Automated Garbage Classification Using ConvNeXt

A Machine Learning Approach to Improving Waste Sorting Efficiency

Alexey Kholodov

20 February 2025

http://www.linkedin.com/in/a-kholodov

Waste Management Challenge & Project Overview

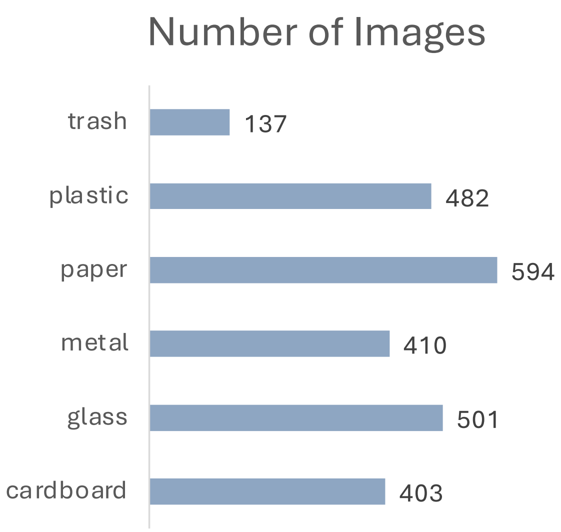
Global Challenge: Manual sorting of waste is often fraught with errors, requires significant labor, and incurs high costs, resulting in inefficiencies in recycling and greater reliance on landfills.

Project Goal: Create a deep learning model capable of categorizing images of waste into six distinct types – paper , plastic, metal, glass, cardboard, and general trash – to enhance sorting precision and minimize labor expenses.

Success Metrics:

- Achieve a classification accuracy of at least **90%.**
- Ensure precision and recall of at least **85%** for each waste category.
- Develop a system that can be deployed for real-time sorting with retraining capabilities.

Scope & Constraints



Scope:

Focus on classifying six categories (paper, plastic, metal, glass, cardboard, trash).

Aim for a solution that can be trained continuously with new data.

Constraints:

Data Limitations: The TrashNet dataset has only 2,527 images, leading to potential class imbalance (especially “trash” with only 137 images).

Computational Resources: Limited GPU/CPU capacity for large-scale deep learning.

The Data Science Problem & Dataset Overview

Data Science Problem Statement: Image classification for garbage types to increase sorting accuracy and reduce manual labor costs.

Dataset:

Name & Source: TrashNet dataset from Stanford, hosted on Kaggle.

Size & Composition: 2,527 images of six classes:

Paper (594), Glass (501), Cardboard (403), Plastic (482), Metal (410), Trash (137)

Data Preporces-sing Steps

Preprocessing:

- Resized all images to 242×242; normalized with ImageNet statistics.
- Handled a limited dataset size by using on-the-fly data augmentation – such as rotations, flips, zoom, translations, and brightness/contrast adjustments – to expand the training dataset.

Dataset Splits (randomly shuffled):

70% Train, 10% Validation, 20% Test.

Key Takeaway: Data augmentation is crucial to address imbalance and enhance model generalization, **and by performing it on-the-fly, it reduces memory usage while continuously introducing novel transformations in each training epoch.**

Approach & Methodology

Models Considered:

- **ConvNeXt:** Primary choice for image classification; pretrained on ImageNet for strong baseline.
- **YOLOv8 (Experimental):** Primarily an object detection model but can be adapted to classification.

Reasoning for ConvNeXt:

Solid performance, easier adaptation for custom tasks, robust with fewer data points.

Hyperparameter Tuning:

Custom grid search used to systematically explore learning rates, batch sizes, etc.

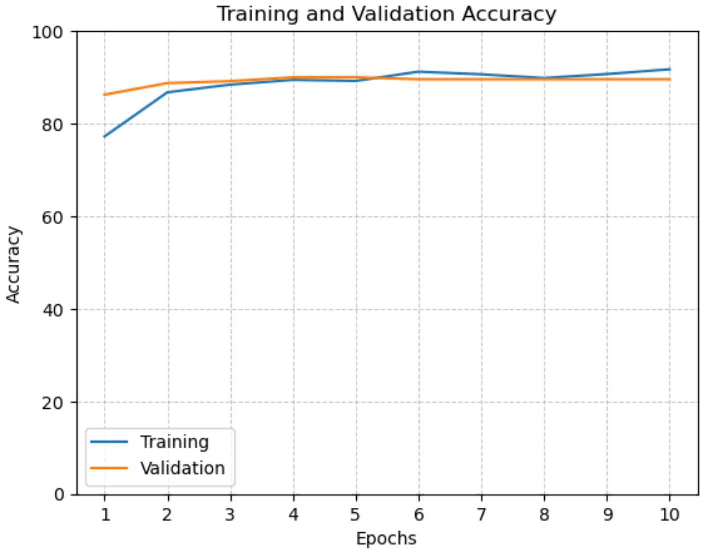
Hyper-parameter Tuning
(the best results)

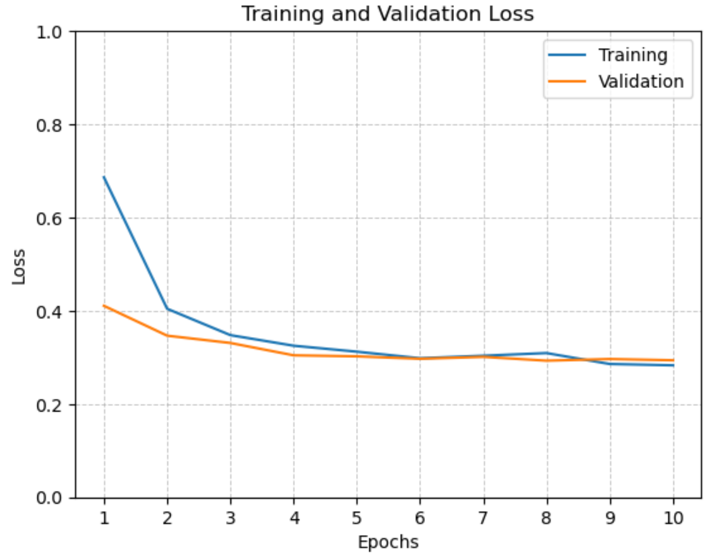
Parameters:

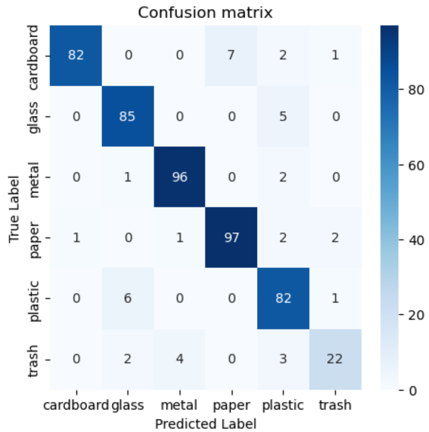
Initial Learning Rate:	0.00275
Learning Rate Decoy:	0.75
Batch size:	24

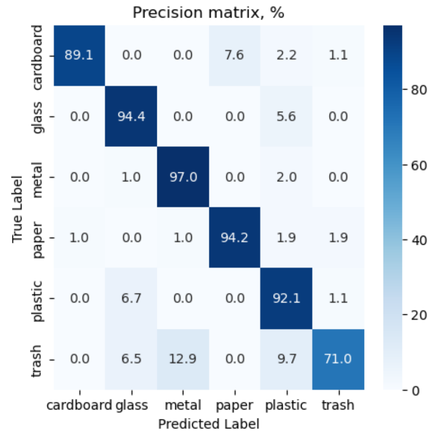
Results:

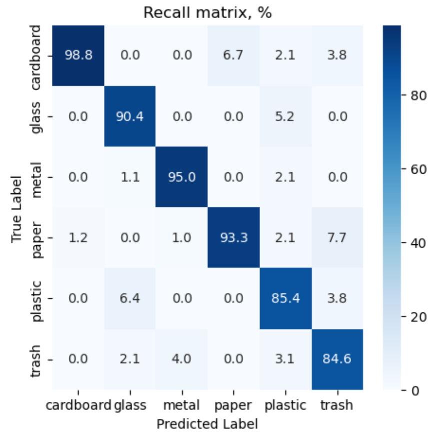
Loss:	0.2865
Accuracy:	92.06%
Worst Precision:	71.0%
Worst Recall:	84.6%











Hyper-parameter Tuning

(other combinations)

Although the target of 85% recall and precision was not achieved, some variants still meet real business requirements.

#	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)**

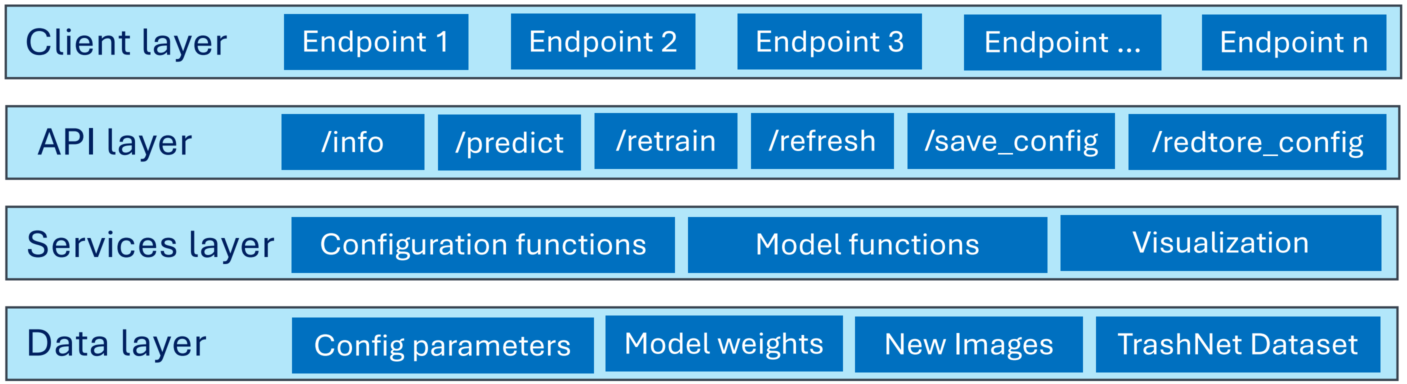
Solution Design & Production Deployment Readiness

Design Principles:

- **Flask Application** for easy model serving (prediction & retraining modes).
- **Cloud-Ready** with Docker containerization and AWS deployment strategies.

Rich API & Configurations:

- Modifiable hyperparameters (learning rate, batch size) via HTTP requests.
- Three retraining options: from scratch, fine-tune with new data, or retraining from scratch adding new data.



Deployment & Scalability

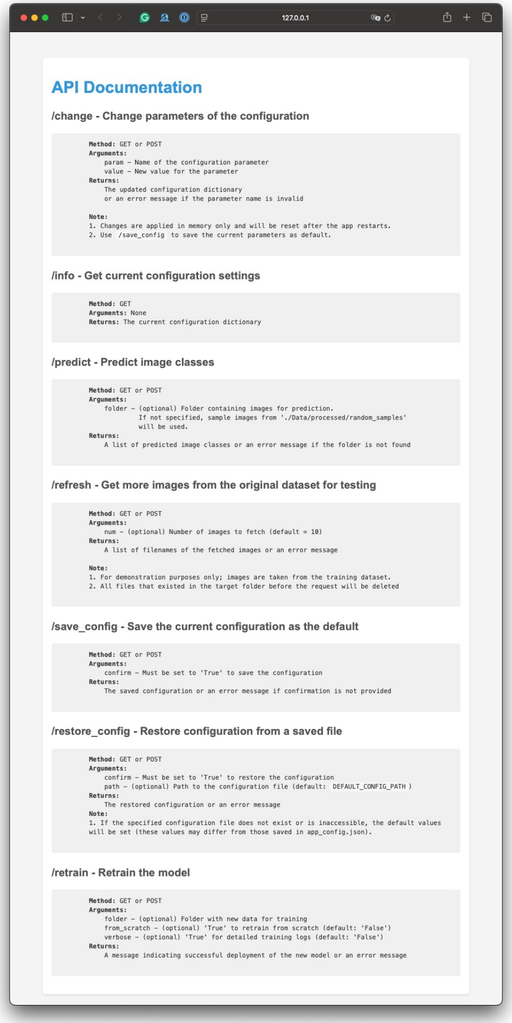
AWS Deployment Steps:

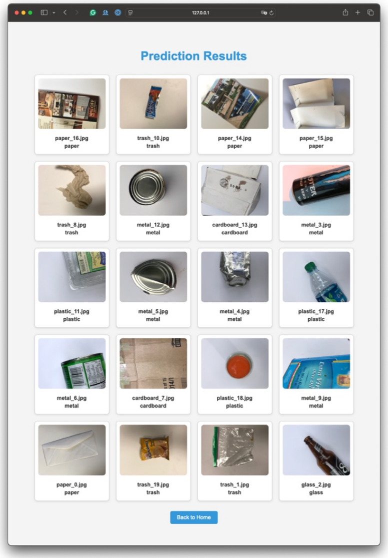
1. Containerize application modules using Docker and push image to Docker Hub.
2. Archive the dataset and the models and upload to AWS S3.
3. Launch AWS EC2 (g5.2xlarge with AWS Linux was used).
4. Install data and pre-trained model weights to EC2 volume.
5. Pull the image and run it on EC2 instance.

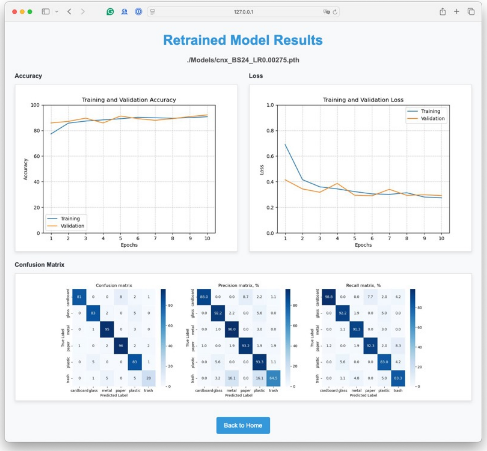
Scalability

- Multiple EC2 instances can be deployed behind a load balancer to handle increased traffic.
- A separate instance can be used for model retraining in one of three possible modes. The updated weights can then be applied to newly launched production nodes, replacing existing nodes without disruption.

Various Deployment Scenarios – *API Endpoints can respond in HTML …*

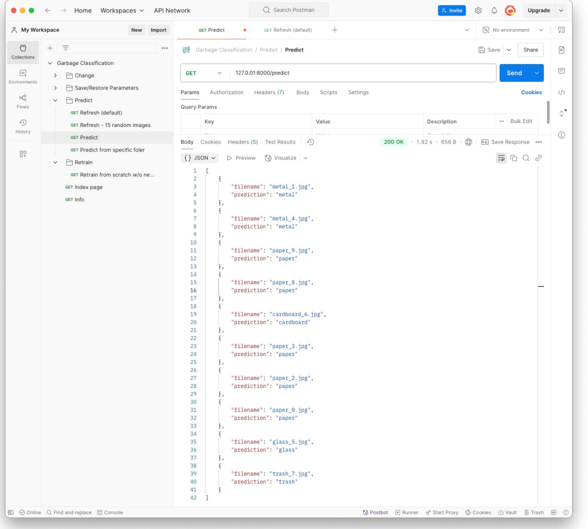


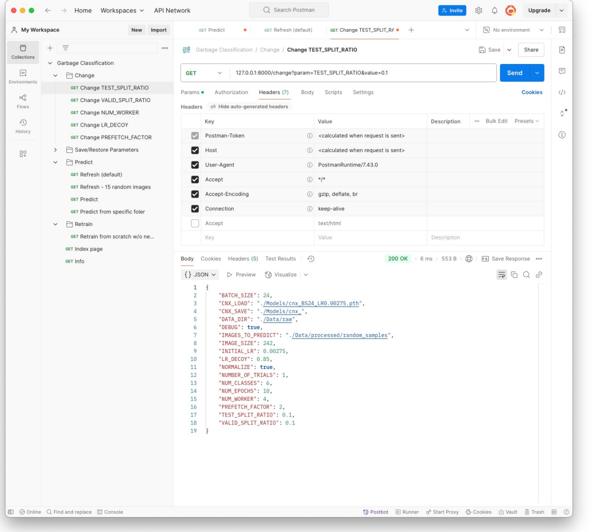




Various Deployment Scenarios –

… *or JSON*





Roadmap & Practical Considera-tions

1. **Comprehensive Cross-Validation:** Use K-fold cross-validation to strengthen model stability and hyperparameter robustness.
2. **Misclassification Analysis:** Investigate misclassified images to uncover hidden patterns and refine performance.
3. **Active Learning:** Highlight uncertain predictions for manual labeling, continually enhancing the model.
4. **YOLOv8 Evaluation:** Test YOLOv8 under identical conditions to compare results directly with ConvNeXt.
5. **Edge Deployment:** Evaluate lightweight variants (e.g., ConvNeXt-Tiny, YOLOv8) for real-time inference on embedded devices.
6. **Class Balancing:** Explore oversampling or SMOTE-like approaches to better handle underrepresented classes.

Thank you!