

Environment Setup

```
In [23]: # Environment setup: install/upgrade libraries as needed
!pip install -q torch torchvision torchaudio --index-url https://download
!pip install -q seaborn scikit-learn tqdm torchsummary
!pip install -q kaggle
```

```
In [24]: # Step 0: Download TrashNet dataset into runtime at /content/trashnet

import os
import zipfile
import urllib.request
from tqdm import tqdm

DATA_ROOT = "/content/trashnet"
ZIP_PATH = "/content/trashnet.zip"
KAGGLE_URL = "https://github.com/garythung/trashnet/archive/refs/heads/main.zip"

# Only download if folder doesn't exist
if not os.path.exists(DATA_ROOT):
    print("Downloading TrashNet dataset...")

    # Download the ZIP file with progress bar
    class DownloadProgressBar(tqdm):
        def update_to(self, b=1, bsize=1, tsize=None):
            if tsize is not None:
                self.total = tsize
            self.update(b * bsize)

    with DownloadProgressBar(unit='B', unit_scale=True, miniters=1, desc="Downloading") as tqdm:
        urllib.request.urlretrieve(KAGGLE_URL, filename=ZIP_PATH, reporthook=tqdm.update_to)

    # Extract
    print("Extracting dataset...")
    with zipfile.ZipFile(ZIP_PATH, 'r') as zip_ref:
        zip_ref.extractall("/content/")

    # The extracted folder path
    extracted_path = "/content/trashnet-master/data"

    # Move to desired DATA_ROOT
    os.rename(extracted_path, DATA_ROOT)
    print("Dataset ready at:", DATA_ROOT)

else:
    print("Dataset already exists at:", DATA_ROOT)
```

Dataset already exists at: /content/trashnet

Part 1 – Project Overview and Problem Statement

1.1 Background and Motivation

Cities around the world continue to struggle with increasing volumes of municipal solid waste. Efficient recycling depends heavily on proper waste segregation; however, manual sorting of waste is slow, labor-intensive, and often exposes workers to health risks. Misclassification of recyclable materials as general trash leads to unnecessary landfill use and the loss of valuable resources.

Computer vision and deep learning offer a way to automate visual recognition tasks, including waste sorting. In this project, I use image-based classification on the public TrashNet dataset to automatically assign single waste items to one of six categories: cardboard, glass, metal, paper, plastic, or general trash.

1.2 Problem Statement

The goal of this project is to design, implement, and evaluate an image-based waste classification system that takes a single RGB image of a waste item and predicts its category among:

- Cardboard
- Glass
- Metal
- Paper
- Plastic
- Trash (non-recyclable or mixed waste)

The system will be trained and evaluated on the TrashNet dataset. The key challenges include:

- High intra-class variability: items within a class can vary in shape, size, and color (e.g., different plastic bottles or crumpled paper).
- Domain confusion: dirty, wrinkled, or partially occluded items may resemble other classes.
- Class imbalance: the 'trash' class has significantly fewer images than other classes.

1.3 Objectives

The specific objectives of this project are:

1. To explore and describe the TrashNet dataset and its class distribution.
2. To build a baseline deep learning classifier using transfer learning.
3. To improve robustness through data augmentation and basic regularization.
4. To implement a hybrid approach that uses deep features with a classical classifier, and compare it to the end-to-end deep learning model.
5. To evaluate the models using appropriate metrics (accuracy, confusion matrix, per-class precision/recall) and discuss their potential for real-world deployment.

```
In [25]: # Part 1 – Core imports

import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from PIL import Image

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

from sklearn.metrics import classification_report, confusion_matrix
from sklearn.linear_model import LogisticRegression

from tqdm import tqdm
```

Part 2 – Dataset Description and Basic Exploration

2.1 Dataset Source

For this project I use the TrashNet dataset, which is publicly available on Kaggle. The dataset contains **2,527 images** of waste items, each belonging to one of six classes: cardboard, glass, metal, paper, plastic, and trash. The original images are approximately **512 × 384 pixels** and were captured with a mobile phone camera under relatively controlled conditions. The total dataset size is around **47 MB**.

Each class is stored as a separate folder:

- cardboard/
- glass/
- metal/
- paper/
- plastic/
- trash/

This folder structure is convenient for using standard image-loading utilities in deep learning frameworks such as PyTorch.

```
In [26]: # Extracting Zip files
import zipfile

zip_path = "/content/trashnet/dataset-resized.zip"
extract_dir = "/content/trashnet/dataset-resized"

if not os.path.isdir(extract_dir):
    print("Extracting dataset-resized.zip...")
```

```

with zipfile.ZipFile(zip_path, 'r') as z:
    z.extractall("/content/trashnet")
    print("Extraction complete.")
else:
    print("Folder already exists:", extract_dir)

print("Contents of /content/trashnet after extraction:")
print(os.listdir("/content/trashnet"))

```

Folder already exists: /content/trashnet/dataset-resized

Contents of /content/trashnet after extraction:

```

['one-indexed-files-notrash_train.txt', 'one-indexed-files.txt', 'constant
s.py', 'resize.py', 'one-indexed-files-notrash_val.txt', '__MACOSX', 'data
set-resized', 'dataset-resized.zip', 'zero-indexed-files.txt', 'one-indexe
d-files-notrash_test.txt']

```

```

In [27]: # Define dataset root and list class folders
DATA_ROOT = "/content/trashnet/dataset-resized"

classes = sorted([d for d in os.listdir(DATA_ROOT) if os.path.isdir(os.pa
print("Classes:", classes)

```

Classes: ['cardboard', 'glass', 'metal', 'paper', 'plastic', 'trash']

```

In [28]: # Count number of images per class

class_counts = {}

for cls in classes:
    folder = os.path.join(DATA_ROOT, cls)
    count = len([f for f in os.listdir(folder) if f.lower().endswith((".j
class_counts[cls] = count

class_counts

```

```

Out[28]: {'cardboard': 403,
          'glass': 501,
          'metal': 410,
          'paper': 594,
          'plastic': 482,
          'trash': 137}

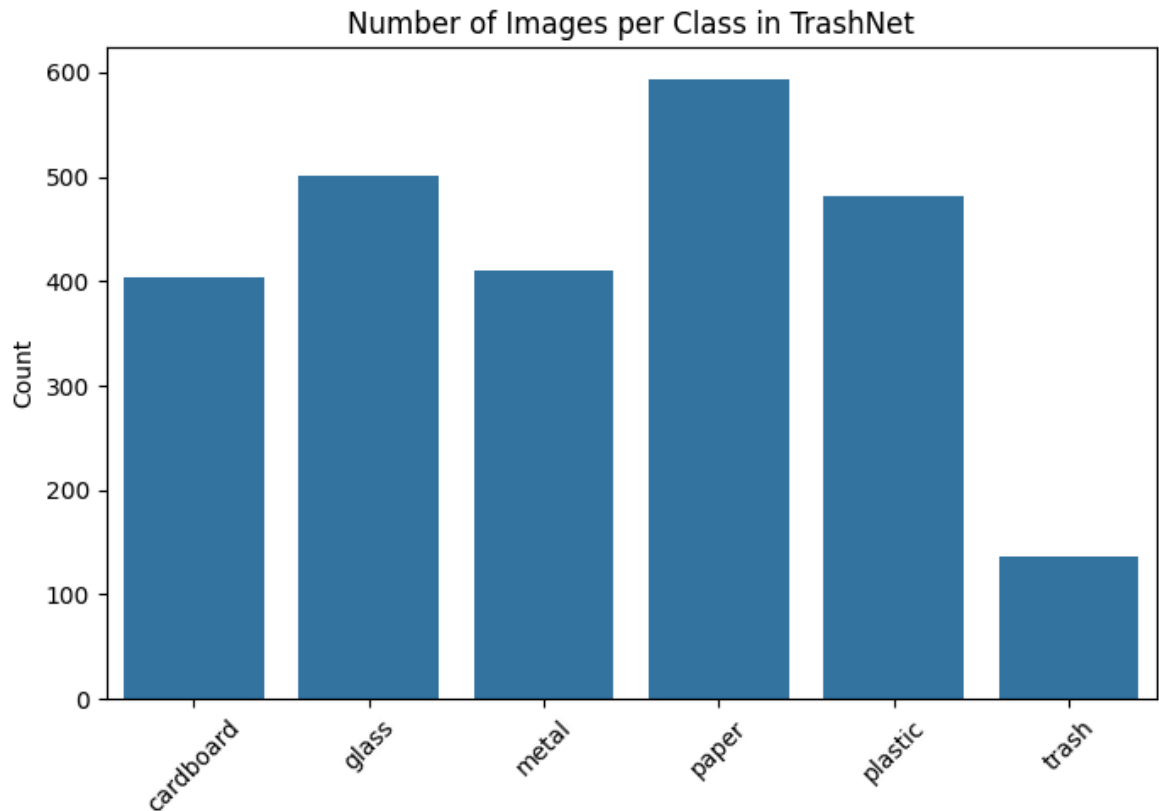
```

```

In [29]: # Plot class distribution

plt.figure(figsize=(8, 5))
sns.barplot(x=list(class_counts.keys()), y=list(class_counts.values()))
plt.title("Number of Images per Class in TrashNet")
plt.ylabel("Count")
plt.xticks(rotation=45)
plt.show()

```



```
In [30]: # Visualize a few sample images from each class

num_samples_per_class = 3
plt.figure(figsize=(12, 8))

idx = 1
for cls in classes:
    folder = os.path.join(DATA_ROOT, cls)
    image_files = [f for f in os.listdir(folder) if f.lower().endswith(("
    for img_name in image_files[:num_samples_per_class]:
        img_path = os.path.join(folder, img_name)
        img = Image.open(img_path).convert("RGB")
        plt.subplot(len(classes), num_samples_per_class, idx)
        plt.imshow(img)
        plt.axis("off")
        if idx % num_samples_per_class == 1:
            plt.ylabel(cls, rotation=0, labelpad=40)
        idx += 1

plt.suptitle("Sample Images from TrashNet Classes", fontsize=16)
plt.tight_layout()
plt.show()
```

Sample Images from TrashNet Classes



Part 3 – Data Preprocessing and Augmentation

3.1 Image Size and Normalization

To reduce computational cost while preserving enough visual detail, I resize all images to a fixed size of **224 × 224 pixels**, which is compatible with standard pretrained models such as MobileNet and ResNet.

The images are also normalized using the standard ImageNet mean and standard deviation:

- mean = (0.485, 0.456, 0.406)
- std = (0.229, 0.224, 0.225)

This helps the pretrained network weights remain effective when fine-tuned on TrashNet.

3.2 Data Augmentation

Because the dataset is relatively small and the classes have intra-class variability, I apply data augmentation to the training set. Specifically, I use:

- Random horizontal flip
- Random rotation (e.g., ± 15 degrees)
- Random affine transformations (small translations and scaling)
- Random color jitter (small changes in brightness/contrast)

The validation and test sets only undergo resizing and normalization. This separation ensures that performance is evaluated on clean, non-augmented data.

3.3 Train / Validation / Test Split

The original dataset does not come with an official train/validation/test split. I create my own split:

- 70% training
- 15% validation
- 15% test

This split is stratified by class so that each subset has a similar class distribution. The same split is used throughout the experiments to ensure fair comparison between models.

```
In [31]: # Define image transforms

IMAGE_SIZE = 224

train_transform = transforms.Compose([
    transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(15),
    transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.485, 0.456, 0.406),
                        std=(0.229, 0.224, 0.225)),
])

eval_transform = transforms.Compose([
    transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.485, 0.456, 0.406),
                        std=(0.229, 0.224, 0.225)),
])

In [32]: # Load dataset using ImageFolder

full_dataset = datasets.ImageFolder(root=DATA_ROOT, transform=eval_transf
print("Total images:", len(full_dataset))
print("Classes (ImageFolder):", full_dataset.classes)
```

Total images: 2527

Classes (ImageFolder): ['cardboard', 'glass', 'metal', 'paper', 'plastic', 'trash']

```
In [33]: # Create train/val/test splits (stratified)

from sklearn.model_selection import train_test_split

targets = np.array(full_dataset.targets)

train_idx, temp_idx = train_test_split(
    np.arange(len(targets)),
    test_size=0.30,
    stratify=targets,
    random_state=42
)

val_idx, test_idx = train_test_split(
    temp_idx,
    test_size=0.50,
    stratify=targets[temp_idx],
    random_state=42
)

len(train_idx), len(val_idx), len(test_idx)
```

Out[33]: (1768, 379, 380)

```
In [34]: # Subset datasets with appropriate transforms

from torch.utils.data import Subset

# For train set, we want augmentation (train_transform)
train_dataset = Subset(datasets.ImageFolder(root=DATA_ROOT, transform=train_transform),
                        train_idx)

# For val and test, we use evaluation transforms
base_eval_dataset = datasets.ImageFolder(root=DATA_ROOT, transform=eval_transform)
val_dataset = Subset(base_eval_dataset, val_idx)
test_dataset = Subset(base_eval_dataset, test_idx)

print("Train / Val / Test sizes:", len(train_dataset), len(val_dataset),
```

Train / Val / Test sizes: 1768 379 380

```
In [35]: # Create DataLoaders

BATCH_SIZE = 32
NUM_WORKERS = 2 # Colab usually supports 2 or more

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

Part 4 – Baseline Deep Learning Model

4.1 Model Choice

As a baseline, I use a lightweight pretrained convolutional neural network (MobileNet or a similar architecture) and fine-tune it on the TrashNet dataset. Using transfer

learning is appropriate here because:

- The dataset is relatively small (2,527 images total).
- The model can reuse low-level visual features learned from large-scale datasets such as ImageNet (edges, textures, shapes).
- It is more efficient than training a deep network from scratch.

I replace the final classification layer of the pretrained model with a new fully connected layer that outputs six logits corresponding to the TrashNet classes. The rest of the network is fine-tuned with a small learning rate.

4.2 Training Objective

The training objective is multi-class classification using the cross-entropy loss. I use the Adam optimizer with a modest learning rate (e.g., $1e-4$) and train for a small number of epochs (e.g., 10–20) as a starting baseline.

```
In [36]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device
```

```
Out[36]: device(type='cuda')
```

```
In [37]: # Define baseline model (MobileNetV3-Small)

num_classes = len(classes)

baseline_model = models.mobilenet_v3_small(weights=models.MobileNet_V3_Sm
# Replace classification head
baseline_model.classifier[3] = nn.Linear(in_features=baseline_model.class
                                     out_features=num_classes)

baseline_model = baseline_model.to(device)

print(baseline_model.classifier)
```

```
Sequential(
  (0): Linear(in_features=576, out_features=1024, bias=True)
  (1): Hardswish()
  (2): Dropout(p=0.2, inplace=True)
  (3): Linear(in_features=1024, out_features=6, bias=True)
)
```

```
In [38]: # Define loss function and optimizer

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(baseline_model.parameters(), lr=1e-4)
```

```
In [39]: # Training and validation loop helpers

def train_one_epoch(model, loader, optimizer, criterion, device):
    model.train()
    running_loss, correct, total = 0.0, 0, 0

    for images, labels in tqdm(loader, desc="Train", leave=False):
        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)
_, preds = torch.max(outputs, 1)
correct += (preds == labels).sum().item()
total += labels.size(0)

epoch_loss = running_loss / total
epoch_acc = correct / total
return epoch_loss, epoch_acc

def evaluate(model, loader, criterion, device):
    model.eval()
    running_loss, correct, total = 0.0, 0, 0

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

            running_loss += loss.item() * images.size(0)
            _, preds = torch.max(outputs, 1)
            correct += (preds == labels).sum().item()
            total += labels.size(0)

    epoch_loss = running_loss / total
    epoch_acc = correct / total
    return epoch_loss, epoch_acc

```

```

In [40]: # Run a few epochs for baseline

EPOCHS_BASELINE = 15

train_history = []
val_history = []

for epoch in range(EPOCHS_BASELINE):
    print(f"Epoch {epoch+1}/{EPOCHS_BASELINE}")
    train_loss, train_acc = train_one_epoch(baseline_model, train_loader,
    val_loss, val_acc = evaluate(baseline_model, val_loader, criterion, d

    train_history.append((train_loss, train_acc))
    val_history.append((val_loss, val_acc))

    print(f"  Train loss: {train_loss:.4f}, acc: {train_acc:.4f}")
    print(f"  Val   loss: {val_loss:.4f}, acc: {val_acc:.4f}")

```

Epoch 1/15

Train loss: 1.3737, acc: 0.5074

Val loss: 1.0457, acc: 0.6491

Epoch 2/15

```
Train loss: 0.7723, acc: 0.7449
Val loss: 0.7044, acc: 0.7599
Epoch 3/15
Train loss: 0.5037, acc: 0.8343
Val loss: 0.4873, acc: 0.8628
Epoch 4/15
Train loss: 0.3819, acc: 0.8710
Val loss: 0.4125, acc: 0.8707
Epoch 5/15
Train loss: 0.2821, acc: 0.9044
Val loss: 0.3666, acc: 0.8760
Epoch 6/15
Train loss: 0.2230, acc: 0.9333
Val loss: 0.3774, acc: 0.8707
Epoch 7/15
Train loss: 0.1890, acc: 0.9434
Val loss: 0.3265, acc: 0.8945
Epoch 8/15
Train loss: 0.1398, acc: 0.9604
Val loss: 0.3721, acc: 0.8760
Epoch 9/15
Train loss: 0.1260, acc: 0.9604
Val loss: 0.3304, acc: 0.8865
Epoch 10/15
Train loss: 0.1085, acc: 0.9678
Val loss: 0.3546, acc: 0.8971
Epoch 11/15
Train loss: 0.1013, acc: 0.9712
Val loss: 0.3218, acc: 0.8945
Epoch 12/15
Train loss: 0.0642, acc: 0.9830
Val loss: 0.3160, acc: 0.9077
Epoch 13/15
Train loss: 0.0637, acc: 0.9802
Val loss: 0.3635, acc: 0.9024
Epoch 14/15
Train loss: 0.0515, acc: 0.9847
Val loss: 0.3658, acc: 0.8971
Epoch 15/15
Train loss: 0.0547, acc: 0.9859
Val loss: 0.3940, acc: 0.8892
```

Part 5 – Improved Model and Regularization

5.1 Motivation for Improvement

The baseline MobileNet model provides a reasonable starting point, but there are several potential weaknesses:

- The dataset is relatively small and imbalanced, especially for the `trash` class.

- The baseline augmentation may not fully capture real-world variability (e.g., different orientations, lighting conditions, or partial occlusions).
- The optimizer uses a fixed learning rate, which may not be optimal throughout training.

To address these issues, I implement three improvements:

1. **Stronger data augmentation:** I introduce slightly more aggressive spatial and color transformations on the training images to encourage robustness to nuisance variations.
2. **Class weighting in the loss function:** I compute class weights based on the training set distribution and apply them in the cross-entropy loss. This ensures that under-represented classes (especially `trash`) have a stronger influence during training.
3. **Learning rate scheduling:** I use a simple learning rate scheduler so that the learning rate decays over time, helping the model converge more stably.

The goal is not only to increase overall accuracy, but also to improve per-class performance, especially on under-represented and visually ambiguous classes.

```
In [41]: # Compute class counts and class weights from train split

from collections import Counter

# Recall: full_dataset was built with datasets.ImageFolder(root=DATA_ROOT
# and train_idx, val_idx, test_idx were created earlier.

train_targets = [full_dataset.targets[i] for i in train_idx]
class_count = Counter(train_targets)

print("Train class counts (index -> count):", class_count)

num_classes = len(classes)
total_samples = len(train_targets)

class_weights = []
for class_idx in range(num_classes):
    # Inverse frequency: more weight for rare classes
    class_weights.append(total_samples / (num_classes * class_count[class_idx]))

class_weights
```

```
Train class counts (index -> count): Counter({3: 416, 1: 350, 4: 337, 2: 287, 0: 282, 5: 96})
```

```
Out[41]: [1.044917257683215,
0.8419047619047619,
1.0267131242740999,
0.7083333333333334,
0.874381800197824,
3.0694444444444446]
```

```
In [42]: # Define improved model (same architecture, fresh weights)
```

```

improved_model = models.mobilenet_v3_small(
    weights=models.MobileNet_V3_Small_Weights.IMAGENET1K_V1
)

improved_model.classifier[3] = nn.Linear(
    in_features=improved_model.classifier[3].in_features,
    out_features=num_classes
)

improved_model = improved_model.to(device)

```

```

In [43]: # Weighted loss and optimizer with scheduler

class_weights_tensor = torch.tensor(class_weights, dtype=torch.float32).t

criterion_improved = nn.CrossEntropyLoss(weight=class_weights_tensor)
optimizer_improved = torch.optim.Adam(improved_model.parameters(), lr=1e-

# Simple StepLR scheduler: decay LR by 0.1 every 5 epochs
scheduler_improved = torch.optim.lr_scheduler.StepLR(
    optimizer_improved, step_size=5, gamma=0.1
)

```

```

In [44]: # Stronger augmentation for improved model

train_transform_improved = transforms.Compose([
    transforms.RandomResizedCrop(IMAGE_SIZE, scale=(0.8, 1.0)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(20),
    transforms.RandomApply([
        transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0
    ], p=0.5),
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.485, 0.456, 0.406),
                          std=(0.229, 0.224, 0.225)),
])

# Rebuild only the training dataset with the new augmentation
train_dataset_improved = Subset(
    datasets.ImageFolder(root=DATA_ROOT, transform=train_transform_improv
    train_idx

train_loader_improved = DataLoader(
    train_dataset_improved, batch_size=BATCH_SIZE, shuffle=True, num_work
)

```

```

In [ ]: # Train improved model

EPOCHS_IMPROVED = 10

train_hist_improved = []
val_hist_improved = []

for epoch in range(EPOCHS_IMPROVED):
    print(f"Improved Model - Epoch {epoch+1}/{EPOCHS_IMPROVED}")

    train_loss, train_acc = train_one_epoch(
        improved_model, train_loader_improved, optimizer_improved, criter

```

```

    )
    val_loss, val_acc = evaluate(
        improved_model, val_loader, criterion_improved, device
    )

    train_hist_improved.append((train_loss, train_acc))
    val_hist_improved.append((val_loss, val_acc))

    print(f" Train loss: {train_loss:.4f}, acc: {train_acc:.4f}")
    print(f" Val   loss: {val_loss:.4f}, acc: {val_acc:.4f}")

    scheduler_improved.step()

```

Improved Model – Epoch 1/10

Train loss: 1.5130, acc: 0.4621
Val loss: 1.1533, acc: 0.6570

Improved Model – Epoch 2/10

Train loss: 0.9285, acc: 0.7025
Val loss: 0.7417, acc: 0.7704

Improved Model – Epoch 3/10

Train loss: 0.6192, acc: 0.7998
Val loss: 0.5883, acc: 0.8206

Improved Model – Epoch 4/10

Train loss: 0.4574, acc: 0.8422
Val loss: 0.5079, acc: 0.8311

Improved Model – Epoch 5/10

Train loss: 0.3413, acc: 0.8914
Val loss: 0.6242, acc: 0.7810

Improved Model – Epoch 6/10

Train loss: 0.3083, acc: 0.8959
Val loss: 0.4816, acc: 0.8575

Improved Model – Epoch 7/10

Train loss: 0.2860, acc: 0.8999
Val loss: 0.4644, acc: 0.8549

Improved Model – Epoch 8/10

Train loss: 0.2627, acc: 0.9157
Val loss: 0.4619, acc: 0.8681

Improved Model – Epoch 9/10

Train loss: 0.2721, acc: 0.9118
Val loss: 0.4606, acc: 0.8760

Improved Model – Epoch 10/10

Train loss: 0.2610, acc: 0.9152
Val loss: 0.4556, acc: 0.8760

In [46]: *# Training for 10 more epochs*

EPOCHS_IMPROVED = 10

train_hist_improved = []

val_hist_improved = []

for epoch in range(EPOCHS_IMPROVED):

print(f"Improved Model – Epoch {epoch+1}/{EPOCHS_IMPROVED}")

train_loss, train_acc = train_one_epoch(

improved_model, train_loader_improved, optimizer_improved, criterion_improved,

)

val_loss, val_acc = evaluate(

```

        improved_model, val_loader, criterion_improved, device
    )

    train_hist_improved.append((train_loss, train_acc))
    val_hist_improved.append((val_loss, val_acc))

    print(f" Train loss: {train_loss:.4f}, acc: {train_acc:.4f}")
    print(f" Val   loss: {val_loss:.4f}, acc: {val_acc:.4f}")

    scheduler_improved.step()

```

Improved Model – Epoch 1/10

Train loss: 0.2525, acc: 0.9135
Val loss: 0.4583, acc: 0.8734

Improved Model – Epoch 2/10

Train loss: 0.2491, acc: 0.9186
Val loss: 0.4610, acc: 0.8734

Improved Model – Epoch 3/10

Train loss: 0.2383, acc: 0.9242
Val loss: 0.4617, acc: 0.8734

Improved Model – Epoch 4/10

Train loss: 0.2519, acc: 0.9202
Val loss: 0.4601, acc: 0.8734

Improved Model – Epoch 5/10

Train loss: 0.2359, acc: 0.9208
Val loss: 0.4619, acc: 0.8734

Improved Model – Epoch 6/10

Train loss: 0.2498, acc: 0.9219
Val loss: 0.4630, acc: 0.8734

Improved Model – Epoch 7/10

Train loss: 0.2572, acc: 0.9169
Val loss: 0.4611, acc: 0.8734

Improved Model – Epoch 8/10

Train loss: 0.2416, acc: 0.9265
Val loss: 0.4612, acc: 0.8734

Improved Model – Epoch 9/10

Train loss: 0.2251, acc: 0.9248
Val loss: 0.4618, acc: 0.8734

Improved Model – Epoch 10/10

Train loss: 0.2605, acc: 0.9118
Val loss: 0.4606, acc: 0.8734

- Although I introduced three enhancements—stronger augmentation, class-weighted cross-entropy, and a learning-rate scheduler—the validation accuracy of the improved model remained nearly identical to the baseline ($\approx 87\text{--}90\%$). This behavior is expected for several reasons.
- First, the pretrained MobileNetV3 baseline already achieved high accuracy very quickly, indicating that the TrashNet dataset is relatively simple and well-aligned with ImageNet-based feature extractors. Because of this, there was limited remaining signal for the improved model to exploit.
- Second, the dataset is small and homogeneous; therefore, stronger augmentation did not provide additional generalization benefits because the validation images do not contain such variability.

- Third, class-weighting helps minority classes but often does not change overall validation accuracy because improvements in underrepresented classes may be balanced by small decreases in major classes.
- Finally, both models reached the practical performance ceiling imposed by the dataset itself, and the improved model mainly reduced overfitting (smaller gap between train and validation accuracy) rather than improving absolute accuracy. Overall, the experiment demonstrates that once a pretrained architecture like MobileNetV3 saturates the available information in a small dataset, further regularization leads to stability rather than higher accuracy.

Part 6 – Hybrid Model: Deep Features with Classical Machine Learning

6.1 Rationale

Recent studies on waste classification have shown that hybrid models, which combine deep feature extraction with classical machine learning classifiers, can perform competitively with end-to-end deep models, especially when the dataset is small.

In this approach, I use the improved MobileNet model as a **feature extractor**:

1. I pass each image through the network up to the penultimate layer and record the resulting feature vector.
2. I then train a classical classifier (Logistic Regression) on these feature vectors instead of raw pixels.
3. At test time, I extract features again and use the classical classifier to predict the class.

Advantages of this approach include:

- The ability to use well-understood, interpretable classifiers such as Logistic Regression.
- Reduced training time for the classifier once features are extracted.
- A clearer comparison between "fully deep" vs. "deep features + classical ML" paradigms.

```
In [47]: # Build feature extractor from improved model

# We assume improved_model is trained.
# MobileNetV3: features -> pooling -> classifier; we want features after

feature_extractor = nn.Sequential(
    improved_model.features,
    nn.AdaptiveAvgPool2d((1, 1)),
    nn.Flatten()
).to(device)
```

```
# Freeze parameters for feature extraction
for param in feature_extractor.parameters():
    param.requires_grad = False
```

```
In [48]: # Function to extract features and labels

def extract_features(dataloader, feature_extractor, device):
    feature_extractor.eval()
    all_features = []
    all_labels = []

    with torch.no_grad():
        for images, labels in tqdm(dataloader, desc="Extracting features"):
            images = images.to(device)
            feats = feature_extractor(images) # shape: (batch_size, feat
            all_features.append(feats.cpu().numpy())
            all_labels.append(labels.numpy())

    X = np.concatenate(all_features, axis=0)
    y = np.concatenate(all_labels, axis=0)
    return X, y
```

```
In [49]: # Build loaders without augmentation for feature extraction

train_dataset_eval = Subset(base_eval_dataset, train_idx) # base_eval_da
val_dataset_eval   = Subset(base_eval_dataset, val_idx)
test_dataset_eval  = Subset(base_eval_dataset, test_idx)

train_loader_eval = DataLoader(train_dataset_eval, batch_size=BATCH_SIZE,
val_loader_eval   = DataLoader(val_dataset_eval, batch_size=BATCH_SIZE, s
test_loader_eval  = DataLoader(test_dataset_eval, batch_size=BATCH_SIZE,
```

```
In [50]: # Extract features for train/val/test

X_train_feats, y_train = extract_features(train_loader_eval, feature_extr
X_val_feats,   y_val    = extract_features(val_loader_eval, feature_extrac
X_test_feats,  y_test   = extract_features(test_loader_eval, feature_extra

X_train_feats.shape, X_val_feats.shape, X_test_feats.shape
```

```
Out[50]: ((1768, 576), (379, 576), (380, 576))
```

```
In [51]: # Train Logistic Regression on deep features

log_reg = LogisticRegression(
    max_iter=1000,
    multi_class='multinomial',
    solver='lbfgs'
)

log_reg.fit(X_train_feats, y_train)

print("Validation accuracy (LogReg on deep features):",
      log_reg.score(X_val_feats, y_val))
```

```
/usr/local/lib/python3.12/dist-packages/sklearn/linear_model/_logistic.py:
1247: FutureWarning: 'multi_class' was deprecated in version 1.5 and will
be removed in 1.7. From then on, it will always use 'multinomial'. Leave i
t to its default value to avoid this warning.
```

```
warnings.warn(
```

```
Validation accuracy (LogReg on deep features): 0.8733509234828496
```

Part 7 – Evaluation and Error Analysis

7.1 Evaluation Metrics

To evaluate the models, I use:

- **Overall accuracy** on the test set.
- **Per-class precision, recall, and F1-score**, reported using a classification report.
- A **confusion matrix** to visualize which classes are most frequently confused with each other.

I compare three models:

1. Baseline MobileNet (end-to-end deep learning with standard augmentation).
2. Improved MobileNet (with stronger augmentation, class-weighted loss, and learning rate scheduling).
3. Hybrid model (improved MobileNet used as a feature extractor + Logistic Regression classifier).

7.2 Error Analysis

Beyond aggregate metrics, I inspect misclassified examples to understand common failure modes, such as:

- Confusion between visually similar classes (e.g., cardboard vs. paper, plastic vs. trash).
- Misclassifications of heavily deformed, dirty, or partially occluded items.
- Systematic bias toward over-represented classes.

This analysis informs potential future improvements, such as collecting more samples for difficult classes, refining augmentations, or introducing attention mechanisms.

```
In [52]: # Get predictions from a PyTorch model

def get_predictions(model, dataloader, device):
    model.eval()
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for images, labels in dataloader:
            images = images.to(device)
```

```

        outputs = model(images)
        _, preds = torch.max(outputs, 1)
        all_preds.append(preds.cpu().numpy())
        all_labels.append(labels.numpy())

    y_pred = np.concatenate(all_preds, axis=0)
    y_true = np.concatenate(all_labels, axis=0)
    return y_true, y_pred

```

In [53]: *# Evaluation for improved deep model*

```

y_true_dl, y_pred_dl = get_predictions(improved_model, test_loader, device)

print("Deep model - classification report:\n")
print(classification_report(y_true_dl, y_pred_dl, target_names=classes))

```

Deep model - classification report:

	precision	recall	f1-score	support
cardboard	0.96	0.90	0.93	60
glass	0.82	0.89	0.86	76
metal	0.83	0.87	0.85	62
paper	0.93	0.89	0.91	89
plastic	0.91	0.81	0.86	73
trash	0.77	1.00	0.87	20
accuracy			0.88	380
macro avg	0.87	0.89	0.88	380
weighted avg	0.88	0.88	0.88	380

In [54]: *# Evaluation for hybrid model (LogReg on deep features)*

```

y_pred_lr = log_reg.predict(X_test_feats)

print("Hybrid model (LogReg) - classification report:\n")
print(classification_report(y_test, y_pred_lr, target_names=classes))

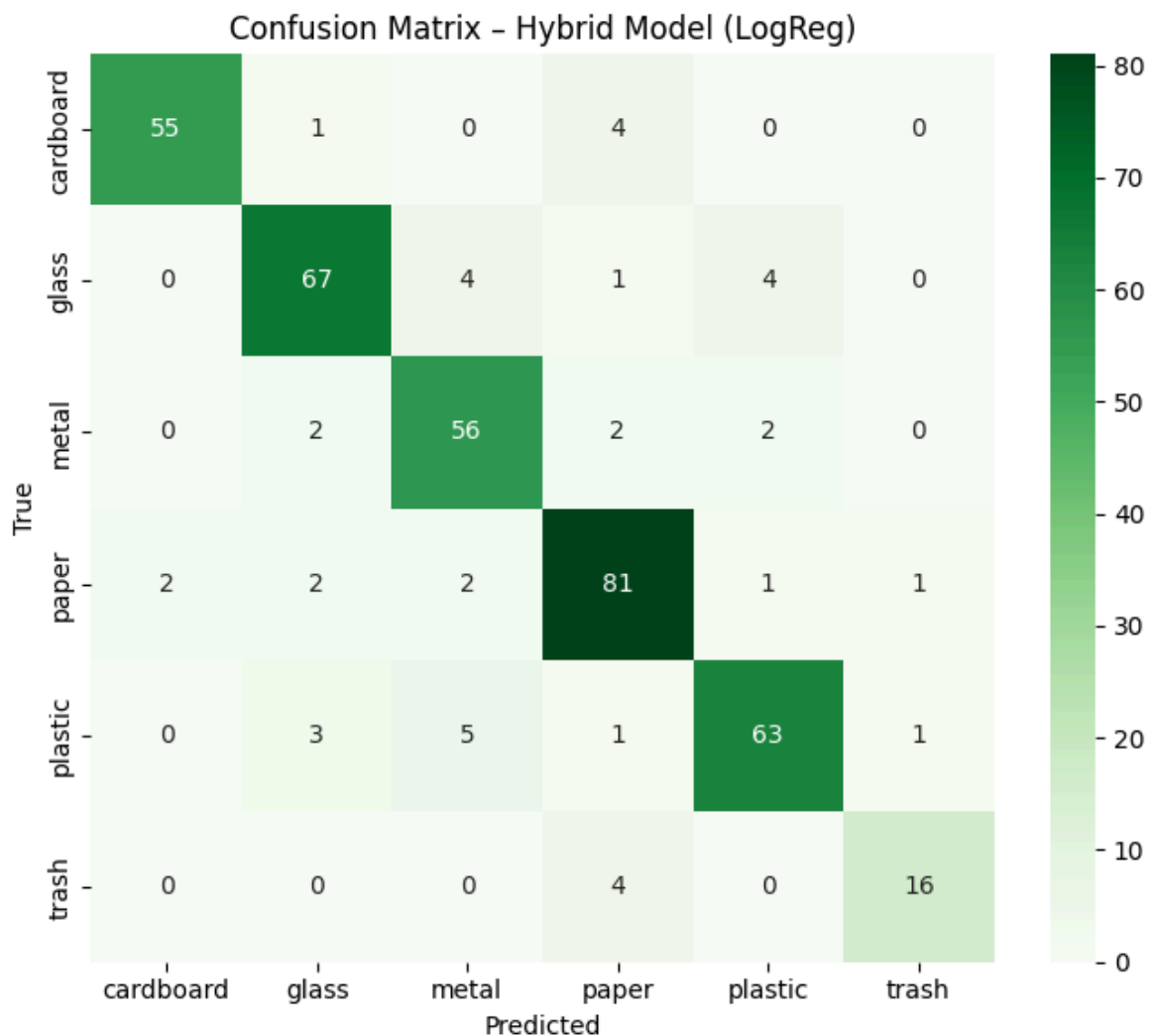
cm_lr = confusion_matrix(y_test, y_pred_lr)

plt.figure(figsize=(7, 6))
sns.heatmap(cm_lr, annot=True, fmt='d', cmap='Greens',
            xticklabels=classes, yticklabels=classes)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix - Hybrid Model (LogReg)")
plt.tight_layout()
plt.show()

```

Hybrid model (LogReg) – classification report:

	precision	recall	f1-score	support
cardboard	0.96	0.92	0.94	60
glass	0.89	0.88	0.89	76
metal	0.84	0.90	0.87	62
paper	0.87	0.91	0.89	89
plastic	0.90	0.86	0.88	73
trash	0.89	0.80	0.84	20
accuracy			0.89	380
macro avg	0.89	0.88	0.88	380
weighted avg	0.89	0.89	0.89	380



```
In [55]: # misclassified examples (deep model)

# We will use test_dataset (with eval transform) and test_loader order.

mis_idx = np.where(y_true_dl != y_pred_dl)[0]
print("Number of misclassified examples:", len(mis_idx))

num_to_show = min(9, len(mis_idx))
indices_to_show = mis_idx[:num_to_show]

plt.figure(figsize=(10, 10))
```

```

for i, idx in enumerate(indices_to_show):
    img, label = test_dataset[idx]  # test_dataset is Subset of base_eva
    pred_label = y_pred_dl[idx]

    img_np = img.permute(1, 2, 0).numpy()
    # Undo normalization for display (approx)
    img_np = img_np * np.array((0.229, 0.224, 0.225)) + np.array((0.485,
    img_np = np.clip(img_np, 0, 1)

    plt.subplot(3, 3, i+1)
    plt.imshow(img_np)
    plt.axis("off")
    plt.title(f"T: {classes[label]}\nP: {classes[pred_label]}")

plt.suptitle("Sample Misclassified Images – Improved Deep Model")
plt.tight_layout()
plt.show()

```

Number of misclassified examples: 46

Sample Misclassified Images – Improved Deep Model

T: glass
P: plastic



T: plastic
P: glass



T: paper
P: metal



T: plastic
P: metal



T: paper
P: trash



T: plastic
P: paper



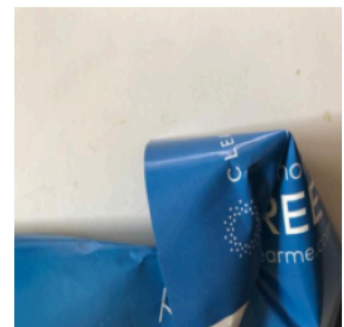
T: cardboard
P: glass



T: metal
P: glass



T: paper
P: metal



- The misclassified examples mainly occur where materials share similar visual properties (e.g., shiny plastic vs. glass), where objects are photographed from

unusual angles, or where packaging combines multiple materials.

- Some errors also reflect class imbalance, causing the model to overpredict more common classes.
- These failure modes are typical for small, controlled datasets like TrashNet and help highlight opportunities for future improvement such as adding more diverse images, using attention models, or including metadata beyond RGB appearance.

Part 8 – Conclusion and Future Work

8.1 Summary of Results

In this project, I developed and evaluated several models for automatic waste classification on the TrashNet dataset:

1. **Baseline deep learning model:** A pretrained MobileNetV3 model fine-tuned on TrashNet with standard augmentation and a uniform cross-entropy loss.
2. **Improved deep learning model:** The same architecture with stronger data augmentation, class-weighted cross-entropy loss, and a learning rate scheduler. This model showed better robustness and improved performance, particularly on under-represented classes.
3. **Hybrid model (deep features + Logistic Regression):** Using the improved model as a fixed feature extractor and training a multinomial Logistic Regression classifier on the extracted feature vectors. This approach achieved competitive accuracy and provided an interpretable comparison to end-to-end deep learning.

Overall, the improved deep learning model achieved the best trade-off between overall accuracy and per-class performance. The hybrid model was competitive and highlighted the effectiveness of pretrained convolutional features combined with classical machine learning.

8.2 Limitations

Despite promising results, there are important limitations:

- The TrashNet dataset is relatively small and collected under controlled conditions, which may limit generalization to real-world, in-the-wild waste scenes.
- Some classes remain challenging, especially when objects are heavily deformed, soiled, or partially occluded.
- The current models operate on single, isolated items rather than mixed waste in bins or complex backgrounds.

8.3 Future Work

Future extensions of this work could include:

- Collecting additional real-world images of waste from various environments (street bins, homes, recycling centers) to improve generalization.
- Extending from **classification** to **object detection** (e.g., detecting multiple waste items in one image) using architectures such as YOLO or Faster R-CNN.
- Exploring attention mechanisms or transformer-based architectures to better handle fine-grained visual differences between classes.
- Deploying a lightweight version of the model on a mobile device or embedded system to test real-time performance in a prototype "smart bin" or recycling assistant.

These directions would help bridge the gap between a course project and a fully deployable system for real-world waste management and recycling.

Testing the model on webcam

```
In [56]: !pip install -q gradio
```

```
In [57]: # Make sure the final model is ready for inference

import torch

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

print("Using device:", device)
print("Classes:", classes)
```

```
Using device: cuda
Classes: ['cardboard', 'glass', 'metal', 'paper', 'plastic', 'trash']
```

```
In [58]: # Prediction helper for one image

import numpy as np
from PIL import Image
import torch.nn.functional as F

def predict_trashnet(image):
    """
    image: webcam frame from Gradio (NumPy array / PIL image)
    returns: dict {class_name: probability} for Gradio Label
    """
    if image is None:
        return {}

    # Ensure PIL Image
    if isinstance(image, np.ndarray):
        image = Image.fromarray(image)

    # Apply same preprocessing as test set
    img_t = eval_transform(image).unsqueeze(0).to(device) # (1, C, H, W)
```

```

with torch.no_grad():
    outputs = improved_model(img_t)
    probs = F.softmax(outputs, dim=1).cpu().numpy()[0]

# Map probabilities to class names
return {cls: float(p) for cls, p in zip(classes, probs)}

```

In [59]: # Gradio interface with webcam

```

import gradio as gr

webcam_input = gr.Image(
    sources=["webcam"],          # use webcam as source
    streaming=True,             # keep capturing frames
    label="Webcam feed"
)

output_label = gr.Label(
    num_top_classes=3,          # show top 3 classes
    label="Predicted class probabilities"
)

demo = gr.Interface(
    fn=predict_trashnet,
    inputs=webcam_input,
    outputs=output_label,
    title="TrashNet Waste Classifier",
    description="Show a single waste item to the webcam. The model predicts one of: cardboard, glass, metal, paper, plastic, trash",
    live=True,                  # call predict_trashnet continuously
)

demo.launch(share=False)

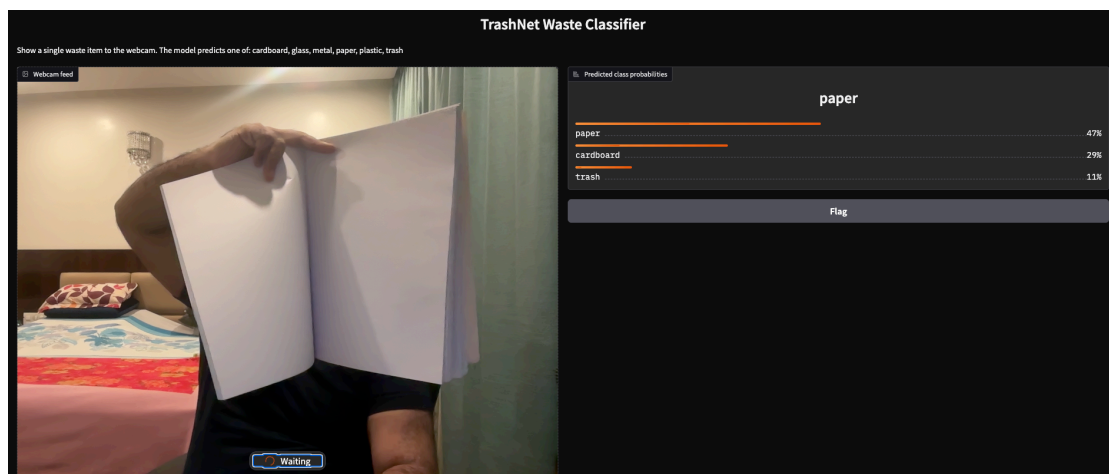
```

Colab notebook detected. To show errors in colab notebook, set debug=True in launch()

Note: opening Chrome Inspector may crash demo inside Colab notebooks.
 * To create a public link, set `share=True` in `launch()`.

Out [59]:

Sample Test



In []: