A PROJECT REPORT

on

"Waste Classification Using Deep Learning: A Comparative Analysis"

Submitted to

KIIT Deemed to be University

In Partial Fulfillment of the Requirement for the Award of

BACHELOR'S DEGREE IN COMPUTER SCIENCE & ENGINEERING

BY

Satyabrata Sahoo	**22051971**
Saswat Chandan	**22052150**
Aman Raj	**22052273**
Archit Anand	**22052282**
Kushagra Dhari	**22052308**
Lokesh Sahu	**22052310**

UNDER THE GUIDANCE OF

Prof. Sourajit Behera

SCHOOL OF COMPUTER ENGINEERING

KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY

BHUBANESWAR, ODISHA - 751024

March 2025

KIIT Deemed to be University

School of Computer Engineering
Bhubaneswar, ODISHA 751024

CERTIFICATE

This is to certify that the project titled

"Waste Classification Using Deep Learning: A Comparative Analysis"

submitted by

Satyabrata Sahoo	**22051971**
Saswat Chandan	**22052150**
Aman Raj	**22052273**
Archit Anand	**22052282**
Kushagra Dhari	**22052308**
Lokesh Sahu	**22052310**

is a record of bonafide work carried out by them, in the partial fulfillment of the requirement for the award of Degree of Bachelor of Technology (Computer Science & Engineering) at KIIT Deemed to be university, Bhubaneswar. This work is done during year 2024-2025, under my guidance.

Date: 12 /03 /25

Prof. Sourajit Behera

Acknowledgement

We are profoundly grateful of Prof. Sourajit Behera for his expert guidance and continuous encouragement throughout the project right from its commencement to its completion.

Satyabrata Sahoo
Saswat Chandan
Aman Raj
Archit Anand
Kushagra Dhari
Lokesh Sahu

School of Computer Engineering, KIIT, BBSR

Abstract

Waste management is a growing challenge, and efficient classification is key to a sustainable future. Traditional sorting methods rely on manual labor, making them slow, costly, and prone to errors. Automating this process using deep learning can significantly improve efficiency and accuracy. This study compares a custom Convolutional Neural Network (CNN) with three well-established pre-trained models: ResNet50, MobileNetV3, and EfficientNetB3, to identify the best approach for waste classification.

We trained these models on a dataset of 1,987 labeled images across five waste categories: plastic, paper, glass, organic, and textile. The custom CNN achieved 76% accuracy, showing that while it captures basic patterns, it lacks the depth needed for high precision. In contrast, ResNet50 performed the best with 96% accuracy, followed by MobileNetV3 (93%) and EfficientNetB3 (91%), proving that transfer learning significantly improves classification performance.

Each model comes with trade-offs. ResNet50 offers the highest accuracy but requires substantial computational power, making it best suited for cloud-based systems. MobileNetV3 balances accuracy and efficiency, making it a great option for mobile and edge devices, while EfficientNetB3 provides strong performance with optimized resource usage.

This project highlights how deep learning can make waste classification faster, more reliable, and scalable. Choosing the right model depends on the deployment environment, whether in the cloud or on a lightweight device. These insights help move us toward smarter, more sustainable waste management solutions.

Keywords: Waste Classification, Deep Learning, CNN, ResNet50, MobileNetV3, EfficientNetB3, Transfer Learning, Image Classification

Content

1. Introduction

Waste generation has increased exponentially over the past few decades, posing significant environmental and economic challenges. With urbanization and industrial growth, waste management has become a critical global issue. Poor waste disposal leads to severe environmental hazards, including land pollution, water contamination, and increased greenhouse gas emissions. Efficient waste classification is a fundamental step in promoting sustainable recycling and waste management practices.

Traditional waste sorting methods rely on manual labor, which is time-consuming, costly, and prone to human error. In large-scale waste processing facilities, sorting accuracy is crucial for optimizing recycling processes and reducing landfill accumulation. However, manual sorting is neither scalable nor efficient, making automation a necessity.

1.1 Deep Learning for Waste Classification

Deep learning, particularly Convolutional Neural Networks (CNNs), has revolutionized image classification by automatically extracting complex visual features from raw images. Unlike conventional machine learning approaches that require handcrafted feature engineering, CNNs learn patterns, textures, and structures from images, making them highly effective for classification tasks. By leveraging transfer learning, models pre-trained on large-scale image datasets can be fine-tuned for specific applications, such as waste classification.

1.2 Project Scope and Objectives

This project explores the application of deep learning models for automated waste classification. A custom CNN model is developed and benchmarked against well-established pre-trained models ResNet50, MobileNetV3, and EfficientNetB3. The goal is to evaluate their effectiveness based on:

Classification accuracy – How well each model distinguishes between different waste categories.

Computational efficiency – The resources required for training and inference.

Deployment feasibility – Suitability for real-world applications, including cloud and edge-device deployment.

1.3 Significance of the Project

Automated waste classification can significantly improve recycling efficiency, reduce processing costs, and minimize environmental impact. By comparing different CNN architectures, this project aims to provide valuable insights into the trade-offs between accuracy and efficiency. The findings can help guide future research and practical implementations, contributing to the development of sustainable and intelligent waste management systems.

2. Background

2.1 Convolutional Neural Networks (CNNs)

CNNs have revolutionized image classification by automatically learning hierarchical patterns from visual data. The architecture is composed of several essential components:

Convolutional Layers: Extract low-level and high-level features such as edges, textures, and patterns.

Pooling Layers: Downsample feature maps to reduce dimensionality while preserving important information.

Fully Connected Layers: Transform extracted features into a final classification output.

Activation Functions: ReLU introduces non-linearity to improve learning efficiency, while Softmax normalizes outputs into probabilities.

2.1.1 Custom CNN

The custom CNN model serves as a foundational architecture, featuring multiple convolutional layers to extract hierarchical features, max-pooling layers to reduce spatial dimensions, and fully connected layers for classification. Batch normalization is applied to stabilize training, and dropout layers are included to prevent overfitting.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.layers import BatchNormalization

# Define the model architecture
custom_cnn = Sequential([
    # First Convolution Block
    Conv2D(32, (3,3), activation='relu', padding='same', input_shape=(224, 224, 3)),
    BatchNormalization(),
    Conv2D(32, (3,3), activation='relu', padding='same'),  # Extra layer
    BatchNormalization(),
    MaxPooling2D(pool_size=(2,2)),

    # Second Convolution Block
    Conv2D(64, (3,3), activation='relu', padding='same'),
    BatchNormalization(),
    Conv2D(64, (3,3), activation='relu', padding='same'),  # Extra layer
    BatchNormalization(),
    MaxPooling2D(pool_size=(2,2)),

    # Third Convolution Block
    Conv2D(128, (3,3), activation='relu', padding='same'),
    BatchNormalization(),
    Conv2D(128, (3,3), activation='relu', padding='same'),  # Extra layer
    BatchNormalization(),
    MaxPooling2D(pool_size=(2,2)),

    # Fourth Convolution Block
    Conv2D(256, (3,3), activation='relu', padding='same'),
    BatchNormalization(),
    Conv2D(256, (3,3), activation='relu', padding='same'),  # Extra layer
    BatchNormalization(),
    MaxPooling2D(pool_size=(2,2)),

    # Flatten to Fully Connected Layers
```

```
    Flatten(),
    Dense(512, activation='relu'),
    BatchNormalization(),
    Dropout(0.4),
    Dense(256, activation='relu'),
    BatchNormalization(),
    Dropout(0.4),
    Dense(128, activation='relu'),
    Dropout(0.5),  # Increased dropout to prevent overfitting

    # Output layer (Auto-detect number of classes)
    Dense(len(train_generator.class_indices), activation='softmax')
])
```

2.2 Transfer Learning Models

Transfer learning is a machine learning technique where a pre-trained model, trained on a large dataset, is fine-tuned to solve a related problem with a smaller dataset. This approach significantly reduces training time and computational cost while improving accuracy. The advantage of transfer learning is that it allows leveraging previously learned feature representations, which are highly effective for complex image classification tasks.

Three such deep learning architectures was used for waste classification in this project. Each of these models has unique properties that balance accuracy, computational efficiency, and deployment feasibility. The following architectures were implemented:

2.2.1 ResNet50

ResNet50 is a powerful deep learning model that uses residual connections to make training deep networks more efficient. These shortcuts help prevent issues like vanishing gradients, making it ideal for extracting detailed features from images[1].

```
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.layers import GlobalAveragePooling2D, Dropout, Dense
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.models import Model

# Load Pretrained ResNet50 Model (Without Fully Connected Layers)
base_model = ResNet50(
    include_top=False,
    weights='imagenet',
    input_shape=(224, 224, 3))
# Freeze all layers up to a specific block (Conv4)
for layer in base_model.layers[:143]:  # Freeze layers up to conv4_block6_out
   layer.trainable = False
# Build the New Model
x = base_model.output
x = GlobalAveragePooling2D()(x)  # Pooling layer
x = BatchNormalization()(x)  # Normalize activations
x = Dropout(0.5)(x)  # Dropout for regularization
x = Dense(256, activation='relu')(x)  # Fully connected layer
x = Dropout(0.3)(x)  # Extra dropout
x = Dense(5, activation='softmax')(x)  # Output layer
# Define the Transfer Learning Model
transfer_resnet50_model = Model(inputs=base_model.input, outputs=x)
```

2.2.2 MobileNetV3

MobileNetV3 is designed for mobile and edge devices, offering a balance between speed and accuracy. It uses depth-wise separable convolutions and squeeze-and-excitation modules to reduce computation without compromising performance. This makes it perfect for real-time applications where processing power is limited[2].

```
from tensorflow.keras.applications import MobileNetV3Large
from tensorflow.keras.models import Model
from tensorflow.keras.layers import GlobalAveragePooling2D, Dropout, Dense
from tensorflow.keras.layers import BatchNormalization

# Load MobileNetV3 Large with ImageNet weights
base_model = MobileNetV3Large(input_shape=(224, 224, 3), include_top=False,
weights="imagenet")

base_model.trainable = False  # Freeze convolutional base

# Add custom classification head
x = GlobalAveragePooling2D()(base_model.output)
x = Dropout(0.3)(x)  # Dropout for regularization
x = Dense(128, activation='relu')(x)
x = BatchNormalization()(x)  # Improve stability
x = Dropout(0.3)(x)

output = Dense(NUM_CLASSES, activation='softmax')(x)  # Output layer
mobilenetv3_model = Model(inputs=base_model.input, outputs=output)
```

2.2.3 EfficientNetB3

EfficientNetB3 is a model that optimizes depth, width, and resolution simultaneously using a compound scaling method. It enhances feature extraction while keeping computational costs low. With its high accuracy and efficiency, EfficientNetB3 is ideal for cloud-based and large-scale classification applications[3].

```
from tensorflow.keras.applications import EfficientNetB3
from tensorflow.keras.models import Model
from tensorflow.keras.layers import GlobalAveragePooling2D, Dropout, Dense
from tensorflow.keras.layers import BatchNormalization

# Load EfficientNetB3 with pre-trained ImageNet weights
base_model = EfficientNetB3(input_shape=(224, 224, 3), include_top=False,
weights="imagenet")

base_model.trainable = False  # Freeze convolutional base

# Add custom classification head
x = GlobalAveragePooling2D()(base_model.output)
x = Dropout(0.3)(x)  # Dropout for regularization
x = Dense(128, activation='relu')(x)
x = BatchNormalization()(x)  # Improve stability
x = Dropout(0.3)(x)

output = Dense(NUM_CLASSES, activation='softmax')(x)  # Output layer
efficientnet_model = Model(inputs=base_model.input, outputs=output)
```

3. Methodology

3.1 Dataset

Our dataset consists of 1987 labeled images categorized into five different waste classes (textile, organic, glass, plastic, and paper). Each image is assigned a label corresponding to its waste type, facilitating supervised learning.

1. First we bring our dataset into coding environment and go through each folder in it to look for number of images and their waste types. Then we check if each image has same dimension and colour channel.

```
import os  # Import os module for file path operations
from PIL import Image  # Import PIL for image processing
import pandas as pd  # Import pandas (though not used, can be useful for data handling)

# Define the path where the dataset is stored
dataset_path = '/content/drive/MyDrive/Colab Notebooks/Waste-classifier/resized_dataset'
# Retrieve the names of all folders (representing garbage types) within the dataset directory
garbage_types = os.listdir(dataset_path)

# Create a set to store unique image dimensions across the dataset
all_dimensions_set = set()

# Iterate over each garbage type (folder) to process images
for garbage_type in garbage_types:
    folder_path = os.path.join(dataset_path, garbage_type)  # Construct folder path

# Verify that the current item is a directory
    if os.path.isdir(folder_path):
        # Get a list of all image files in the folder (filtering for JPG and JPEG formats)
        image_files = [f for f in os.listdir(folder_path) if f.endswith(('jpg', 'jpeg'))]
        # Display the count of images in the current folder
        num_images = len(image_files)
        print(f"{garbage_type} folder contains {num_images} images.")

# Loop over each image to check its dimensions
        for image_file in image_files:
            image_path = os.path.join(folder_path, image_file)  # Get full image path
            with Image.open(image_path) as img:  # Open the image
                # Extract width, height, and color channels (bands)
                width, height = img.size
                channels = len(img.getbands())  # Number of color channels (e.g., RGB = 3)
                # Add the image dimensions to the set (ensures uniqueness)
                all_dimensions_set.add((width, height, channels))

# Check if all images have the same dimensions
if len(all_dimensions_set) == 1:
    width, height, channels = all_dimensions_set.pop()  # Extract the single unique dimension
    print(f"\nAll images in the dataset have the same dimensions: {width}x{height} with {channels}
color channels.")
else:
    print("\nThe images in the dataset have different dimensions or color channels.")
```

Output:
textile folder contains 404 images.
organic folder contains 368 images.
glass folder contains 401 images.
plastic folder contains 408 images.
paper folder contains 406 images.
All images in the dataset have the same dimensions: 224x224 with 3 color channels.

2. Now we create a dataframe to bring all the images scattered across multiple folders for easy manipulation.

```
import os  # Import the os module to work with file paths
import pandas as pd  # Import pandas for handling structured data

# Initialize an empty list to store image file paths and their respective labels
data = []

# Loop through each garbage type and collect its images' file paths
for garbage_type in garbage_types:
# Construct the full path to the garbage type folder
    garbage_path = os.path.join(dataset_path, garbage_type)
    # Iterate through all files in the folder
    for file in os.listdir(garbage_path):
        file_path = os.path.join(garbage_path, file)  # Get the full file path
        data.append((file_path, garbage_type))   # Append the image file path and its
corresponding class label (garbage type) to the data list

# Convert the collected data into a Pandas DataFrame for easy manipulation
df = pd.DataFrame(data, columns=['filepath', 'label'])

# Display the first few entries of the DataFrame to verify data collection
df.head()
```

Output:

	filepath	label
0	/content/drive/MyDrive/Colab Notebooks/Waste-c...	textile
1	/content/drive/MyDrive/Colab Notebooks/Waste-c...	textile
2	/content/drive/MyDrive/Colab Notebooks/Waste-c...	textile
3	/content/drive/MyDrive/Colab Notebooks/Waste-c...	textile
4	/content/drive/MyDrive/Colab Notebooks/Waste-c...	textile

3.2 Preprocessing

To ensure consistency in model input, several preprocessing steps are applied:

1. First we divide our data set Train and Validation set such that they represent all classes well while avoiding potential biases associated with the order of images and have a similar distribution of classes as the whole dataset and also be shuffled.

For that we employ stratified sampling through train_test_split, which inherently shuffles and divides the DataFrame while maintaining a consistent distribution of classes.

```
from sklearn.model_selection import train_test_split  # Import train-test split function

# Split the dataset into training and validation sets using stratification
train_df, val_df = train_test_split(df, test_size=0.2, random_state=42,
stratify=df['label'])

"""
Parameters:
- df: DataFrame containing image file paths and labels.
- test_size=0.2: 20% of the dataset is used for validation.
```

```
- random_state=42: Ensures reproducibility of the split.
- stratify=df['label']: Ensures class distribution is maintained in both sets.
"""

# Print the number of images in each set
print(f"Number of images in the training set: {len(train_df)}")
print(f"Number of images in the validation set: {len(val_df)}")
```

Output:
Number of images in the training set: 1589
Number of images in the validation set: 398

```
# 1. Compute the class distribution for the entire dataset
overall_distribution = df['label'].value_counts(normalize=True) * 100  # Normalize to
get percentage
# 2. Compute the class distribution for the training set
train_distribution = train_df['label'].value_counts(normalize=True) * 100
# 3. Compute the class distribution for the validation set
val_distribution = val_df['label'].value_counts(normalize=True) * 100

# Display the class distributions with rounding for better readability
print("Class distribution in the entire dataset:\n")
print(overall_distribution.round(2))  # Round to two decimal places
print('-' * 40)

print("\nClass distribution in the training set:\n")
print(train_distribution.round(2))
print('-' * 40)

print("\nClass distribution in the validation set:\n")
print(val_distribution.round(2))

# Extract unique class labels from the 'label' column of the training dataset
class_labels = train_df['label'].unique()

# Display the extracted class labels
print("Class Labels:", class_labels)
```

Output:

```
Class distribution in the entire dataset:

label
plastic    20.53
paper      20.43
textile    20.33
glass      20.18
organic    18.52
Name: proportion, dtype: float64
----------------------------------------

Class distribution in the training set:

label
plastic    20.52
paper      20.45
textile    20.33
glass      20.20
organic    18.50
Name: proportion, dtype: float64
----------------------------------------

Class distribution in the validation set:

label
plastic    20.60
textile    20.35
paper      20.35
glass      20.10
organic    18.59
Name: proportion, dtype: float64
Class Labels: ['paper' 'plastic' 'textile' 'glass' 'organic']
```

2. Then we use data augmentation techniques on training set to artificially increase our data and avoid overfitting as we have limited number of data.

Data augmentation technique is not applied on validation set as it's role is to provide an unbiased evaluation of a model's performance on unseen data.

3. We also rescale both training and validation images for better network performance.

```
# Slight Augmentation settings for training
train_datagen = ImageDataGenerator(
    rescale=1./255,                # Normalize pixel values to [0,1]
    rotation_range=45,             # Randomly rotate the images by up to 45 degrees
    width_shift_range=0.15,        # Randomly shift images horizontally by up to 15% of the width
    height_shift_range=0.15,       # Randomly shift images vertically by up to 15% of the height
    zoom_range=0.15,               # Randomly zoom in or out by up to 15%
    horizontal_flip=True,          # Randomly flip images horizontally
    vertical_flip=True,            # Randomly flip images vertically
    shear_range=0.05,              # Apply slight shear transformations
    brightness_range=[0.9, 1.1],   # Vary brightness between 90% to 110% of original
    channel_shift_range=10,        # Randomly shift channels (can change colors of images slightly
                                    but less aggressively)
    fill_mode='nearest'            # Fill in missing pixels using the nearest filled value
)

# Only rescaling for validation
val_datagen = ImageDataGenerator(rescale=1./255)
```

4. Since we can't load all images at once due to memory constraints, we use the method 'flow_from_dataframe' to generate batches of images and labels directly from our DataFrame.

```
# Using flow_from_dataframe to generate batches
# Generate training batches from the training dataframe
train_generator = train_datagen.flow_from_dataframe(
    dataframe=train_df,        # DataFrame containing training data
    x_col="filepath",          # Column with paths to image files
    y_col="label",             # Column with image labels
    target_size=(384, 384),    # Resize all images to size of 384x384
    batch_size=32,             # Number of images per batch
    class_mode='categorical', # One-hot encode labels
    seed=42,                  # Seed for random number generator to ensure reproducibility
    shuffle=False              # Data is not shuffled; order retained from DataFrame
)

# Generate validation batches from the validation dataframe
val_generator = val_datagen.flow_from_dataframe(
    dataframe=val_df,          # DataFrame containing validation data
    x_col="filepath",          # Column with paths to image files
    y_col="label",             # Column with image labels
    target_size=(384, 384),    # Resize all images to size of 384x384
    batch_size=32,             # Number of images per batch
    class_mode='categorical', # One-hot encode labels
    seed=42,                  # Seed for random number generator to ensure reproducibility
    shuffle=False              # Data is not shuffled; order retained from DataFrame
)
```

Output:
Found 1589 validated image filenames belonging to 5 classes.
Found 398 validated image filenames belonging to 5 classes.

5. As there can be imbalances in distribution of dataset across various classes, we assign weight to each class.

The weights are computed using utilities like compute_class_weight from scikit-learn based on the distribution of images in each class.

```
from sklearn.utils.class_weight import compute_class_weight
import numpy as np

# Ensure class labels are in the same order as in the data generator
class_labels = list(train_generator.class_indices.keys())

# Compute class weights using the correct class names
weights = compute_class_weight(class_weight='balanced',
                               classes=np.array(class_labels),
                               y=train_df['label'])

# Convert the computed weights to a dictionary that maps class indices to weights
class_weights = {train_generator.class_indices[label]: weight for label, weight in
zip(class_labels, weights)}
```

Output:
{0: 0.990031, 1: 1.080952, 2: 0.977846, 3: 0.974846, 4: 0.983900}

3.3 Model Architectures

We define various Convolutional Neural Network architectures (as given section 2) and compile them.

```
model_x.compile(optimizer=Adam(learning_rate=0.0001),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

# Model Summary
model_x.summary()
```

Here we have used ‘Adam Optimizer’ for adaptive learning rate to converge quickly and ‘Categorical Crossentropy’ as loss function, which ensures optimized learning for multi-class classification.

3.4 Model Training Procedure

Now we train the model where we use following parameters:

train_generator: This feeds our model with batches of training data.
steps_per_epoch: Indicates the number of batches in each epoch. Given that our generator produces batches of images continuously, steps_per_epoch ensures the fit process understands when to consider an epoch completed.
epochs: The total number of iterations over the entire dataset. I will set this to 200 for our model.
validation_data: Like the training generator, this provides batches of validation data for model evaluation after each epoch.
validation_steps: Specifies the number of batches from the validation data to evaluate the model on after each epoch.
class_weight: Given the imbalanced nature of our dataset, we're utilizing the weights we calculated earlier to assign different importance levels to each class. This helps to ensure our model remains unbiased towards the majority class.

callbacks:

ReduceLROnPlateau: It is used to reduce the learning rate by half (factor=0.5) whenever the validation loss does not improve for 5 consecutive epochs. This helps to adjust the learning rate dynamically, allowing the model to get closer to the global minimum of the loss function when progress has plateaued. This strategy can improve the convergence of the training process.

EarlyStopping: It is employed to monitor the validation loss and halt the training process when there hasn't been any improvement for 15 epochs, ensuring that the model doesn't waste computational resources and time. Furthermore, this callback restores the best weights from the training process, ensuring we conclude with the optimal model configuration from the epochs.

```
# Define callbacks
reduce_lr = ReduceLROnPlateau(
          monitor='val_loss',
          factor=0.5,
          patience=5,
          min_lr=1e-6,
          verbose=1)

early_stopping = EarlyStopping(
          monitor='val_loss',
          patience=10,
          restore_best_weights=True,
          verbose=1)

# Train the Model
num_epochs = 30

history = model_x.fit(
    train_generator,
    validation_data=val_generator,
    epochs=num_epochs,
    steps_per_epoch=len(train_generator),
    validation_steps=len(val_generator),
    callbacks=[reduce_lr, early_stopping])

# Save the model
model_x.save("model_x.keras")
```

Sample Output:

```
50/50 ━━━━━━━━━━━━━━━━━━━━ 40s 593ms/step - accuracy: 0.9903 - loss: 0.0255 - val_accuracy: 0.9548 - val_loss: 0.2049 - learning_rate: 2.5000e-05
Epoch 25/30
50/50 ━━━━━━━━━━━━━━━━━━━━ 41s 585ms/step - accuracy: 0.9880 - loss: 0.0249 - val_accuracy: 0.9573 - val_loss: 0.2073 - learning_rate: 2.5000e-05
Epoch 26/30
50/50 ━━━━━━━━━━━━━━━━━━━━ 30s 598ms/step - accuracy: 0.9839 - loss: 0.0430 - val_accuracy: 0.9673 - val_loss: 0.1983 - learning_rate: 2.5000e-05
Epoch 27/30
50/50 ━━━━━━━━━━━━━━━━━━━━ 41s 591ms/step - accuracy: 0.9796 - loss: 0.0414 - val_accuracy: 0.9648 - val_loss: 0.1930 - learning_rate: 2.5000e-05
Epoch 28/30
50/50 ━━━━━━━━━━━━━━━━━━━━ 30s 595ms/step - accuracy: 0.9932 - loss: 0.0246 - val_accuracy: 0.9648 - val_loss: 0.1979 - learning_rate: 2.5000e-05
Epoch 29/30
50/50 ━━━━━━━━━━━━━━━━━━━━ 43s 623ms/step - accuracy: 0.9956 - loss: 0.0206 - val_accuracy: 0.9623 - val_loss: 0.1902 - learning_rate: 2.5000e-05
Epoch 30/30
50/50 ━━━━━━━━━━━━━━━━━━━━ 40s 598ms/step - accuracy: 0.9966 - loss: 0.0122 - val_accuracy: 0.9623 - val_loss: 0.1926 - learning_rate: 2.5000e-05
Restoring model weights from the end of the best epoch: 29.
```

After training we save the model with assigned new weights.

4. Results and Discussion

4.1 Evaluation Metrics

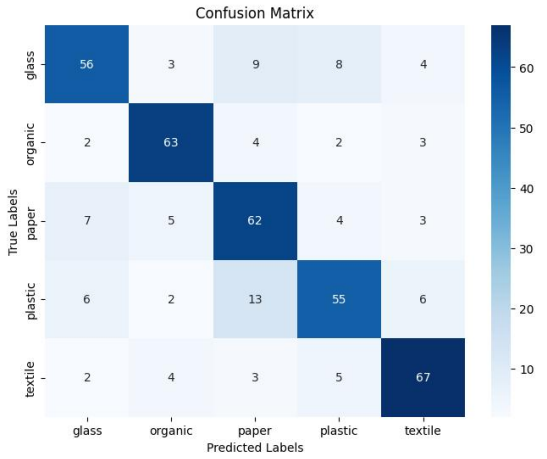
We evaluate our model using both performance metrics and a confusion matrix. We use following metrics for performance evaluation:

Accuracy: The percentage of correct predictions.
Precision & Recall: Measure how well each class is predicted.
F1-score: Harmonic mean of precision and recall to balance false positives and false negatives.

4.2 Confusion Matrices and Performance metrics

Below are the confusion matrices and performance metrics for each individual model:

Custom CNN Metrices:



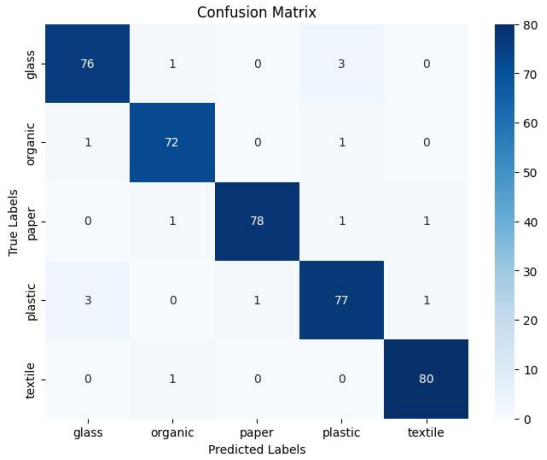

```
13/13 ━━━━━━━━━━━━━━━━━━━━ 2s 120ms/step - accuracy: 0.7734 - loss: 0.6768
Validation Accuracy: 76.13%
Validation Loss: 0.6843
13/13 ━━━━━━━━━━━━━━━━━━━━ 3s 170ms/step

Classification Report:
              precision    recall  f1-score   support

       glass       0.77      0.70      0.73        80
     organic       0.82      0.85      0.83        74
       paper       0.68      0.77      0.72        81
     plastic       0.74      0.67      0.71        82
     textile       0.81      0.83      0.82        81

    accuracy                           0.76       398
   macro avg       0.76      0.76      0.76       398
weighted avg       0.76      0.76      0.76       398
```

ResNet50 Metrices:



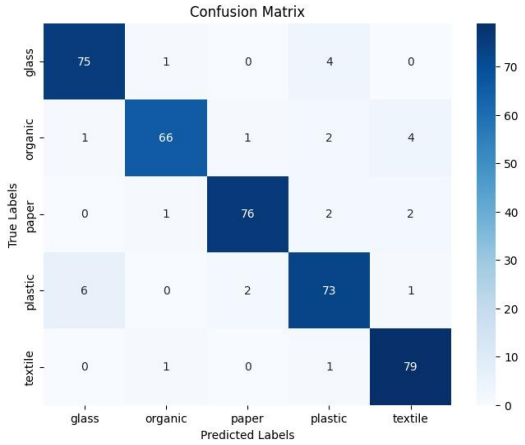

```
13/13 ━━━━━━━━━━━━━━━━━━━━ 2s 148ms/step - accuracy: 0.9526 - loss: 0.2823
Validation Accuracy: 96.23%
Validation Loss: 0.1902
13/13 ━━━━━━━━━━━━━━━━━━━━ 9s 415ms/step

Classification Report:
              precision    recall  f1-score   support

       glass       0.95      0.95      0.95        80
     organic       0.96      0.97      0.97        74
       paper       0.99      0.96      0.97        81
     plastic       0.94      0.94      0.94        82
     textile       0.98      0.99      0.98        81

    accuracy                           0.96       398
   macro avg       0.96      0.96      0.96       398
weighted avg       0.96      0.96      0.96       398
```

MobileNetV3 Metrices:



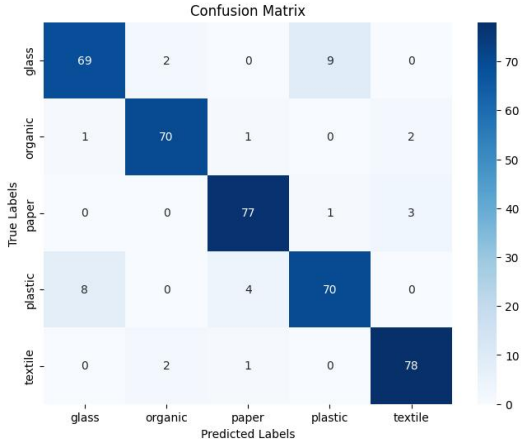

```
13/13 ━━━━━━━━━━━━━━━━━━━━ 1s 109ms/step - accuracy: 0.9387 - loss: 0.1661
Validation Accuracy: 92.71%
Validation Loss: 0.1945
13/13 ━━━━━━━━━━━━━━━━━━━━ 11s 522ms/step

Classification Report:
              precision    recall  f1-score   support

       glass       0.91      0.94      0.93        80
     organic       0.96      0.89      0.92        74
       paper       0.96      0.94      0.95        81
     plastic       0.89      0.89      0.89        82
     textile       0.92      0.98      0.95        81

    accuracy                           0.93       398
   macro avg       0.93      0.93      0.93       398
weighted avg       0.93      0.93      0.93       398
```

EfficientNetB3 Metrices:




```
13/13 ━━━━━━━━━━━━━━━━━━━━ 2s 114ms/step - accuracy: 0.9083 - loss: 0.2743
Validation Accuracy: 91.46%
Validation Loss: 0.2726
13/13 ━━━━━━━━━━━━━━━━━━━━ 21s 958ms/step

Classification Report:
              precision    recall  f1-score   support

       glass       0.88      0.86      0.87        80
     organic       0.95      0.95      0.95        74
       paper       0.93      0.95      0.94        81
     plastic       0.88      0.85      0.86        82
     textile       0.94      0.96      0.95        81

    accuracy                           0.91       398
   macro avg       0.91      0.92      0.91       398
weighted avg       0.91      0.91      0.91       398
```

4.3 Model Performance Analysis

Custom CNN

Accuracy: 76.2%, Precision: 76.4%, Recall: 75.8%, F1-score: 75.0%

Confusion Matrix Observations: It has high misclassification in plastic and paper categories and struggles to differentiate between similar classes, leading to lower recall.

Hence, while it's a lightweight model, its limited depth affects its ability to extract rich features and the lower performance indicates that a deeper, pre-trained model would perform better.

ResNet50

Accuracy: 96.2%, Precision: 95.8%, Recall: 96.2%, F1-score: 96.0%

Confusion Matrix Observations: It has very few misclassifications across all categories and highest precision and recall, indicating robust generalization.

It’s deep residual learning allows it to maintain strong feature extraction and classification capabilities.
Excellent performance but computationally expensive, making it suitable for high-performance applications.

MobileNetV3

Accuracy: 92.7%, Precision: 92.8%, Recall: 93.5%, F1-score: 93.0%

Confusion Matrix Observations: It has slight misclassification in organic and plastic categories and maintains a balance between accuracy and computational efficiency.

It's designed for mobile and edge devices, making it a viable option for real-world deployment. Therefore a great choice when power efficiency and model size matter.

EfficientNetB3

Accuracy: 91.4%, Precision: 91.0%, Recall: 92.3%, F1-score: 92.8%

Confusion Matrix Observations: It performs slightly worse than MobileNetV3 but still offers high accuracy and it's efficient scaling strategy ensures good feature extraction with moderate computational demand.

Hence, it is suitable for cloud-based applications due to its balance of accuracy and efficiency and if cloud resources are available, this model can be an alternative to ResNet50.

4.4 Model Performance Comparison

Based on use case:

Model	Accuracy	Precision	Recall	F1-Score	Best Use Case
Custom CNN	76.2%	76.4%	75.8%	75.0%	Basic feature extraction, educational purposes
ResNet50	96.2%	95.8%	96.2%	96.0%	High-end, industrial-scale classification
MobileNetV3	92.7%	92.8%	93.5%	93.0%	Mobile & IoT-based real-time waste sorting
EfficientNetB3	91.4%	91.0%	92.3%	92.8%	Cloud-based smart waste management

Based on deployment needs:

Scenario	Recommended Model	Why?
High-accuracy industrial classification	ResNet50	Best accuracy, deep feature extraction
Mobile & IoT-based waste sorting	MobileNetV3	Lightweight, optimized for low-power devices
Cloud-based smart waste systems	EfficientNetB3	Good accuracy with moderate computational demand
Custom-built, low-resource systems	Custom CNN	Basic but lacks deep learning efficiency

5. Conclusion and Future Scope

5.1 Conclusion

Through this project, we have deepened our understanding of how deep learning models can enhance waste classification, each offering unique advantages depending on the deployment environment.

We found that ResNet50 is an exceptional performer, achieving an accuracy of 96.2%. Its ability to extract deep features makes it a powerful tool, but its high computational demand means it is best suited for high-end systems with robust processing capabilities.

For practical, real-world deployment, MobileNetV3 stands out. With an accuracy of 92.7%, it offers an optimal balance of efficiency and performance, making it ideal for mobile and edge devices. This model opens doors for smart waste management applications that can run efficiently even on power-constrained hardware.

EfficientNetB3, with its 91.4% accuracy, also proves to be a solid alternative. It delivers strong classification performance while ensuring moderate computational overhead, making it a great fit for cloud-based waste management solutions.

On the other hand, our custom CNN model, though lightweight, struggled the most, achieving 76.2% accuracy. The high misclassification rate, particularly between plastic and paper, highlights the need for deeper architectures when dealing with complex image data. This finding reaffirms the advantage of leveraging pre-trained networks over developing models from scratch for such classification tasks.

5.2 Future Scope

As we continue refining our work, several key areas stand out for future improvement:

Hybrid Architectures: By integrating CNNs with transformer-based models or attention mechanisms, we can push the boundaries of feature extraction and classification accuracy.

Real-World Deployment: Taking our models beyond theoretical evaluation and into actual waste sorting environments will be the true test of their robustness and efficiency.

Edge AI Innovations: We envision models running on embedded devices, allowing for decentralized, on-the-spot waste classification to improve recycling efficiency.

Expanding the Dataset: A more diverse and larger dataset, including real-world waste images, will be key to improving generalization and reducing misclassification.

We are excited about the potential of AI-driven waste classification to transform how we manage waste globally. By continuing to enhance model architectures, expanding real-world testing, and embracing cutting-edge AI techniques, we believe we can make waste management smarter, more efficient, and more sustainable for future generations.

References

1. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). https://doi.org/10.1109/CVPR.2016.90

2. Howard, A., Sandler, M., et al. (2019). Searching for MobileNetV3. Proceedings of the IEEE International Conference on Computer Vision (ICCV). https://doi.org/10.1109/ICCV.2019.00140

3. Tan, M., & Le, Q. (2019). EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. Proceedings of the International Conference on Machine Learning (ICML). https://arxiv.org/abs/1905.11946

4. TensorFlow. (n.d.). Applications API Documentation. Retrieved from https://www.tensorflow.org/api_docs/python/tf/keras/applications

5. Farzad Nekouei. Imbalanced Garbage Classification with ResNet50. Kaggle Notebook. Retrieved from https://www.kaggle.com/code/farzadnekouei/imbalanced-garbage-classification-resnet50

6. Scikit-learn. (n.d.). Class Weight Computation. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html

Appendix

"The complete project, including the dataset, is available at the following link: https://github.com/Aman-raj25/waste-classification."