Garbage Classification Capstone Project

Contents

CONTENTS 1

1. INTRODUCTION AND PROJECT OVERVIEW 2

2. BACKGROUND AND MOTIVATION 2

3. PROBLEM STATEMENT AND OBJECTIVES 2

4. DATA COLLECTION AND PREPARATION 3

4.1. Dataset Overview 3

4.2. Preprocessing Steps 3

4.3. Dataset Splits 4

5. EXPLORATORY DATA ANALYSIS (EDA) 4

6. MODELING APPROACHES 5

6.1. ConvNeXt 6

6.2 YOLOv8 (Experimental) 6

7. HYPERPARAMETER TUNING AND VALIDATION 6

8. IMPLEMENTATION AND API ARCHITECTURE 9

8.1. Flask Application Design 9

8.2. Key API Endpoints 11

8.3. Configuration parameters 11

Model Hyperparameters (default values): 11

Location parameters (default values) 12

General parameters (default values) 12

9. DEPLOYMENT STRATEGY 12

9.1 Docker Containerization 12

9.2 AWS Cloud Deployment 13

Deployment Steps: 13

10. RESULTS AND OBSERVATIONS 13

11. DISCUSSION AND LIMITATIONS 13

12. FUTURE DIRECTIONS 14

13. CONCLUSION 14

1. Introduction and Project Overview

Waste management is a global challenge, with increasing urban populations generating larger volumes of waste that need to be sorted and processed accurately to maximize recycling and reduce landfill usage. Manual waste sorting is labor-intensive, prone to errors, and costly, leading to the development of automated solutions capable of handling diverse waste streams in real-time.

This capstone project involves constructing a **deep learning-based image classification system** to identify six waste categories: paper, plastic, metal, glass, cardboard, and trash. By employing advanced machine learning techniques – particularly **ConvNeXt** – and an accompanying **Flask API**, the project shows how computer vision models can be integrated into practical environments for efficient, scalable waste-sorting applications.

2. Background and Motivation

The **TrashNet** dataset, created by Stanford University researchers, is crucial for this project. Waste sorting errors in recycling facilities incur financial and environmental costs. Recyclable materials often end up in landfills, while non-recyclables contaminate recyclables, diminishing their value. Automated classification systems help by:

- **Increasing Sorting Accuracy**: Minimizing human error by reducing reliance on human labor.
- **Lowering Operating Costs**: Streamlining sorting processes, reducing the need for workers.
- **Improving Recycling Rates**: Correctly identifying waste categories for proper recycling or disposal.

3. Problem Statement and Objectives

The primary objective is to classify images of garbage into one of the six main categories – paper, plastic, metal, glass, cardboard, and trash – using a limited dataset of 2,527 images. The goals are:

- **Achieve Classification Accuracy ≥ 90%**: The model should accurately identify objects under different lighting conditions and angles.
- **Sustain Precision and Recall ≥ 85%**: Each category, including those with fewer images (such as trash), should have strong predictive performance.
- **Develop a Production-Ready API**: The final solution should be deployable in real-world environments, supporting both inference and re-training.

- **Enable Continuous Learning**: Implement mechanisms for on-the-fly retraining with newly labeled data.

4. Data Collection and Preparation

4.1. Dataset Overview

- **Name**: TrashNet
- **Source**: Stanford University on Kaggle
- **Total Images**: 2,527
- **Classes**:
    - Glass: 501 images
    - Paper: 594 images
    - Cardboard: 403 images
    - Plastic: 482 images
    - Metal: 410 images
    - Trash: 137 images
- **Image Format**: JPGs (3264×2448 or 4032×3024 pixels)
- **Size**: ~3.5 GB uncompressed

4.2. Preprocessing Steps

- **Resizing**: Each image was downscaled to **242×242** pixels, balancing visual detail and computational efficiency.
- **Normalization:** Apply standard transformations **(mean subtraction** and **division by standard deviation)** using **ImageNet** statistics (mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225]) to ensure consistency with **the pre-trained ConvNeXt model.**
- Data Augmentation (On-the-Fly): Included random rotations, flips, zooms, brightness and contrast shifts, and translations to effectively increase dataset size and improve model generalization.
    - Augmentations help the model learn invariance to variations in orientation, position, lighting, and scale, reducing overfitting.
    - Dynamic (On-the-Fly) augmentations minimize memory usage by avoiding the need to store multiple augmented copies, allowing new, randomized transformations each epoch for better real-world generalization.

4.3. Dataset Splits

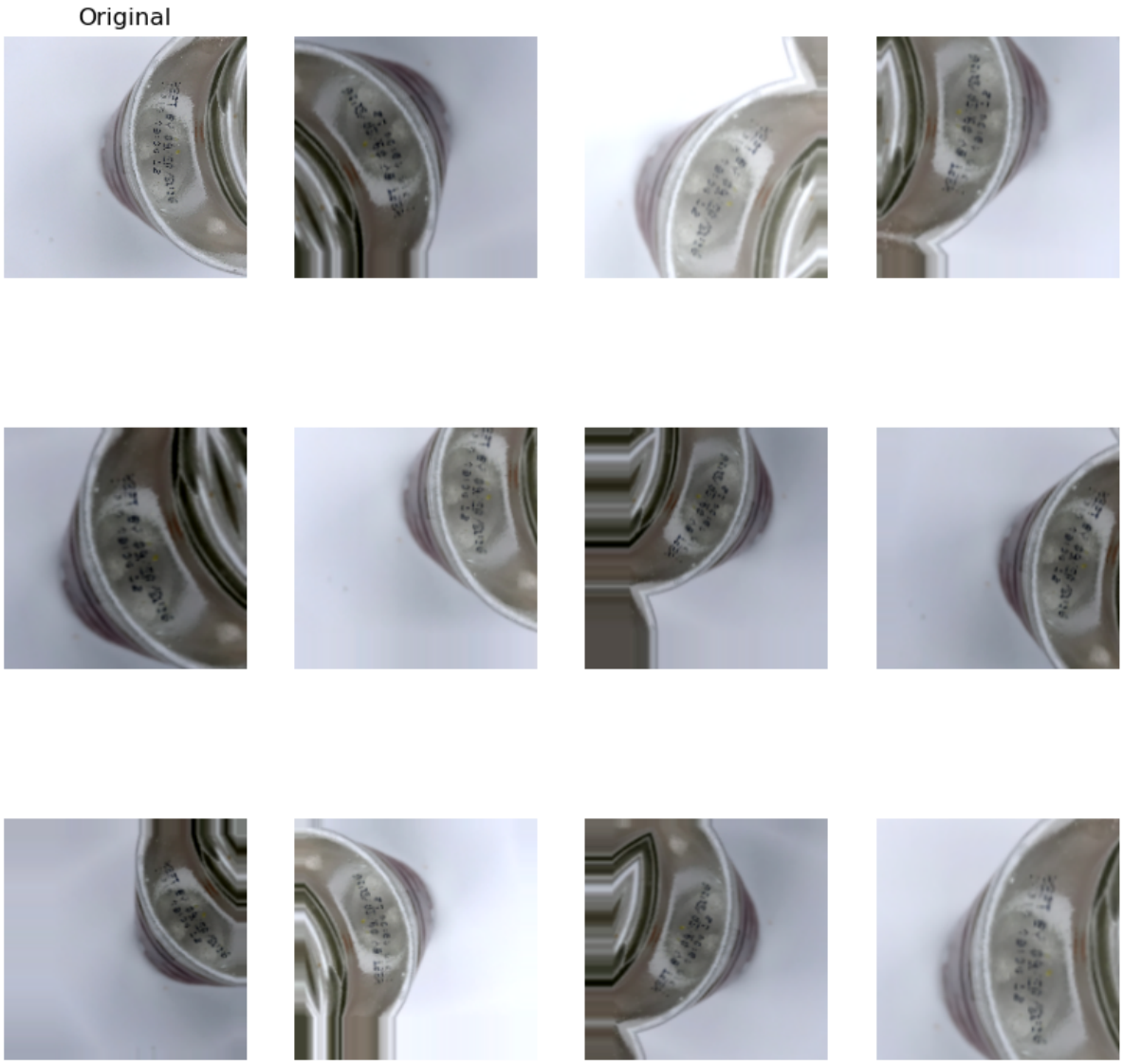
- **Training Set**: 70% of the data
- **Validation Set**: 10% of the data
- **Testing Set**: 20% of the data

This methodology ensures an appropriate partitioning of the dataset, while preserving a dedicated testing set for **unbiased performance evaluation**. Moreover, the distribution of classes across these sets varies with each training instance because **the dataset undergoes shuffling prior to splitting.** This randomization helps to mitigate potential selection bias and contributes to a more robust estimate of the model's performance.

5. Exploratory Data Analysis (EDA)

Sample Visualization: Random subsets of images from each class were examined. The sample images for each class are shown below.

The plot below shows the original image along with 11 randomly augmented versions to verify the results of the augmentation process.



Class Imbalance: The “trash” category includes only 137 images, which raises concerns about potential underfitting in that class. This imbalance underscores the importance of data augmentation and highlights the need for more advanced techniques such as class weighting or oversampling. Addressing this issue will be a key consideration for the project’s further development.

6. Modeling Approaches

For this project, two modern deep learning models were selected as the main candidates for evaluation: **ConvNeXt** and **YOLOv8**.

ConvNeXt is an updated version of traditional convolutional neural networks like ResNet. It incorporates ideas from newer models called vision transformers to enhance

its design. This model has a straightforward, organized structure and was trained on a large dataset called ImageNet, which enables it to classify images accurately while maintaining a low computational cost.

YOLOv8 originates from the "You Only Look Once" series, which is primarily designed for real-time object detection. It uses a single-stage process to quickly identify and classify objects in images. Although YOLOv8 is built for object detection, it can also be adjusted to perform image classification tasks.

Both models were chosen because they are among the best available methods in their fields, and the goal was to explore how effectively they can automate waste classification tasks in this project.

6.1. ConvNeXt

ConvNeXt was selected as the primary architecture due to its strong performance in image classification benchmarks and its ease of adaptation for various classification tasks.

- **Model Modification**: The final classification layer was replaced with a **6-class output** layer.
- **Pretrained Weights**: Pretrained weights (on ImageNet) were utilized to enhance the model's performance on a smaller dataset like TrashNet.
- **Fine-Tuning Strategy**: The classification layer of the model was unfrozen to enable adaptation to the new task while maintaining general visual features.

6.2 YOLOv8 (Experimental)

While YOLOv8 is widely recognized for its object detection capabilities, it also supports classification tasks and stands out for its ease of implementation. Out-of-the-box, YOLOv8 works impressively well and delivers faster results with minimal configuration compared to other models. However, for this project, I chose to implement the ConvNeXt model. This decision was driven by my desire to gain hands-on experience with the PyTorch framework and to delve into more complex customization and fine-tuning challenges. As a result, YOLOv8 is set aside for possible future development when its advantages can be leveraged in a different phase of the project.

The test results for the YOLOv8 implementation for garbage classification can be found in Notebooks/YOLO_testing.ipynb.

7. Hyperparameter Tuning and Validation

To systematically search for the best hyperparameters during model training, I implemented a custom grid search algorithm. This implementation is designed to handle multiple sets of hyperparameter configurations, each specified as a dictionary.

Each dictionary contains named parameters and the respective values to be tested. The algorithm processes each dictionary separately and uses a Cartesian product approach to derive all possible combinations of the parameters within that dictionary. For instance, consider the following parameter grid:

```
param_grid = [
  {
    'BATCH_SIZE': 24,
    'LR_DECOY': [0.85, 0.8, 0.75],
    'INITIAL_LR': [0.00275, 0.0025, 0.00225],
  },
  {
    'BATCH_SIZE': 24,
    'NUM_EPOCHS': [4, 8, 10],
    'LR_DECOY': 0.85,
    'INITIAL_LR': 0.00275,
  },
]
```

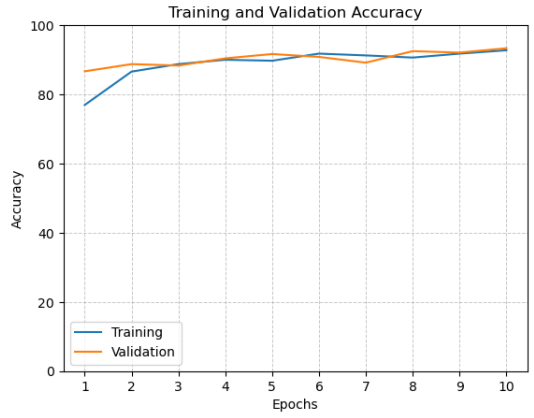
In the first dictionary, the algorithm will generate 9 combinations (3 values for 'LR_DECOY' multiplied by 3 values for 'INITIAL_LR'), while 'BATCH_SIZE' remains constant. In the second dictionary, since 'LR_DECOY' and 'INITIAL_LR' are fixed and only 'NUM_EPOCHS' varies with three different values, 3 combinations are produced. Each dictionary is processed independently, so the combinations derived from one dictionary are not mixed with those from another. This approach allows for flexible and efficient exploration of different hyperparameter configurations, which is essential for systematic fine-tuning model performance.

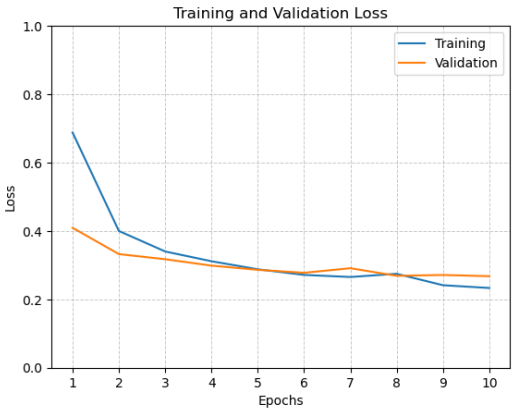
Using this **grid search** approach, I tested various configurations of:

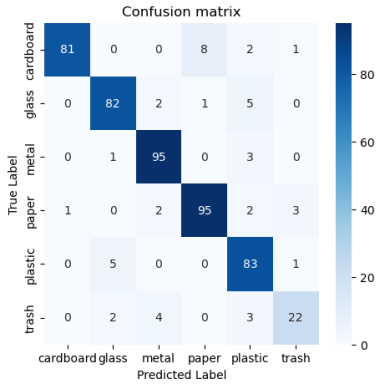
1. **Learning Rate**: 0.004, 0.003, 0.00275, 0.0025, 0.00225, 0.002, 0.001
2. **Learning Rate Decay**: 0.9, 0.85, 0.8, 0.75
3. **Batch Sizes**: 16, 24, 32, 48

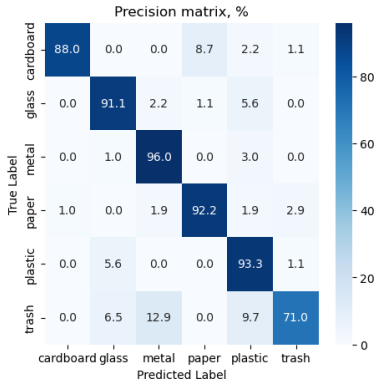
Training logs, accuracy/loss curves, and confusion matrices were inspected to identify optimal settings. The output below represents an example of output produced by the application for each model tested during one round (with one combination of parameters – Initial LR = 0.00275, LR Decoy = 0.85).

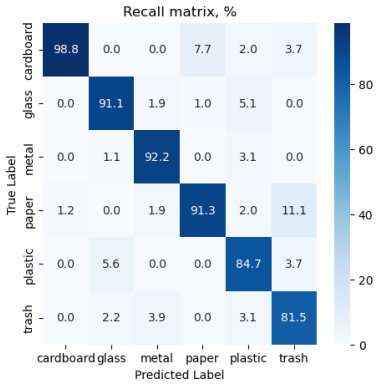
```
2025-02-18 20:36:57,172 - __main__ - DEBUG - Combination 1/1
2025-02-18 20:37:00,604 - __main__ - DEBUG - train_model() uses mps device
2025-02-18 20:37:00,607 - __main__ - DEBUG - From scratch: True
2025-02-18 20:37:00,607 - __main__ - DEBUG - Source folder: ../Data/raw
2025-02-18 20:37:00,607 - __main__ - DEBUG - New dataset folder: None
2025-02-18 20:37:00,608 - __main__ - DEBUG - Total number of elements for training - 1770, validation - 252, testing - 505
2025-02-18 20:37:00,610 - __main__ - DEBUG - Learning Rate: 0.00275, LR Decoy: 0.85, Batch size: 24
2025-02-18 20:38:37,909 - __main__ - DEBUG - Epoch [ 1/10]: Train Loss: 0.6879, Train Acc: 76.94%, Val Loss: 0.4093, Val Acc: 86.67%, Time:  97 sec
2025-02-18 20:40:09,137 - __main__ - DEBUG - Epoch [ 2/10]: Train Loss: 0.4000, Train Acc: 86.59%, Val Loss: 0.3323, Val Acc: 88.75%, Time:  91 sec
2025-02-18 20:41:40,635 - __main__ - DEBUG - Epoch [ 3/10]: Train Loss: 0.3400, Train Acc: 88.76%, Val Loss: 0.3174, Val Acc: 88.33%, Time:  91 sec
2025-02-18 20:43:13,051 - __main__ - DEBUG - Epoch [ 4/10]: Train Loss: 0.3115, Train Acc: 90.01%, Val Loss: 0.2986, Val Acc: 90.42%, Time:  92 sec
2025-02-18 20:44:45,077 - __main__ - DEBUG - Epoch [ 5/10]: Train Loss: 0.2879, Train Acc: 89.73%, Val Loss: 0.2863, Val Acc: 91.67%, Time:  92 sec
2025-02-18 20:46:16,335 - __main__ - DEBUG - Epoch [ 6/10]: Train Loss: 0.2715, Train Acc: 91.78%, Val Loss: 0.2777, Val Acc: 90.83%, Time:  91 sec
2025-02-18 20:47:49,422 - __main__ - DEBUG - Epoch [ 7/10]: Train Loss: 0.2654, Train Acc: 91.27%, Val Loss: 0.2911, Val Acc: 89.17%, Time:  93 sec
2025-02-18 20:49:19,718 - __main__ - DEBUG - Epoch [ 8/10]: Train Loss: 0.2747, Train Acc: 90.64%, Val Loss: 0.2688, Val Acc: 92.50%, Time:  90 sec
2025-02-18 20:50:49,589 - __main__ - DEBUG - Epoch [ 9/10]: Train Loss: 0.2413, Train Acc: 91.78%, Val Loss: 0.2713, Val Acc: 92.08%, Time:  90 sec
2025-02-18 20:52:21,597 - __main__ - DEBUG - Epoch [10/10]: Train Loss: 0.2332, Train Acc: 92.75%, Val Loss: 0.2677, Val Acc: 93.33%, Time:  92 sec
2025-02-18 20:52:50,203 - __main__ - DEBUG - Trial: 1, Test Loss: 0.2582, Test Acc: 90.87%, Avg Time/Image: 0.011 sec
2025-02-18 20:52:50,221 - __main__ - INFO - Final Test: Loss: 0.2582, Acc: 90.87%
2025-02-18 20:52:50,230 - __main__ - INFO - Mean processing time per image: 0.011 sec
2025-02-18 20:53:50,272 - __main__ - DEBUG - Total cycle elapsed time: 16 min 53 sec
```











The table below summarizes the results for the best-performing models and their parameters.

#	Initial Learning rate	Learning Rate decoy	Batch size	Test loss	Test accuracy	Worst precision, %(class)	Worst recall, %(class)
1	0.00400	0.85	24	0.2494	90.67%	71.0% (trash)	81.5% (trash)
2	0.00300	0.85	24	0.2444	90.87%	71.0% (trash)	81.5% (trash)
3	0.00275	0.85	24	0.2586	90.87%	71.0% (trash)	81.5% (trash)
4	0.00250	0.85	24	0.2887	91.87%	67.7% (trash)	84.0% (trash)
5	0.00225	0.85	24	0.2740	90.87%	83.9% (trash)	76.5% (trash)
6	0.00200	0.85	24	0.2978	91.47%	64.5% (trash)	87.0% (trash)
7	0.00100	0.85	24	0.2977	91.27%	74.2% (trash)	79.3% (trash)
8	0.00400	0.80	24	0.2882	91.87%	90.3% (trash)	77.8% (trash)
9	0.00300	0.80	24	0.2838	91.07%	80.6% (trash)	75.8% (trash)

10	0.00275	0.80	24	0.2730	91.07%	77.4% (trash)	80.0% (trash)
11	0.00250	0.80	24	0.2782	91.27%	80.6% (trash)	75.8% (trash)
12	0.00225	0.80	24	0.2789	90.87%	77.4% (trash)	75.0% (trash)
13	0.00200	0.80	24	0.2666	91.27%	77.4% (trash)	75.0% (trash)
14	0.00100	0.80	24	0.3128	90.48%	71.0% (trash)	73.3% (trash)
15	0.00400	0.75	24	0.2798	91.47%	83.9% (trash)	78.8% (trash)
16	0.00300	0.75	24	0.2787	91.07%	74.2% (trash)	76.7% (trash)
17	0.00275	0.75	24	0.2865	92.06%	71.0% (trash)	84.6% (trash)
18	0.00250	0.75	24	0.2646	91.27%	77.4% (trash)	77.4% (trash)
19	0.00225	0.75	24	0.2676	91.47%	77.4% (trash)	75.0% (trash)
20	0.00200	0.75	24	0.2734	91.47%	77.4% (trash)	75.0% (trash)
21	0.00100	0.75	24	0.3278	90.48%	67.6% (trash)	72.4% (trash)

All trials surpassed the **90% accuracy** target. The best-performing trial (Trial #17) achieved **a 92.06% test accuracy** with a test loss of approximately 0.2865, using an initial learning rate of 0.00275, an LR decay of 0.75, and a batch size of 24. However, in that trial, neither precision nor recall reached the 85% threshold. **Precision exceeded 85% in Trial #8** (initial LR of 0.004, LR decay of 0.80), **while recall surpassed 85% in Trial #6** (initial LR of 0.002, LR decay of 0.85). Depending on which type of error is more critical for the business, one of these trials may be preferable.

The worst-performing class in terms of both precision and recall was “trash,” which is also the most imbalanced class in our dataset. This highlights the need to address class imbalance in future work, either by obtaining more images of this class and/or by using oversampling or other techniques.

Additionally, caution is warranted when interpreting these results. Due to high computational costs, cross-validation was not performed during the trials. Consequently, some outcomes may have occurred by chance, although they did not show a significant advantage over other trials. Implementing cross-validation in future work should help mitigate this variability.

8. Implementation and API Architecture

8.1. Flask Application Design

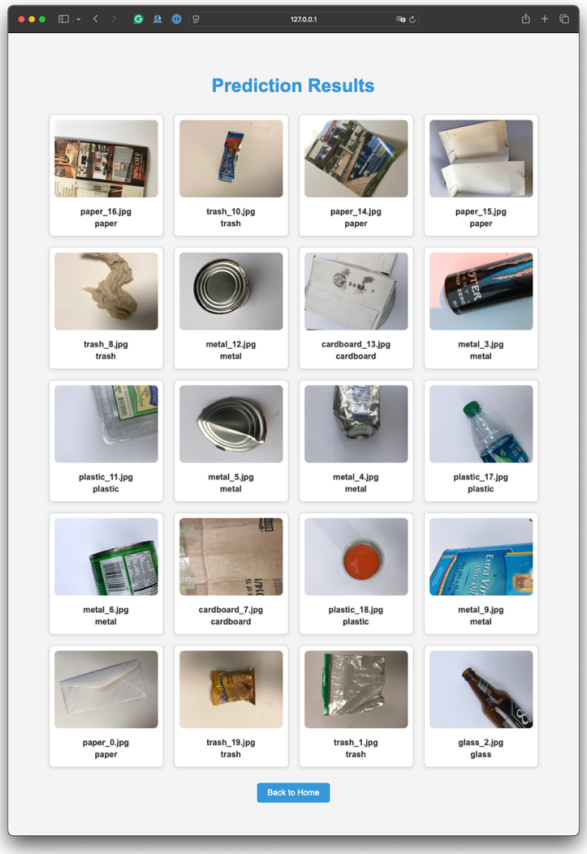
A **Flask** web application was built to facilitate easy interaction with the classification model. The architecture includes the following components:

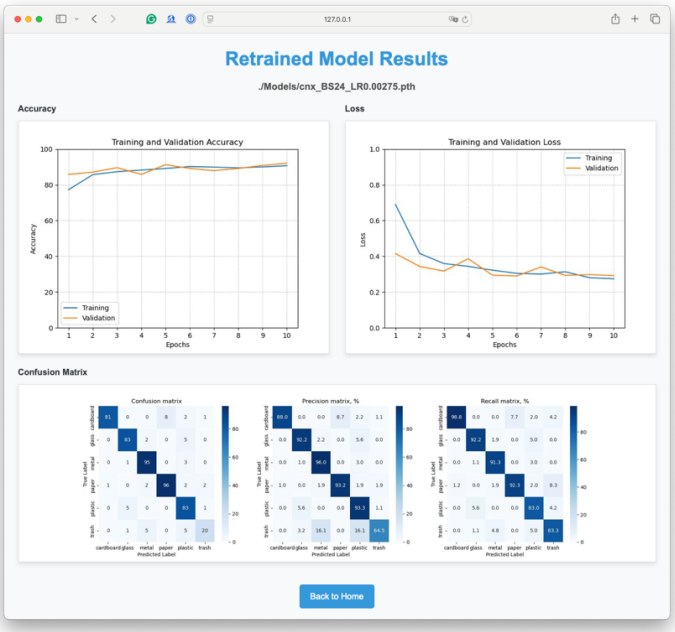
1. **Model Loader**: Loads the fine-tuned ConvNeXt model into memory.
2. **Prediction Pipeline**: Accepts images from a specified folder, applies the same transformations used during training, and outputs predicted labels.
3. **Configuration Module**: Manages hyperparameters and model settings, enabling dynamic changes without requiring restarts.

4. **Retraining Module**: Handles new data ingestion and either retrains or fine-tunes the model.

The application interacts via HTTP requests.

For example, the screenshots below display the model's predictions (accessible via the API) as well as the outcomes from model retraining.





8.2. Key API Endpoints

- **/**: Display the API documentation.
- **/change**: Modify configuration parameters (e.g., learning rate, batch size).
- **/save_config**: Save the current configuration to app_config.json file.
- **/restore_config**: Load a config from the specified .json file or from the default app_config.json file.
- **/info**: Retrieve current parameter settings.
- **/predict**: Run batch or single-image prediction from the specified folder or from the default testing folder.
- **/refresh**: Pull more images from the dataset into a testing folder.
- **/retrain**: Retrain the model using new data, either from scratch or by fine-tuning the existing model.

Endpoints can respond in **HTML** (for browsers) or **JSON** (for programmatic clients). This flexibility supports various deployment scenarios.

8.3. Configuration parameters

The application includes several configuration parameters that can be logically divided into three groups:

- **Model Hyperparameters:** Settings that control the training and performance of the model.
- **Location Parameters:** Paths specifying where data, models, and other resources are stored.
- **General Parameters:** Settings that affect overall application behavior and runtime features.

Model Hyperparameters (default values):

- **BATCH_SIZE (24)**: Number of samples processed before the model's weights are updated during training.
- **INITIAL_LR (0.00275)**: The starting learning rate for the optimizer, which controls how much the model weights are adjusted during training.
- **LR_DECOY (0.85)**: The factor by which the learning rate is decayed over time to gradually reduce the step size during optimization.
- **NUM_EPOCHS (10)**: Total number of complete passes through the training dataset.
- **IMAGE_SIZE (242)**: The dimension (in pixels) to which images are resized; here, images are resized to 242×242 pixels.
- **NUMBER_OF_TRIALS (1)**: Number of independent training trials to run; useful for assessing variability in elapsed time.

- **NORMALIZE (true)**: Boolean flag indicating that image normalization is applied, using the standard ImageNet mean and standard deviation.

Location parameters (default values)

- **DATA_DIR ('../Data/raw')**: Directory path where the raw dataset is stored.
- **CNX_LOAD ('../Models/cnx_BS24_LR0.00275.pth')**: File path to load a pre-trained ConvNeXt model checkpoint.
- **CNX_SAVE ('../Models/cnx_')**: Base file path for saving trained ConvNeXt model checkpoints.
- **IMAGES_TO_PREDICT ('../Data/processed/random_samples')**: Directory containing processed sample images for running model predictions.

General parameters (default values)

- **NUM_CLASSES (6)**: The number of distinct classes the model is trained to classify.
- **TEST_SPLIT_RATIO (0.2)**: Proportion of the dataset allocated for testing the model's performance.
- **VALID_SPLIT_RATIO (0.1)**: Proportion of the dataset reserved for validating the model during training.
- **DEBUG (true)**: Boolean flag that enables debug mode, providing detailed logging and error information for troubleshooting.
- **NUM_WORKER (4)**: Number of parallel worker processes used for loading data during training.
- **PREFETCH_FACTOR (2)**: Number of batches that each worker preloads in advance to speed up data processing.

The user can change the parameters through API and save them to be used if the system restarts.

9. Deployment Strategy

9.1 Docker Containerization

A **Docker** image encapsulates the entire environment:

• **Base Image**: Python + CUDA (for GPU support, if applicable)

• **Dependencies**: PyTorch, Flask, and other required libraries

• **Application Code**: app.py, convnext.py, random_samples.py, myutils.py, plus configuration files

Model weights are not packaged within the container; instead, they are loaded separately from an AWS S3 bucket so that multiple instances can share the same model.

The container image is hosted on **Docker Hub**, simplifying distribution. Future improvements might include shifting this image to a private repository on AWS Elastic Container Registry (ECR).

9.2 AWS Cloud Deployment

- **EC2 G5.x2large** Instance: Provides **8 vCPUs**, **1 GPU**, **32 GB** memory, and **70 GB** of storage.
- **AWS S3**: Stores archived datasets and model weights to speed up the installation/initialisation process and save costs.
- **Shell Scripts**: Automate environment setup (installing dataset and model weights, pulling Docker images, setting environment variables).

Deployment Steps:

1. Spin up EC2 instance.
2. Pull Docker image from Docker Hub.
3. Load dataset/model from S3.
4. Run a container with a specified port, shared memory, availability of GPU and volume mappings for API access.

10. Results and Observations

The training of the model provides the following results:

1. **Accuracy**: The model consistently achieves **~92.06%** on the test set under optimal hyperparameters.
2. **Precision & Recall**: Since no single trial achieved both precision and recall above 85%, a trade-off is required. **In Trial #8** (initial LR of 0.004, LR decay of 0.80), **precision exceeded 85%**, while in **Trial #6** (initial LR of 0.002, LR decay of 0.85), **recall surpassed 85%.** Depending on which type of error is more critical for the business, one of these trials may be preferable.
3. **Training Stability**: Training remained stable at the chosen LR with no significant overfitting observed once augmentation was applied.

11. Discussion and Limitations

1. **Data Imbalance**: The project overcame some imbalance with on-the-fly augmentation, but performance on minority classes can still improve.

2. **Computational Constraints**: Limited GPU memory restricted the ability to conduct extensive experiments (e.g., large batch sizes, multiple cross-validations).
3. **Single Validation Split**: Relying on one validation set rather than multiple folds means hyperparameter selection might not be optimally robust.
4. **Real-World Generalizability**: The dataset images were taken under controlled conditions (white background, stable lighting). In practice, real garbage streams might differ significantly in backgrounds, lighting, or object occlusions.

12. Future Directions

1. **Comprehensive Cross-Validation**: Implement K-fold cross-validation to better assess model stability and hyperparameter robustness.
2. **Misclassification:** In future work, it will be essential to rigorously investigate the images that were misclassified by the current model to identify underlying patterns and refine its performance.
3. **Active Learning**: Incorporate an active learning workflow that highlights uncertain predictions for manual labeling, thus continually refining the model.
4. **YOLOv8 Expansion**: Rigorously test YOLOv8 under the same conditions to compare results with ConvNeXt directly.
5. **Edge Deployment**: Explore lightweight versions of the model (e.g., ConvNeXt-Tiny or YOLOv8 variants) for real-time inference on embedded devices.
6. **Class Balancing Methods**: Investigate oversampling or other methods (e.g., SMOTE for images) to handle underrepresented classes more effectively.

13. Conclusion

In summary, this capstone project demonstrates the feasibility and practical value of an automated garbage classification system. By leveraging an advanced **ConvNeXt** model, it achieves **over 91% accuracy** on the TrashNet dataset while providing robust support for **re-training**, **dynamic hyperparameter tuning**, and **scalable deployment**. The solution holds clear potential for reducing manual labor costs and enhancing recycling rates in waste management facilities. Moving forward, refinements in data collection, model performance, and large-scale deployments will determine the ultimate impact of this technology on sustainability efforts worldwide.