanding of model performance across all classes.

Table 4: Classification Report for Adam Optimizer

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Cardboard	0.94	0.83	0.88	70
Food Organics	0.91	0.94	0.92	63
Glass	0.88	0.90	0.89	63
Metal	0.76	0.92	0.83	119
Miscellaneous Trash	0.82	0.55	0.66	75
Paper	0.87	0.83	0.85	75
Plastic	0.80	0.88	0.84	139
Textile Trash	0.82	0.67	0.74	49
Vegetation	0.94	0.98	0.96	66
Accuracy			**0.84**	719
Macro Avg	0.86	0.83	0.84	719
Weighted Avg	0.85	0.84	0.84	719



Actual \ Predicted	Cardboard	Food Organics	Glass	Metal	Miscellaneous Trash	Paper	Plastic	Textile Trash	Vegetation
Cardboard	58	0	2	2	1	5	1	1	0
Food Organics	0	59	0	1	1	0	1	0	1
Glass	0	0	57	2	1	0	3	0	0
Metal	0	0	1	110	2	1	4	1	0
Miscellaneous Trash	1	5	1	9	41	1	10	4	3
Paper	2	0	3	2	1	62	5	0	0
Plastic	1	0	1	12	2	0	122	1	0
Textile Trash	0	0	0	7	1	2	6	33	0
Vegetation	0	1	0	0	0	0	0	0	65

Figure 3: The Confusion Matrix with the ADAM Optimizer

3 Transfer Learning with State-of-the-Art Networks

3.1 Chosen Architectures

In this study, we have selected **ResNet-18** and **VGG-16** as the two state-of-the-art pretrained convolutional neural network architectures for transfer learning. Both models were originally trained on the **ImageNet** dataset, which contains over 1.2 million labeled images across 1000 categories, providing them with strong feature extraction capabilities.
VGG-16, developed by the Visual Geometry Group (VGG) at the University of Oxford, is known for its simple and uniform architecture, consisting of 16 weight layers using small 3×3 convolution filters. Its deep yet straightforward structure makes it an excellent baseline for visual recognition tasks.
ResNet-18, a variant of the Residual Network (ResNet) family introduced by Microsoft Research, incorporates residual connections (or skip connections) that help overcome the vanishing gradient problem, enabling much deeper networks to be trained effectively. Despite being smaller compared to deeper ResNet variants, ResNet-18 achieves a strong balance between accuracy and computational efficiency.
Both architectures are widely adopted in transfer learning applications due to their proven performance and generalization ability on diverse image classification and feature extraction tasks.

3.2 Model Training

To train the selected pretrained architectures, **ResNet-18** and **VGG-16**, transfer learning was applied using the PyTorch framework. Both models were fine tuned on the prepared real waste image dataset with previous training and validation splits. The models were optimized using the Adam optimizer and trained for 20 epochs with a learning rate of 1×10^{-4}. Early stopping and learning rate scheduling were also implemented to improve convergence and prevent overfitting.

3.2.1 Data Augmentation and Loading

To enhance the generalization of the models and prevent overfitting, data augmentation techniques such as random cropping, rotation, color jittering and horizontal flipping were applied to the training images. Validation images were resized and center cropped for evaluation consistency.

```python
data_transforms = {
    "train": transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(15),
        transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
    "val": transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
}
```

3.2.2 Model Definition

The pretrained ResNet-18 and VGG-16 models were loaded from `torchvision.models`. The final fully connected layers were replaced with custom layers containing a dropout and a linear output layer to match the number of target classes in the dataset.

```python
def get_model(model_name, num_classes, fine_tune=True, dropout_p=0.5):
    if model_name == "resnet":
        model = models.resnet18(pretrained=True)
        in_features = model.fc.in_features
        model.fc = nn.Sequential(
            nn.Dropout(dropout_p),
            nn.Linear(in_features, num_classes)
        )

    elif model_name == "vgg":
        model = models.vgg16(pretrained=True)
        in_features = model.classifier[6].in_features
        model.classifier[6] = nn.Sequential(
            nn.Dropout(dropout_p),
            nn.Linear(in_features, num_classes)
        )
    return model.to(DEVICE)
```

3.2.3 Training Procedure

A generalized training function was implemented to train and validate the models in each epoch. The function included learning rate scheduling using `StepLR`, early stopping to avoid overfitting and tracking of both loss and accuracy metrics.

```python
def train_model(model, criterion, optimizer, scheduler, num_epochs=20, patience=5):
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    early_stop_counter = 0

    for epoch in range(num_epochs):
        for phase in ["train", "val"]:
            model.train() if phase == "train" else model.eval()
            running_loss, running_corrects = 0.0, 0

            for inputs, labels in dataloaders[phase]:
                inputs, labels = inputs.to(DEVICE), labels.to(DEVICE)
                optimizer.zero_grad()

                with torch.set_grad_enabled(phase == "train"):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)
                    if phase == "train":
                        loss.backward()
                        optimizer.step()

                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            if phase == "train":
                scheduler.step()

            epoch_acc = running_corrects.double() / dataset_sizes[phase]
            if phase == "val" and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())
                early_stop_counter = 0
            elif phase == "val":
                early_stop_counter += 1

        if early_stop_counter >= patience:
            print("Early stopping triggered.")
            break

    model.load_state_dict(best_model_wts)
    return model
```

3.2.4 Model Fine Tuning and Optimization

Both models were fine tuned using identical optimization configurations. The `CrossEntropyLoss` function was used as the loss criterion and the learning rate was reduced by a factor of 0.1 every 10 epochs using the `StepLR` scheduler.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-4)
scheduler = StepLR(optimizer, step_size=10, gamma=0.1)

model, train_loss, val_loss, train_acc, val_acc = train_model(
    model, criterion, optimizer, scheduler, num_epochs=20)
```

The use of transfer learning allowed faster convergence and improved accuracy with limited training data. Early stopping and learning rate scheduling helped stabilize training and prevented overfitting.

3.3 Training and Validation Losses

To evaluate the performance of the selected pretrained models, both **ResNet-18** and **VGG-16** were fine tuned on the dataset for 20 epochs. During training, the *training loss*, *validation loss*, *training accuracy* and *validation accuracy* were recorded for each epoch. The results are summarized in the following tables.

3.3.1 ResNet-18 Training Results

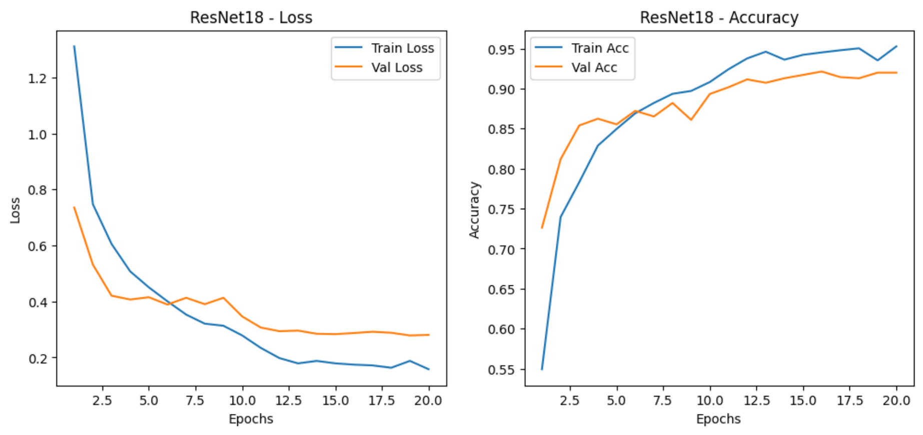


Figure 4: Training and Validation Accuracy/Loss Curves for ResNet-18

Table 5: Training and Validation Loss/Accuracy for ResNet-18

Epoch	**Train Loss**	**Train Acc**	**Val Loss**	**Val Acc**
1	1.3107	0.5492	0.7353	0.7261
2	0.7472	0.7394	0.5315	0.8118
3	0.6055	0.7830	0.4205	0.8539
4	0.5074	0.8288	0.4069	0.8624
5	0.4504	0.8495	0.4150	0.8553
6	0.4004	0.8691	0.3889	0.8722
7	0.3532	0.8820	0.4130	0.8652
8	0.3206	0.8935	0.3902	0.8820
9	0.3132	0.8971	0.4130	0.8610
10	0.2789	0.9082	0.3468	0.8933
11	0.2343	0.9242	0.3069	0.9017
12	0.1978	0.9377	0.2937	0.9115
13	0.1789	0.9461	0.2959	0.9073
14	0.1876	0.9362	0.2845	0.9129
15	0.1790	0.9422	0.2833	0.9171
16	0.1743	0.9452	0.2872	0.9213
17	0.1715	0.9479	0.2918	0.9143
18	0.1633	0.9503	0.2881	0.9129
19	0.1877	0.9353	0.2784	0.9199
20	0.1580	0.9528	0.2806	0.9199

The model achieved the highest validation accuracy of **92.13%** at epoch 16, indicating strong convergence and generalization capability. The loss values show a consistent downward trend, confirming stable optimization.

3.3.2 VGG-16 Training Results

Table 6: Training and Validation Loss/Accuracy for VGG-16

Epoch	**Train Loss**	**Train Acc**	**Val Loss**	**Val Acc**
1	1.5859	0.4048	0.9529	0.6713
2	0.9869	0.6590	0.7281	0.7570
3	0.8258	0.7099	0.7298	0.7654
4	0.7067	0.7454	0.5780	0.7963
5	0.6147	0.7866	0.5539	0.8202
6	0.5426	0.8164	0.5716	0.8174
7	0.5114	0.8255	0.5959	0.8132
8	0.4811	0.8354	0.4940	0.8287
9	0.4396	0.8432	0.4113	0.8525
10	0.4113	0.8628	0.4528	0.8469
11	0.2852	0.8995	0.3341	0.8778
12	0.2307	0.9187	0.3112	0.8848
13	0.2129	0.9248	0.3078	0.8848
14	0.1878	0.9335	0.2965	0.8975
15	0.1866	0.9356	0.2971	0.8947
16	0.1769	0.9356	0.2837	0.9059
17	0.1693	0.9446	0.3128	0.8975
18	0.1677	0.9416	0.2991	0.8961
19	0.1668	0.9419	0.2957	0.8947
20	0.1662	0.9416	0.2904	0.9101

The **VGG-16** model reached its best validation accuracy of **91.01%** at epoch 20. Although it required longer training time compared to ResNet-18, it demonstrated strong performance with gradual improvement across epochs.

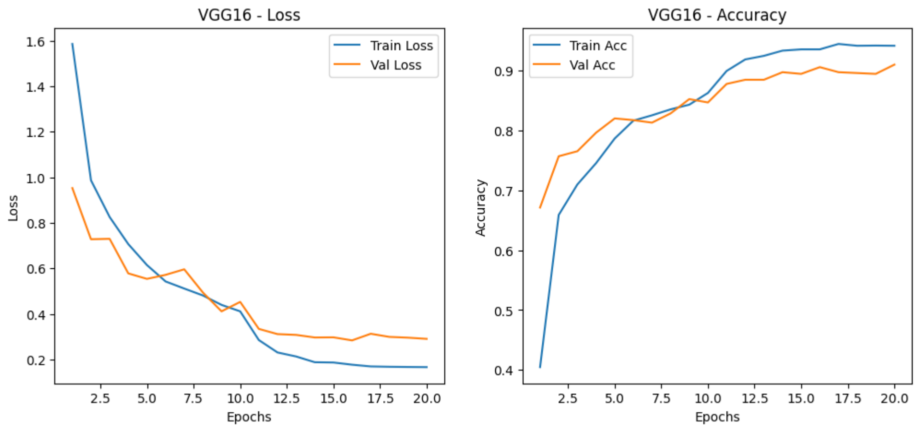


Figure 5: Training and Validation Accuracy/Loss Curves for VGG-16

3.4 Model Evaluation

After fine tuning the pretrained models (**ResNet-18** and **VGG-16**), their performance was evaluated on the unseen testing dataset to measure generalization capability. The evaluation metrics considered include *accuracy*, *precision*, *recall*, *F1-score*, and the *confusion matrix*. These metrics provide a comprehensive understanding of model performance across all classes.

3.4.1 ResNet-18 Evaluation Results

The fine tuned ResNet-18 model achieved a **test accuracy of 90.93%**. Table 7 summarizes the macro and weighted average performance metrics.

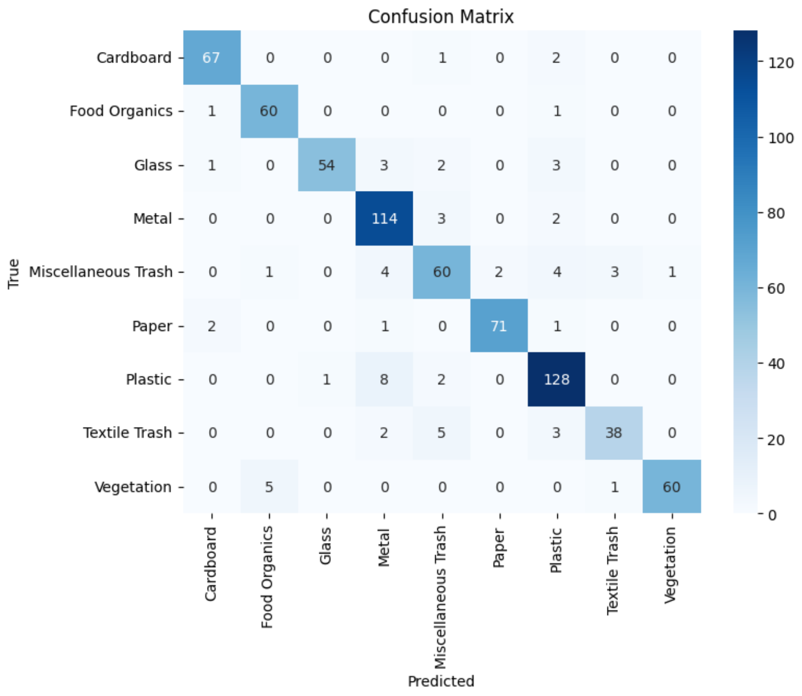
Table 7: Performance Metrics for ResNet-18 on Test Dataset

Metric	**Macro Average**	**Weighted Average**
Precision	0.9189	0.9115
Recall	0.9009	0.9093
F1-Score	0.9084	0.9091
Accuracy	0.9093 (90.93%)	

The detailed class wise classification report is given below:

Table 8: Class-wise Evaluation Results for ResNet-18

Class	**Precision**	**Recall**	**F1-Score**
Cardboard	0.94	0.96	0.95
Food Organics	0.91	0.97	0.94
Glass	0.98	0.86	0.92
Metal	0.86	0.96	0.91
Miscellaneous Trash	0.82	0.80	0.81
Paper	0.97	0.95	0.96
Plastic	0.89	0.92	0.90
Textile Trash	0.90	0.79	0.84
Vegetation	0.98	0.91	0.94



True \ Predicted	Cardboard	Food Organics	Glass	Metal	Miscellaneous Trash	Paper	Plastic	Textile Trash	Vegetation
Cardboard	67	0	0	0	1	0	2	0	0
Food Organics	1	60	0	0	0	0	1	0	0
Glass	1	0	54	3	2	0	3	0	0
Metal	0	0	0	114	3	0	2	0	0
Miscellaneous Trash	0	1	0	4	60	2	4	3	1
Paper	2	0	0	1	0	71	1	0	0
Plastic	0	0	1	8	2	0	128	0	0
Textile Trash	0	0	0	2	5	0	3	38	0
Vegetation	0	5	0	0	0	0	0	1	60

Figure 6: Confusion Matrix for ResNet-18 Model

The confusion matrix demonstrates that ResNet-18 effectively distinguishes between most classes, with minor confusion between visually similar waste categories such as plastic and metal.

3.4.2 VGG-16 Evaluation Results

The fine tuned VGG-16 model achieved a **test accuracy of 90.10%**. The macro and weighted average metrics are summarized in Table 9.

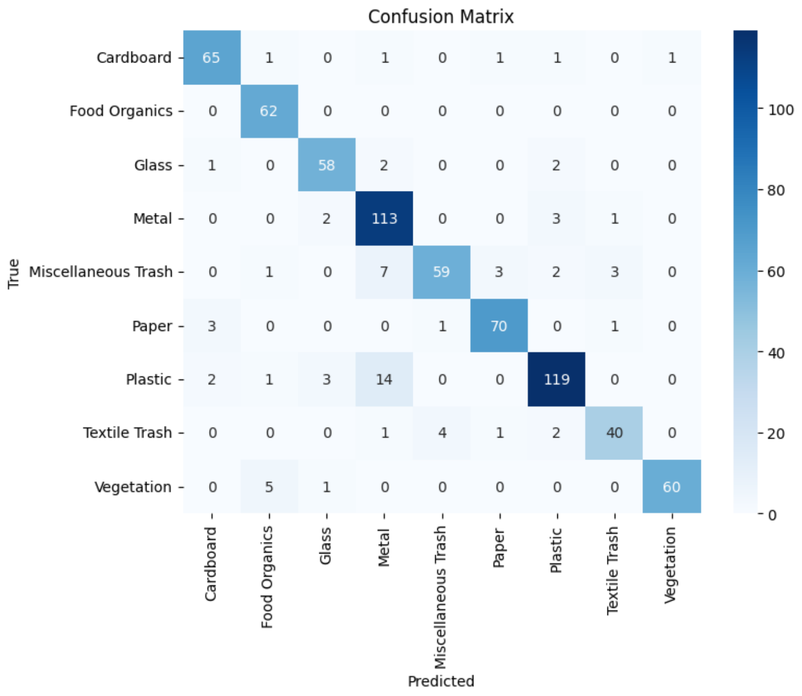
Table 9: Performance Metrics for VGG-16 on Test Dataset

Metric	Macro Average	Weighted Average
Precision	0.9085	0.9044
Recall	0.9019	0.9010
F1-Score	0.9033	0.9006
Accuracy	0.9010 (90.10%)	

The detailed class wise classification report for VGG-16 is presented below:

Table 10: Class-wise Evaluation Results for VGG-16

Class	Precision	Recall	F1-Score
Cardboard	0.92	0.93	0.92
Food Organics	0.89	1.00	0.94
Glass	0.91	0.92	0.91
Metal	0.82	0.95	0.88
Miscellaneous Trash	0.92	0.79	0.85
Paper	0.93	0.93	0.93
Plastic	0.92	0.86	0.89
Textile Trash	0.89	0.83	0.86
Vegetation	0.98	0.91	0.94



True \ Predicted	Cardboard	Food Organics	Glass	Metal	Miscellaneous Trash	Paper	Plastic	Textile Trash	Vegetation
Cardboard	65	1	0	1	0	1	1	0	1
Food Organics	0	62	0	0	0	0	0	0	0
Glass	1	0	58	2	0	0	2	0	0
Metal	0	0	2	113	0	0	3	1	0
Miscellaneous Trash	0	1	0	7	59	3	2	3	0
Paper	3	0	0	0	1	70	0	1	0
Plastic	2	1	3	14	0	0	119	0	0
Textile Trash	0	0	0	1	4	1	2	40	0
Vegetation	0	5	1	0	0	0	0	0	60

Figure 7: Confusion Matrix for VGG-16 Model

The VGG-16 model also performed well across all classes, achieving slightly lower overall accuracy compared to ResNet-18. However, its results were consistent and it demonstrated particularly strong recall for the *Food Organics* class.

3.4.3 Summary

Overall, both models achieved over **90% test accuracy**, indicating robust generalization and effective transfer learning. ResNet-18 showed slightly superior macro precision and recall, while VGG-16 performed comparably with more stable learning trend.

4 Model Comparison and Discussion

4.1 Comparison of Custom vs. State-of-the-Art Models

The performance of the custom CNN model was compared with two fine tuned state-of-the-art architectures, ResNet-18 and VGG-16, to evaluate its effectiveness on the RealWaste dataset. The comparison was based on test accuracy, precision, recall, F1-score and class wise performance metrics.

Table 11: Comparison of Custom CNN with ResNet-18 and VGG-16 on Test Dataset

Model	**Test Accuracy (%)**	**Macro Avg F1-Score**	**Weighted Avg F1-Score**
Custom CNN (Adam)	84.42	0.84	0.84
ResNet-18	90.93	0.9084	0.9091
VGG-16	90.10	0.9033	0.9006

From the results:

- The fine tuned **ResNet-18** achieved the highest test accuracy of **90.93%**, outperforming both the custom CNN (**84.42%**) and VGG-16 (**90.10%**).

- Both ResNet-18 and VGG-16 also showed higher macro and weighted average F1-scores, indicating better overall class wise performance and generalization.
- Class wise evaluation revealed that the custom CNN underperformed on challenging classes such as **Miscellaneous Trash** and **Textile Trash**, whereas the state-of-the-art models demonstrated more balanced performance across all categories.
- The superior performance of ResNet-18 can be attributed to its **residual connections**, which help mitigate vanishing gradients and allow training of deeper networks, while VGG-16 benefits from **a deeper convolutional architecture with larger receptive fields**.

Overall, while the custom CNN provides a relatively good performance with fewer parameters and a simpler architecture, fine tuned state-of-the-art models, especially ResNet-18, achieve significantly better accuracy and more consistent class wise predictions, making them preferable for high accuracy waste classification tasks.

4.2 Trade offs and Limitations of Transfer Learning

4.2.1 Advantages of Custom CNN:

- **Simplicity:** Custom CNNs are easier to design and modify according to specific dataset characteristics.
- **Fewer Parameters:** Requires less memory and computational resources, making it suitable for resource constrained environments.
- **Faster Training:** Training a small custom model is quicker compared to deep pre trained networks.

4.2.2 Limitations of Custom CNN:

- **Lower Accuracy:** Generally achieves lower performance, especially on complex datasets compared to deep pre trained networks.
- **Limited Generalization:** May struggle with diverse or unseen data due to its simpler architecture.
- **Manual Design Effort:** Requires careful design and tuning of layers, filters and hyperparameters.

4.2.3 Advantages of Pre-trained Models:

- **High Accuracy:** Leveraging large scale pre training provides better feature extraction and overall performance.
- **Better Generalization:** Pre trained weights allow the model to generalize well even with limited task specific data.
- **Reduced Design Effort:** Architectural design is already optimized, reducing experimentation time.

4.2.4 Limitations of Pre trained Models:

- **Higher Computational Cost:** Require more memory, GPU power and training time for fine tuning.
- **Complexity:** Larger architectures are harder to debug and modify for specific needs.
- **Overfitting Risk:** Without careful regularization, fine tuning on small datasets may lead to overfitting.

4.2.5 Trade offs:

The choice depends on the application constraints. Custom CNNs are suitable when resources are limited or fast prototyping is needed, whereas pre trained models are preferred when accuracy and robust generalization are the primary goals.

5 Appendix

The Github repository containing all the codes and plots can be found in the following link `https://github.com/lahirunie-dulsara/EN3150-Assignment-3-CNN`.

References

[1] UCI Machine Learning Repository, “RealWaste Dataset,” *UC Irvine*, 2023. [Online]. Available: `https://archive.ics.uci.edu/dataset/908/realwaste`.

[2] “MIT:Convolutional Neural Networks,” *YouTube*, Apr. 2023. [Online]. Available: `https://www.youtube.com/watch?v=NmLK_WQBxB4&list=PLtBw6njQRU-rwp5__7C0oIVt26ZgjG9NI&index=4`.

[3] PyTorch, “ResNet — PyTorch Vision Models,” *PyTorch Hub*, 2024. [Online]. Available: `https://pytorch.org/hub/pytorch_vision_resnet/`.

[4] PyTorch, “VGG16 — Torchvision Models,” *PyTorch Documentation*, 2024. [Online]. Available: `https://docs.pytorch.org/vision/main/models/generated/torchvision.models.vgg16.html`.

[5] PyTorch, “Deep Learning with PyTorch: A 60 Minute Blitz,” *PyTorch Tutorials*, 2024. [Online]. Available: `https://docs.pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html`..

[6] PyTorch, “Torchvision Models,” *PyTorch Documentation*, 2024. [Online]. Available: `https://docs.pytorch.org/vision/main/models.html`.