Smart Waste Classification and Recycling Assistant

Dataset – Garbage Classification

Dataset: Garbage Classification

– 6 classes: cardboard, glass, metal, paper, plastic, trash.
Typical dataset layout:

```
/kaggle/input/garbage-classification/
│
├── Garbage classification/        <-- sometimes an inner
folder with the same name
│   ├── cardboard/                 <-- image files inside
│   ├── glass/
│   ├── metal/
│   ├── paper/
│   ├── plastic/
│   └── trash/
│
├── one-indexed-files-notrash_train.txt
├── one-indexed-files-notrash_val.txt
├── one-indexed-files-notrash_test.txt
├── one-indexed-files.txt
├── zero-indexed-files.txt
```

Report – Table of Contents (Arranged)

Waste Classification Report

Author: Abdelrhman Wael Ahmeda

Executive Summary

This report summarizes the experiments, models, and evaluation performed across several notebooks that together address multi-class garbage classification and supporting research directions. The core classification work compares a MobileNetV2 transfer-learning approach and a custom convolutional neural network (CNN) trained from scratch on a subset of the Garbage Classification dataset (six classes: cardboard, glass, metal, paper, plastic, trash). Beyond classification, the project includes complementary experiments and tools to improve robustness and enable deployment:

- Denoising Autoencoder: a PyTorch encoder–decoder trained to remove additive noise and mild blur from images, producing denoised outputs for qualitative review and potential preprocessing of classifier inputs.
- GANs (Data Augmentation): a conditional DCGAN-style pipeline to synthesize class-conditional 64x64 images for augmenting under-represented classes.
- Multimodal (CLIP-based) classifier: computes CLIP image and text embeddings, concatenates them, and trains a lightweight MLP classifier on the joint embeddings to leverage short text descriptions alongside images.
- YOLO (object detection) extension: converts the classification dataset to YOLO format and provides scripts/notebooks to train and visualize a YOLO detector for localization tasks.
- Streamlit app: an interactive UI that loads saved models (MobileNetV2.h5 and custom_CNN.h5), provides sample images, and runs inference for quick demonstrations and manual inspection.

Artifacts produced:
- Trained model files (MobileNetV2.h5, custom_CNN.h5), denoiser checkpoints and outputs, GAN checkpoints and generated samples, multimodal embeddings and classifier checkpoint, and the Streamlit app script (streamlit_app.py).

How to reproduce (high level):

1. Place the Garbage Classification dataset at the path expected by the notebooks or update the INPUT_ROOT/ROOT_PATH variables used in each notebook.
2. Install required packages per-experiment (TensorFlow/Keras for classification, PyTorch/torchvision for denoiser/GAN/multimodal, transformers for CLIP, ultralytics for YOLO if used, and streamlit for the app).
3. Run the notebooks in order for the experiments you wish to reproduce. Each notebook saves checkpoints and example outputs to the working directories described in the notebook headers.

For exact numeric results (accuracies, confusion matrices, classification reports), run the evaluation cells in the corresponding notebooks. The following notebooks hold the main code and outputs:
- Deep Learning & CNNs (classification experiments)
- Denoising Autoencoder (denoiser)
- GANs (data augmentation)
- Multimodal Extension (CLIP-based classifier)
- YOLO (object detection) extension
- streamlit_app.py (interactive demo)

This report highlights methodology, trade-offs, and recommended next steps for improving model performance and deployment readiness.

Dataset and Preprocessing

Dataset source: The notebook uses the Garbage Classification dataset (commonly available on Kaggle) located in the local path used in the notebook. From this dataset, six classes were selected: metal, glass, paper, trash, cardboard, and plastic. The notebook reads file paths and labels into a pandas DataFrame, normalizes class names (for example mapping 'white-glass' to 'glass'), and shuffles the list of image paths prior to splitting.

Splitting strategy: The notebook implements a stratified-like split by grouping by class and splitting a fixed ratio per class. A dedicated function selects approximately 85% of each class for training and 15% for testing/validation to ensure each class is represented in both sets.

Image preprocessing and augmentation: Two ImageDataGenerator instances are used. The training generator applies rescaling (1./255), zoom_range, horizontal and vertical flips, and rotation_range to produce augmented images on the fly. The test generator only rescales images (1./255) and uses shuffle=False for consistent evaluation. Images are resized to IMG_SIZE = (224, 224) for the MobileNetV2 model and (150, 150) for the custom CNN experiment.

Batching and balancing: Training used batch sizes of 64 for some generators and 8 for test evaluation in certain cells. The notebook calculates steps_per_epoch and validation_steps using generator sample counts and batch sizes. A class_weight dictionary is computed from training class counts to address class imbalance if needed.

Modeling Approaches

A. MobileNetV2 transfer learning setup
- Base model: MobileNetV2 with weights pre-trained on ImageNet and include_top=False to remove the dense classifier head.
- Input shape: 224x224x3.
- Additional head layers appended in the notebook: Flatten(), Dense(64, activation='relu'), BatchNormalization(), Dropout(0.08), and final Dense(6, activation='softmax').
- Freezing strategy: Most layers of the MobileNetV2 base are frozen except for the last few layers (the code leaves the last 3 layers trainable while freezing earlier layers).
- Compilation: optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'].
- Callbacks: EarlyStopping monitoring validation accuracy with patience=4 and restore_best_weights=True is used during training.

B. Custom CNN architecture
- Input shape: 150x150x3 for the custom CNN pipeline.
- Layers: Sequential stack of Conv2D and MaxPooling2D layers with increasing filters (64 -> 128 -> 256 -> 512), followed by Flatten(), Dense(512, activation='relu'), Dropout(0.5), and Dense(6, activation='softmax').
- Compilation: Adam optimizer with learning_rate=0.001, loss='categorical_crossentropy', and metrics=['accuracy'].
- Training: The custom model is trained using ImageDataGenerator with validation_split=0.2, batch_size=64, and up to 30 epochs in the notebook.

Training Procedure

Batch sizes and epochs: MobileNetV2 training uses batch_size=64 for training generator and batch_size=8 for test generator in a subset of cells. The provided training loop sets epochs=50 for the MobileNetV2 experiment (with early stopping). The custom CNN used epochs=30 in the notebook.

Steps calculation: Steps per epoch are computed as trainGenerator.samples // trainGenerator.batch_size; validation_steps as ceil(testGenerator.samples / testGenerator.batch_size). These values ensure full coverage of the training and validation data.

History objects: The notebook stores returned History objects from Model.fit into variables named history (for MobileNetV2) and history_custom (for the custom CNN), enabling plotting of training/validation accuracy and loss curves after training.

Evaluation and Results

Evaluation metrics used in the notebook:
- Test accuracy: computed via Model.evaluate(testGenerator). This provides a global scalar accuracy over the test set (refer to the notebook training and evaluation cells to see numeric values).
- Confusion matrix: computed with sklearn.metrics.confusion_matrix using true and predicted class indices, then visualized with seaborn heatmap to analyze per-class performance and common misclassifications.
- Classification report: generated with sklearn.metrics.classification_report to show precision, recall, f1-score, and support for each class.
- Visual inspection: several cells sample batches, display predicted vs. true labels for individual images, and create gallery plots of predictions for qualitative assessment.

Please run the notebook cells to obtain exact numeric values. In this report, references to notebook cell numbers (starting from 1 in the original file) are provided to help quickly locate training and evaluation code:
- Model training (MobileNetV2): Cell 27 (Model.fit for MobileNetV2)
- Model evaluation (MobileNetV2): Cells 34-39 (prediction, evaluation, confusion matrix, classification report)
- Custom CNN training: Cell 58 (history_custom fit)
- Custom CNN evaluation: Cell 60 (evaluate on validation generator)

- Saved model filenames: MobileNetV2.h5 (Cell 29) and custom_CNN.h5 (Cell 64).

Analysis and Comparison

Expected strengths of MobileNetV2:
- Pretrained weights on ImageNet provide strong low-level features and transfer learning usually converges faster with less data.
- Lower parameter count and efficient architecture make MobileNetV2 suitable for on-device or edge inference scenarios.

Expected strengths of the Custom CNN:
- Architecture can be tailored to the specific dataset characteristics and may capture domain-specific features if trained sufficiently.
- With sufficient data and training time, custom models can match or exceed transfer-learned models but typically require more compute and careful regularization.

Trade-offs and practical considerations:
- Training from scratch (custom CNN) requires more labeled data and more epochs to generalize; it is more sensitive to overfitting without strong augmentation and regularization.
- Transfer learning (MobileNetV2) benefits from pretrained filters; it is often the pragmatic first choice when dataset size is limited.
- Inference latency and model size favor MobileNetV2 over a bulky custom CNN; however, a well-pruned or quantized custom model can also be competitive.

Denoising Autoencoder

Overview: A denoising autoencoder was implemented to learn to remove noise and mild blur from images in the Garbage Classification dataset. The notebook builds a convolutional encoder–decoder in PyTorch, trains it on 64x64 crops, and saves checkpoints and denoised example outputs for qualitative review.

Dataset and preprocessing:
- The notebook locates the dataset under /kaggle/input/garbage-classification/Garbage classification and detects the correct inner folder. Images are loaded using torchvision.datasets.ImageFolder and resized/center-cropped to IMGSZ = 64.

- Transform pipeline ensures images are converted to tensors in [0,1]. A DataLoader is instantiated with BATCH_SIZE=64 and NUM_WORKERS=2.
- During training, synthetic corruption is applied per-batch: additive Gaussian noise (NOISE_SIGMA=0.15) and probabilistic Gaussian blur (BLUR_PROB=0.15) applied to individual images.

Model architecture and training:
- The ConvDenoiser model is a lightweight convolutional encoder-decoder:
  - Encoder: Conv2d layers [3 -> 64 -> 128 -> 256] with ReLU activations and downsampling (stride=2).
  - Decoder: ConvTranspose2d layers [256 -> 128 -> 64 -> 3] with ReLU and final Sigmoid to output [0,1].
- Training configuration: EPOCHS=200, LR=1e-3, optimizer=Adam, criterion=MSELoss. The model trains on DEVICE = 'cuda' if available else 'cpu'.
- A checkpointing strategy saves model states every SAVE_EVERY epochs to CHECKPOINT_DIR.

Noise model and augmentation:
- add_noise(batch) adds Gaussian noise (clipped to [0,1]) and randomly applies a PIL Gaussian blur to individual images with BLUR_PROB probability. This aims to teach the autoencoder to remove both additive noise and mild smoothing artifacts.

Outputs and artifacts:
- During training the notebook saves preview grids (clean / noisy / recon) to EXAMPLES_DIR for visual inspection after each epoch.
- After training, the notebook runs the model on up to M=100 dataset images and saves reconstructed denoised images to OUT_ROOT/denoised_outputs. The notebook can also zip these outputs for download.

Reproducibility and notes:
- Required files: the garbage-classification dataset placed at the path referenced in the notebook. The notebook writes outputs under /kaggle/working/denoiser_out by default.
- To reproduce: run the cells sequentially; ensure torch, torchvision, and other dependencies are installed; adjust batch size or workers for your environment.

Integration with classification pipeline:
- The denoiser can be used as a preprocessing step for the classification models to reduce noise in captured images. For best results, retrain or fine-tune classifiers on denoised images or perform domain adaptation.

Reference: The denoising experiment is available at ./Denoising Autoencoder/denoising-autoencoder-notebook.ipynb – run its cells (1–n) to reproduce model training and output generation.

GANs (Data Augmentation)

Overview: A conditional GAN (DCGAN-style) was implemented to synthesize class-conditional 64x64 images for data augmentation. The notebook implements a labeled generator and discriminator to produce images conditioned on class labels and trains them using adversarial loss with label smoothing and checkpointing.

Dataset and preprocessing:
- The notebook uses the Garbage Classification dataset located under /kaggle/input/garbage-classification/Garbage classification and auto-detects the inner folder structure.
- Images are transformed to 64x64 and normalized to the range [-1, 1], consistent with a tanh generator output.
- DataLoader is created with BATCH_SIZE=64 and NUM_WORKERS=2 for training.

Model architectures:
- Generator: projects a Z_DIM latent vector and concatenates a learned label map that is upsampled. The generator uses a sequence of ConvTranspose2d layers to upsample from 4x4 to 64x64, producing RGB outputs with Tanh activation.
- Discriminator: receives the image concatenated with a channel-wise label map and applies a conv stack that reduces spatial size to a single logit. The discriminator outputs logits used with BCEWithLogitsLoss.
- Weight initialization: convolutional and linear layers are initialized from a normal distribution (mean=0, std=0.02).

Training setup and losses:
- Adversarial training follows the standard alternating updates: update discriminator on real images and generated

fakes, then update generator to maximize discriminator logits on generated samples.
- Label smoothing is applied to real labels (real_label_val=0.9) to stabilize training.
- Optimizers: Adam with LR=2e-4 and betas=(0.5, 0.999).
- Training is configured for many epochs (e.g., EPOCHS=800 in the notebook) with periodic checkpointing and sample grids saved per epoch.

Outputs and artifacts:
- Checkpoints are stored under OUT_ROOT/checkpoints (files like dcgan_epoch{n}.pth) and sample grids are saved under OUT_ROOT/samples/epoch_{n}_samples.png.
- The notebook includes a generation step that uses the trained generator to synthesize N_GEN_PER_CLASS images per class and saves them to OUT_ROOT/gan_augmented/<class>/gen_*.png for downstream augmentation.

Integration and recommended usage:
- Generated images can be used to augment the real training data for classes with limited examples; ensure synthetic images are curated to avoid label noise.
- Consider filtering generated images with a classifier (out-of-distribution detection) or manual curation before mixing into the training set.
- Alternative approaches: use conditional diffusion models or class-conditional StyleGAN for higher-fidelity synthesis if compute permits.

Reproducibility notes:
- The notebook expects PyTorch and torchvision available in the environment and will detect GPU if present. Adjust NUM_WORKERS and BATCH_SIZE for your environment.
- To reproduce results, run the cells in order. Checkpoints and samples are produced during training and can be used to resume or generate augmented datasets.

Reference: The GANs experiment is available at ./GANs/GANs_notebook.ipynb — run its cells (1–n) to reproduce training and generated outputs.

Multimodal Extension (CLIP-based Image + Text)

Overview: A CLIP-based multimodal pipeline was developed to combine image embeddings and short textual descriptions for classification. The notebook computes CLIP image and text embeddings, concatenates them, and trains a lightweight MLP classifier on the concatenated features.

Workflow and preprocessing:
- The notebook loads CLIP (openai/clip-vit-base-patch32) via transformers and processor utilities.
- Image preprocessing uses CLIP's image processor (224x224) and short text descriptions are auto-generated per class (e.g., "a photo of a {class}").
- A dataset splitter creates train/val lists; embeddings are computed and saved under embeddings/ for both images and text.

Embedding and classifier pipeline:
- compute_embeddings() maps batches of (image, text) pairs through CLIP to produce normalized image and text embeddings which are saved to disk.
- Embeddings are concatenated (image || text) to form classifier inputs. A TensorDataset and DataLoader are built for training.
- Classifier: a simple MLP (Linear -> ReLU -> Dropout -> Linear) trained with CrossEntropyLoss and Adam for ~20 epochs. Best model by validation F1 is saved.

Evaluation and outputs:
- After training, the notebook loads the best checkpoint and prints validation accuracy, weighted F1, classification_report, and a confusion matrix heatmap.
- An inference helper function (infer) encodes a single image+text pair through CLIP and the trained classifier to return predicted class and probabilities.

Use-cases and benefits:
- Multimodal inputs (image+text) can improve robustness and accuracy when short descriptive text is available (e.g., sensor metadata, user captions, or pre-labeled descriptions).
- This approach enables flexible extensions: replace text with richer metadata or use per-image human captions to boost performance.

Reproducibility notes:
- Requires transformers (Hugging Face), torch, torchvision, scikit-learn. Computation of embeddings can be done on GPU for speed.
- To reproduce: run the notebook cells sequentially to compute embeddings, train the classifier, and evaluate results. Saved embeddings and model weights are stored under /kaggle/working/multimodal_clip/ by default.

Reference: The multimodal notebook is available at ./Multimodal Extension/multimodal-waste-notebook.ipynb — run its cells (1–n) to reproduce the experiments and outputs.

YOLO (Object Detection) Extension

Overview: The accompanying YOLO notebook converts the classification dataset into YOLO format (full-image bounding boxes) and trains an object detection model (YOLOv8 or YOLOv5 fallback). This section summarizes the steps implemented in the YOLO notebook, maps them to notebook cell numbers for quick navigation, and provides notes on execution, outputs, and how to integrate detection results with the previously described classification experiments.

Cell-by-cell mapping (YOLO notebook):
- Cell 1: Setup & imports — installs required packages (ultralytics, tensorflow, scikit-learn, matplotlib, pillow) and imports Python modules.
- Cell 2: Configuration — defines dataset ROOT_PATH, output folder OUT, class list, CLASS_TO_ID, and seed.
- Cell 3: Inspect dataset — auto-detects class folders and prints counts per class.
- Cell 4: Preview images — displays one sample image per class for quick visual inspection.
- Cell 5: Convert classification → YOLO format — collects all images, attempts to use provided train/val split lists, otherwise performs a stratified train/val split and prepares lists.
- Cell 6: Save split & labels — copies images into OUT/train and OUT/val and writes corresponding YOLO .txt label files (full-image boxes: "cid 0.5 0.5 1.0 1.0").
- Cell 7: Create dataset.yaml — writes dataset.yaml with train/val paths and names mapping for YOLO training.
- Cell 8: Verify produced files — checks that expected image and label folders exist and prints a few sample labels.
- Cell 9: Train YOLOv8 — installs ultralytics if necessary, instantiates a YOLO model (yolov8n.pt) and calls model.train(...) with specified epochs, imgsz, batch size, and output name.
- Cell 10: Prediction & visualization — after training, the notebook runs model.predict on the validation images and saves

annotated visual outputs under runs/detect/<run>/predict.
Additional cells gather available .pt checkpoints, locate validation images, and produce sample annotated images for qualitative inspection.

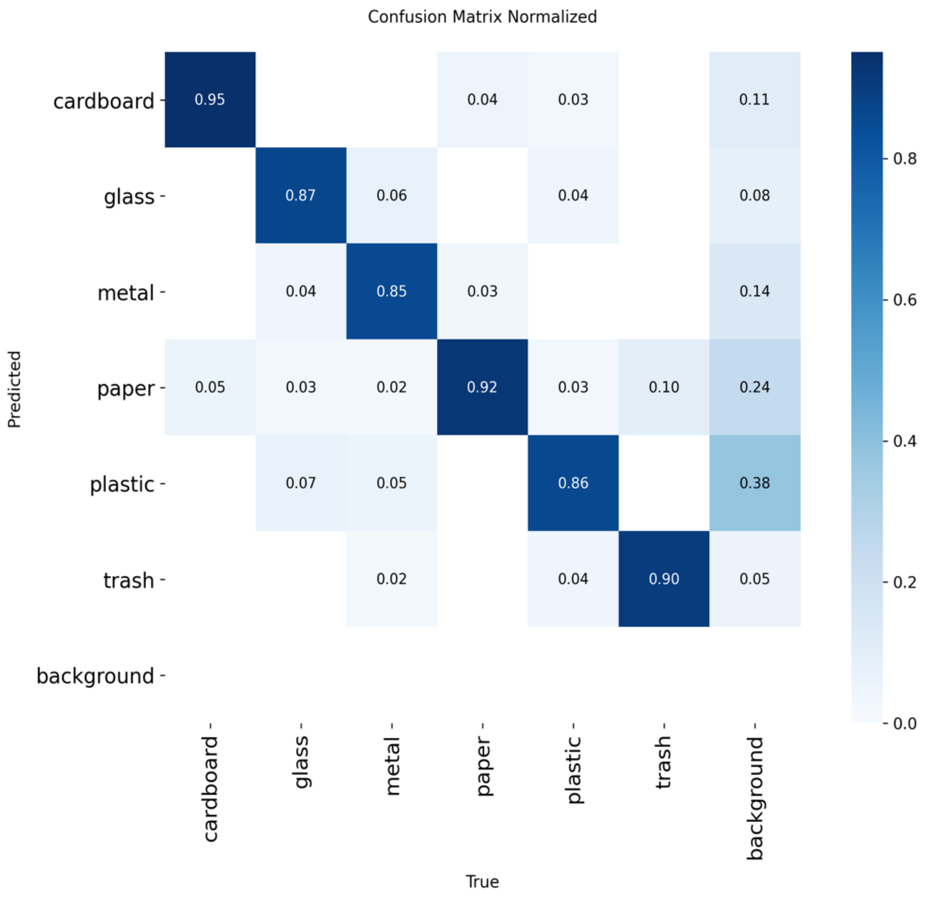
Execution notes and outputs:
- Training metrics: The YOLO API prints validation metrics including mAP and precision/recall when model.val() is called. The notebook deliberately avoids hardcoding numeric results; run the cells to obtain current metrics.
- Checkpoints and artifacts: Trained weights and checkpoints are saved under the runs directory (e.g., /kaggle/working/runs). The notebook looks for best.pt, last.pt, or the newest .pt when loading weights for prediction and visualization.
- Visual predictions: Annotated images are saved to runs/detect/<run>/predict for quick review. Use those images to verify detection quality and common failure cases.

How YOLO complements classification:
- Detection + classification pipeline: Use the detector to localize objects in images and then run the classification model on cropped detections to improve class-specific accuracy and reduce background noise.
- Data augmentation and localization: Detection models benefit from multi-scale training, mosaic augmentation, and jittering; these can improve robustness when combined with classification augmentation strategies.
- Use-cases: YOLO is useful when images contain multiple objects or complex scenes where a full-image single-label classifier is insufficient. It also enables counting and localization metrics for downstream analytics.

Recommendations and next steps for YOLO experiments:
- Evaluate per-class AP and confusion of detections; analyze false positives and missed detections using the saved annotated images.
- Tune training hyperparameters: epochs, imgsz, batch size, learning rate schedule, and augmentation mix (mosaic, mixup), and consider using larger backbones (yolov8s/yolov8m) if compute allows.
- Export and deploy: Export the best checkpoint to ONNX/TFLite (ultralytics export) for edge inference and apply quantization/pruning as needed.
- Integrate with classification models: Crop detected boxes and run the MobileNetV2 or custom CNN on crops to build a two-stage detector-classifier pipeline; compare single-stage vs two-stage accuracy and latency.

Reproducibility checklist specific to YOLO section:
- Ensure ultralytics (YOLOv8) is installed and GPU is available for faster training.
- Confirm OUT and dataset.yaml paths are correct and that label files exist under train/labels and val/labels.
- Inspect saved checkpoints under runs and use model.val() to obtain final mAP; locate annotated predictions under runs/detect/<run>/predict for qualitative checks.

Reference: the YOLO notebook included with this project is located at ./YOLO/yolo-notebook.ipynb — run its cells in order (Cells 1–10 as summarized above) to reproduce dataset conversion, training, and predicions.



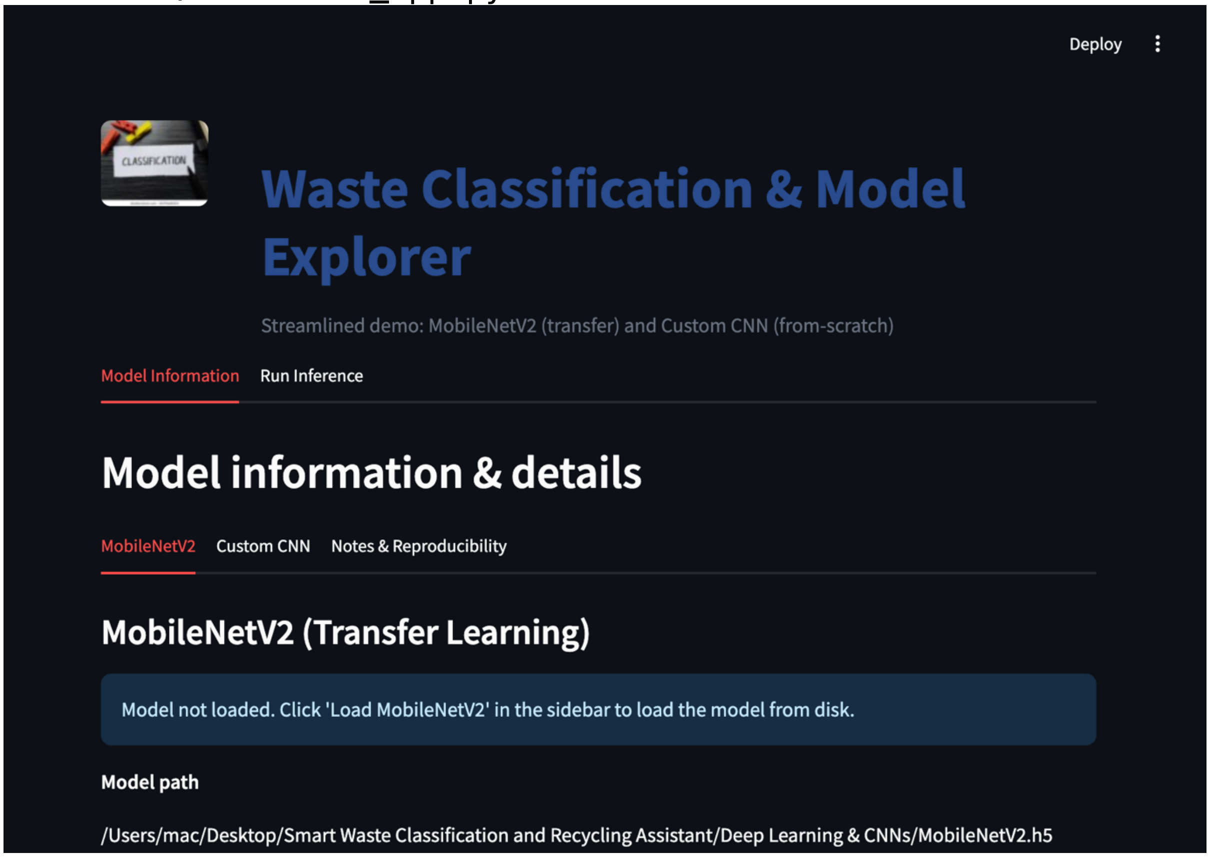
plastic244.jpg
metal213.
plastic332.jpg
paper389.jpg
glass439.jpg
glass87.jpg
paper43.jpg
paper311.jpg
glass34.jpg
glass251.jpg
0
1
2
3
4

metal
metal182.jpg
metal201.jpg
metal162.jpg
metal187.jpg
metal202.jpg
metal169.jpg
metal191.jpg
metal204.jpg
metal164.jpg
metal170.jpg

Web Application — Streamlit

A companion Streamlit web application was developed to run inference with the trained classification models and to provide a lightweight user interface for evaluation and demonstration. The app file is included in the project at:

- /Users/mac/Desktop/Smart Waste Classification and Recycling Assistant/streamlit_app.py



Waste Classification & Model Explorer

Streamlined demo: MobileNetV2 (transfer) and Custom CNN (from-scratch)

Model information & details

MobileNetV2 (Transfer Learning)

Model not loaded. Click 'Load MobileNetV2' in the sidebar to load the model from disk.

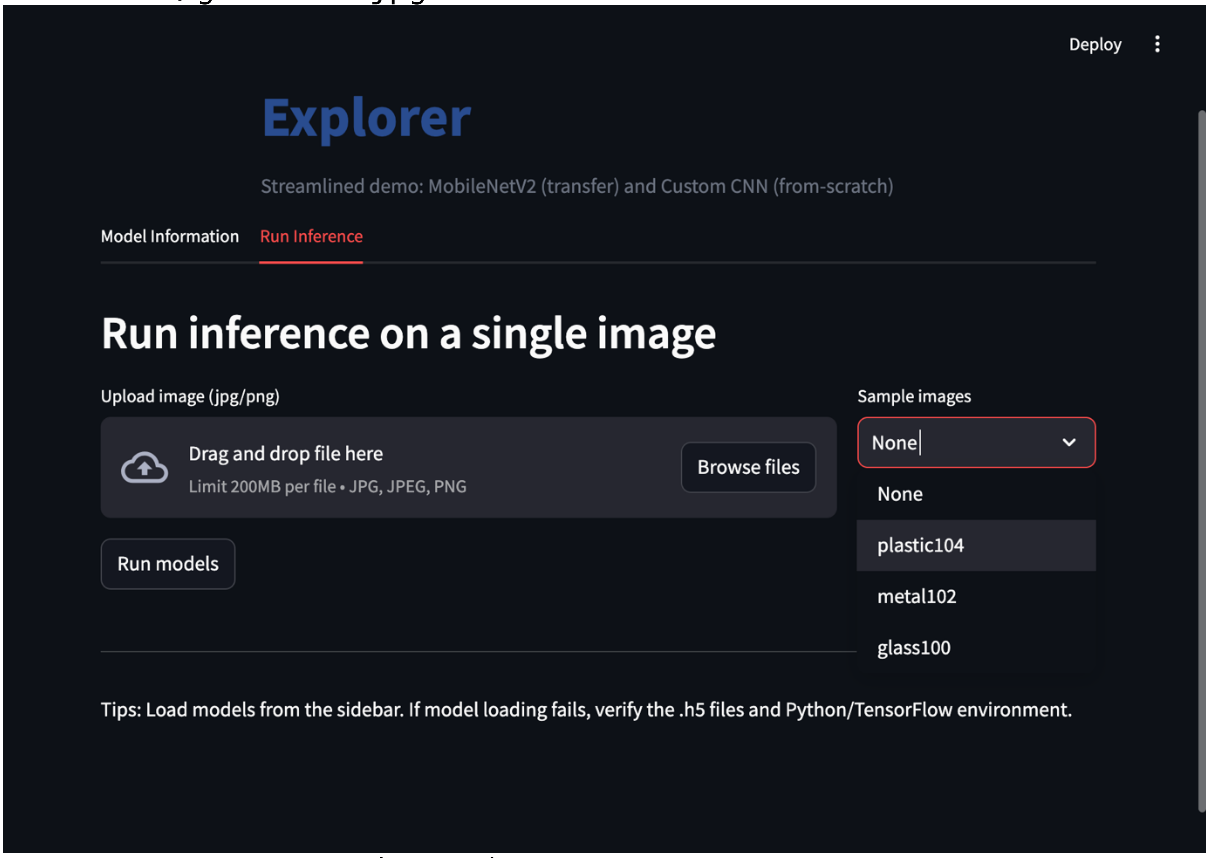
Model path

/Users/mac/Desktop/Smart Waste Classification and Recycling Assistant/Deep Learning & CNNs/MobileNetV2.h5

Key features
- Load pretrained TensorFlow Keras models saved by the notebook (MobileNetV2.h5 and custom_CNN.h5).
- Select and preview an input image (upload or choose one of three project sample images).
- Run inference with MobileNetV2 and the Custom CNN and display predicted class and per-class probabilities.
- Simple model information panel that shows model summaries (when models are loaded) and reproducibility notes.
- Clean UX with spinners and status badges indicating model load state.

Included sample images (bundled paths used by the app)
- /Users/mac/Desktop/Smart Waste Classification and Recycling Assistant/plastic104.jpg

- /Users/mac/Desktop/Smart Waste Classification and Recycling Assistant/metal102.jpg
- /Users/mac/Desktop/Smart Waste Classification and Recycling Assistant/glass100.jpg



Streamlined demo: MobileNetV2 (transfer) and Custom CNN (from-scratch)

Run inference on a single image

Tips: Load models from the sidebar. If model loading fails, verify the .h5 files and Python/TensorFlow environment.

How to run the app (local)

1. Create and activate a Python environment and install dependencies, e.g.:
   - pip install streamlit tensorflow pillow numpy

   (If you want GPU acceleration, install a GPU-compatible tensorflow build matching your system.)
2. Ensure the trained model files are present:
   - Deep Learning & CNNs/MobileNetV2.h5
   - Deep Learning & CNNs/custom_CNN.h5
3. From the project root run:
   - streamlit run "./streamlit_app.py"
4. Open the address shown in the terminal (default http://localhost:8501) and use the sidebar to load models and the Run Inference tab to test images.

Integration notes

- The Streamlit app uses the same preprocessing resolutions and normalization applied in the notebook: MobileNetV2 expects 224x224, custom CNN expects 150x150, and both use rescaling by 1./255.
- The app loads the saved .h5 files produced by the notebook; any changes to model architecture or class ordering require updating the app mapping (CLASS_NAMES) or re-saving models.

Author
- App integrated into the project by Abdelrhman Wael Ahmeda.

Conclusion

This project evaluated multiple approaches for automated waste classification and explored complementary techniques to improve robustness and deployment readiness. Key takeaways:

- Transfer learning with MobileNetV2 provides an efficient, accurate baseline for multi-class waste classification and is well-suited for edge or on-device inference after pruning/quantization.
- A custom CNN trained from scratch can reach competitive performance but typically requires more data, careful augmentation, and regularization to avoid overfitting.
- Denoising autoencoders and GAN-based augmentation are valuable tools: the denoiser improves input quality and GANs can synthesize additional examples for under-represented classes, but generated images should be curated before use in training.
- The multimodal CLIP-based pipeline demonstrates that combining image and short text signals can improve robustness when textual metadata or captions are available.

Limitations:
- Reported procedures depend on dataset placement, environment configuration, and available compute (GPU recommended for training and CLIP embedding). Numeric results are not embedded here — run the notebooks to reproduce exact metrics.

Recommended next steps:
- Curate and expand the dataset, prioritizing under-represented classes.
- Fine-tune MobileNetV2 and experiment with learning-rate schedules and regularization strategies; evaluate pruned/quantized models for deployment.
- Validate augmented data (GAN outputs) with a classifier-in-the-loop or human review before mixing into training sets.
- Explore joint training of denoiser + classifier or end-to-end fine-tuning on denoised images.
- Package the Streamlit app with a requirements.txt, and optionally provide a Dockerfile or TFLite/ONNX exports for edge deployment.

This concludes the report. Run the individual notebooks in the project directory to reproduce experiments, inspect artifacts, and obtain numerical evaluation results.