جامعة مصر للمعلوماتية
EGYPT UNIVERSITY OF INFORMATICS

FACULTY OF ENGINEERING

CSE-351

Waste Image Classification Using CNN

PROJECT REPORT

Rokaya Ramy	22-101171
Omar El Nabarawy	22-101123
Jana Sherif	22-101234

Project Description

Introduction

Recycling is an important part of reducing waste and protecting the environment, but sorting waste correctly is still a big challenge. Many recyclable items end up in landfills because they are not properly separated. This project aims to solve that problem by using the ResNet-18 model and a custom CNN model to automatically classify waste images into categories like plastic, paper, metal, and glass. By doing this, we hope to make recycling faster, more accurate, and easier to manage on a larger scale.

Dataset Overview

For this project, we initially worked with a publicly available waste classification dataset containing approximately 10,000 labeled images across 20 classes. Each image depicted a single item of waste, ranging from different types of cans and bottles to various plastic containers, bags, and packaging materials. The dataset included images taken under diverse lighting conditions and backgrounds, which made it suitable for training models that need to generalize to real-world environments.

After training our initial model on the full 20-class dataset, we evaluated its performance using standard metrics and a confusion matrix. The results revealed significant overlap and confusion between certain visually similar categories—such as multiple subtypes of plastic containers or different kinds of glass bottles. This class redundancy negatively impacted the model's ability to distinguish between them, limiting its test accuracy to around 62%.

To address this, we refined the dataset by merging or removing overlapping and redundant classes, reducing the total number of categories from 30 to 20. This simplification helped clarify decision boundaries for the model and improved class balance. All images were resized to 224×224 pixels to ensure consistency in input dimensions, and the dataset was divided into 60% training, 20% validation, and 20% testing splits. We also applied data augmentation techniques—such as rotation, flipping, and brightness adjustment—to the training set to improve robustness and prevent overfitting.

Following this revision, the model's test accuracy significantly improved to 85%, confirming that reducing visual ambiguity in the class labels led to better generalization and more reliable performance.

Data Preprocessing and Transformations

1. Dataset Loading and Splitting

To efficiently manage and utilize the dataset, a custom dataset class named LoadDataset was implemented. This class systematically traverses the directory structure, which consists of multiple classes, each containing two subfolders: default and real_world. For every class, the images within these subfolders are first listed and then shuffled randomly to avoid any bias from the original ordering. This randomization is crucial to ensure the model does not learn any unintended patterns related to image order.
The dataset is then divided into three subsets based on predefined proportions:

- Training set: Consists of the first 60% of the shuffled images.
- Validation set: Contains the subsequent 20% of the images (i.e., from 60% to 80%).
- Test set: Includes the final 20% of the images (from 80% to 100%).

This split strategy maintains class balance and diversity in each subset by drawing from both subfolders (default and real_world). Assigning numeric labels to classes based on folder names enables supervised learning by mapping each image to its correct category. The class also supports image transformations to facilitate data augmentation during training.

2. Dataset Path and Training Hyperparameters

The dataset is stored locally on the user's machine under a specified directory path. Defining this path is important for seamless data loading and accessibility during training.
Several key hyperparameters for the training process are also established at this stage:

- Batch size (32): Determines how many images the model processes simultaneously during each training iteration. Using batches speeds up training and stabilizes gradient updates.
- Number of epochs (25): Defines how many complete passes the training algorithm makes over the entire training dataset. More epochs can lead to better learning but also risk overfitting if too high.
- Learning rate (0.001): Controls the step size during optimization when updating model parameters. A suitable learning rate ensures steady and effective convergence.

Setting these hyperparameters upfront enables controlled and reproducible model training.

Data Preprocessing and Transformations

3. Data Augmentation and DataLoader Preparation

To enhance the model's ability to generalize beyond the training data, several image augmentation techniques are applied to the training images. These include:

- Resizing: All images are resized to a uniform dimension of 224x224 pixels to maintain consistency and meet model input requirements. (Required)
- Random Horizontal Flip: Images are randomly flipped horizontally, effectively doubling the variability and helping the model learn orientation-invariant features. (Augmentation)
- Random Rotation: Small rotations (up to ±10 degrees) introduce additional diversity in perspective. (Augmentation)
- Colour Jittering: Slight random changes in brightness, contrast, and saturation simulate different lighting conditions. (Augmentation)

For the validation and test sets, only resizing and normalization are applied to ensure evaluation reflects real-world performance without augmentation-induced variability. Finally, DataLoader objects are created for each dataset split. These facilitate efficient batch loading, shuffling (only for training data), and parallel processing, which significantly speeds up training and evaluation. The use of separate transforms for training and validation/test datasets ensures that data augmentation is applied only during training, preserving the integrity of validation and test results.

Model Training and Validation

Custom CNN Model

Model Architecture:

- For feature extraction:
  - 5 convolutional layers with increasing filters (32 → 512)
  - ReLU activation functions
  - MaxPooling with Kernel Size = 2 and Stride = 2
  - Adaptive Average Pooling
- For Classification:
  - Fully-Connected layer 1 (fc1)
  - Dropout with p = 0.25
  - Fully-Connected layer 2 (fc2)
- Additional features:
  - Weight initialization:
    - Kaiming initialization for convolutional layers (good for ReLU)
    - Xavier initialization for linear layers (helps with gradient stability)

Custom CNN Structure

This class defines a custom Convolutional Neural Network (CNN) model for waste image classification, implemented as a subclass of nn.Module, PyTorch's base class for all neural network models.

- **Constructor:** The constructor initializes all the layers of the CNN model. It takes a single parameter:
  - **waste_classes_nb**: Number of output classes corresponding to different waste categories.
  - The **super().__init__()** call ensures that the base class nn.Module is properly initialized.

Model Training and Validation

- **Model Architecture:** The model consists of the following components:
  a. **Convolutional Layers:**
     - Five 2D convolutional layers (nn.Conv2d) used to extract spatial features from the input images.
     - Kernel size: 3×3
     - Stride: 1
     - Padding: 1
     - These settings preserve the spatial resolution (height and width) of the input feature maps after each convolution.
  b. **Activation Function:**
     - **relu:** A nn.ReLU activation function applied after each convolution to introduce non-linearity and allow the model to learn complex patterns.
  c. **Pooling Layer:**
     - **maxpooling:** A nn.MaxPool2d layer used to downsample the feature maps. It reduces both spatial dimensions (height and width) by a factor of 2, helping to reduce computation and control overfitting.
     - **Adaptive Average Pooling:** Used to produce a fixed-size output (1×1) regardless of the input size. It also helps decouple the classifier from the input feature map size and makes the model more flexible.
  d. **Fully Connected Layers:**
     - **fc1:** A linear (nn.Linear) layer that takes 512 input features (output of adaptive pooling) and outputs 512 features.
     - **fc2:** A second linear layer that takes the 512 features from fc1 and outputs logits for each of the waste classes.
  e. **Prediction:**
     - The model outputs raw scores (logits) from fc2. During inference, the predicted class corresponds to the index of the highest value in the output logits vector (via torch.argmax).

Model Training and Validation

- **Forward:** This function defines how input data flows through the model.
  - Input tensor mod has shape [batch_size, channels, height, width].
  - The input passes through 5 blocks, each block applies in order:
    - Convolution → ReLU activation → MaxPooling
    - This progressively extracts features and reduces spatial dimensions by half at each maxpooling.
  - After the last maxpooling, the tensor shape is [batch_size, 512, 7, 7].
  - Adaptive Average Pooling (AdaptiveAvgPool2d((1,1))) reduces each feature map from size 7×7 to 1×1, producing a tensor of shape [batch_size, 512, 1, 1].
  - The tensor is then flattened into [batch_size, 512], converting the spatial maps into a feature vector.
  - The vector passes through:
    - A fully connected layer (fc1) with ReLU activation
    - A dropout layer (with dropout probability 0.25) to reduce overfitting
    - A final fully connected layer (fc2) producing the output logits.
  - The output shape is [batch_size, waste_classes_nb], representing raw class scores for each sample.

Model Training and Validation

Hyperparameters

- **batch_size = 32:** During each training epoch, the model processes the training dataset in batches of 32 samples at a time.
- **num_epochs = 25:** The model will train for 25 complete passes over the entire training dataset.
- **learning_rate = 0.001:** Controls how much the model's weights are updated during training. A learning rate of 0.001 means the weights are adjusted gradually to minimize the loss.

Criterion

nn.CrossEntropyLoss(): commonly used for multi-class classification tasks. It measures the dissimilarity between the predicted class probabilities and the true class labels, providing a measure of how well the model is performing.

Optimizer

optim.Adam(model.parameters(), lr=learning_rate): responsible for updating the model's parameters during the training process based on the computed gradients and the specified learning rate.

Model Training and Validation

ResNet18

We use a pretrained ResNet18, a deep residual network known for strong image classification performance.

- The final fully connected layer of ResNet18 is replaced to output Waste_Classes_NB logits, matching the number of classes in your dataset.
- **Pretrained Weights:** Leverages weights trained on ImageNet, helping faster convergence and better generalization.
- **Learning Rate Scheduler:** ReduceLROnPlateau monitors validation loss and reduces learning rate by half if no improvement after 2 epochs, improving fine-tuning.

Model Training and Validation

Training and Validation

Runs based on the num_epochs. Each epoch training loop runs batch by batch, validation loop runs batch by batch, early stopping condition is checked, and average training and validation losses are displayed.

- **best_val_loss = float("inf"):** variable that stores the best validation loss value during training, initialized to infinity, and used for early stopping condition.
- **patience, counter = 5, 0:** patience is used to terminate the training if validation loss did not decrease for #patience number of epochs.
- train_losses = [] : stores the average training losses for each epoch.
- val_losses = [] : stores the average validation losses for each epoch.

Training Phase

- Set model to training mode.
- Initialize total training loss to 0.
- For each batch in the training dataloader:
  - Move batch (images, labels) to GPU.
  - Forward pass: model predicts outputs.
  - Compute the training loss using the loss function.
  - Backward pass: compute gradients.
  - Optimizer step: update model weights.
  - Accumulate total training loss (scaled by batch size).
- Compute average training loss for the epoch and save it.

Validation Phase

- Set model to evaluation mode (disables dropout, etc.).
- No gradient computation (torch.no_grad()).
- For each batch in the validation dataloader:
  a. Move batch to GPU.
  b. Forward pass only.
  c. Compute validation loss and accumulate.
- Compute average validation loss for the epoch and save it.

Model Training and Validation

Early Stopping Check

- If current val_loss is lower than best_val_loss:
  - Update best_val_loss.
  - Reset counter to 0.
- Else:
  - Increment counter.
  - If counter ⩾ patience, stop training early.

Display Progress

At the end of each epoch, print training and validation losses.

Model Testing, Evaluation And Visualization

Testing

After training, we evaluated our model on a held-out test set to measure its generalization performance. The test set contained images not seen during training or validation, allowing us to assess how well the model performs on unseen data.
The testing set was predefined as 20% of the project's dataset.

Evaluation

The model achieved an overall test accuracy of [insert accuracy] and a macro-averaged F1 score of [insert F1 score]. We computed a full classification report that included precision, recall, and F1 score for each of the 20 classes. A confusion matrix was also generated to visualize class-wise prediction performance and highlight common misclassifications.
In addition, we tracked training and validation loss over each epoch to monitor learning progress and detect signs of overfitting.

Visualization

To better understand how the model interprets input data, we visualized the classification process at two levels:

- Image Processing Stages: We selected a random image and displayed it at three key points: the original input, the resized version, and the final normalized tensor used by the model. This helped clarify the preprocessing steps the data undergoes during training and inference.
- Prediction Grid: We created a 3×3 grid showing random images from the test set along with their true and predicted labels. Each title was color-coded to indicate whether the prediction was correct (green) or incorrect (red). This visualization helped qualitatively evaluate model reliability and identify patterns in errors, such as confusion between visually similar classes.

These visualizations provide both technical insight into the model's behavior and intuitive feedback on its real-world applicability.

Time and Space Analysis

Time Analysis

The overall time complexity of the project is dominated by the training phase of the CNN. For a CNN with L layers, the time complexity depends on:

- The number of convolutional layers and their kernel sizes
- The input image size (224×224)
- The number of training samples (~10,000)
- The number of epochs

Roughly, each convolutional layer has time complexity:

$$O(N * D * K^2 * M)$$

Where:

- N = number of input feature maps
- D = number of output feature maps
- K = kernel size
- M = spatial dimension of the feature map

Training time is also linear in the number of epochs and samples. Therefore, the training time complexity is:

$$O(E * S * T_{CNN})$$

Where:

- E = number of epochs
- S = number of training samples
- T_{CNN} = total time per sample

Time Complexity - Custom CNN:

313 batches * 32 images * 4 ms/image ≈ 40 ms per epoch

Therefore, training for 25 epochs, total time is roughly:
15 - 17 minutes

Time Complexity - ResNet18:

313 batches * 32 images * 4 ms/image ≈ 160 ms per epoch

Therefore, training for 25 epochs, total time is roughly:
27 - 28 minutes

Time and Space Analysis

Space Analysis - Custom CNN

Convolution Layers:

Layer	In → Out	Kernal	Parameters
conv 1	3 → 32	3 x 3	896
conv 2	32 → 64	3 x 3	18,492
conv 3	64 → 128	3 x 3	73,856
conv 4	128 → 256	3 x 3	295,168
conv 5	256 → 512	3 x 3	1,180,160

Fully-Connected Layers:

Layer	Parameters
fc 1	262,656
fc 2	10,260

Total Space Used:

1,841,492 parameters ≈ *7.36 MB*

Time and Space Analysis

Space Analysis - ResNet18

Layers:

Layer	Parameters
Initial conv + 16 layers	11.68 million
Final Fully Connected Layer	10,260

Total Space Used:

$$11{,}689{,}512 \text{ parameters} \approx 46.8\ MB$$

Results

In addition to our custom CNN, we evaluated a pretrained ResNet-18 model for comparison. ResNet-18, with its deeper architecture and transfer learning benefits, provided a useful benchmark. The table below summarizes the performance of both models on the test set:

Features	**Custom_CNN_model**	**ResNet18 Model (Pretrained)**
Model Type	Custom-built convolutional neural network	Deep residual network (ResNet18) with pretrained weights
Architecture Depth	5 convolutional layers + 2 fully connected layers	18 convolutional layers with residual (skip) connections
Pretrained Weights	No (trained from scratch)	Yes (pretrained on ImageNet dataset)
Epochs	25 epochs	25 epochs
Accuracy	**0.8550**	**0.8845**
F1-score	**0.8562**	**0.8865**
Training loss	**0.3467**	**0.1176**
Validation loss	**0.6797**	**0.4720**
Hardware Requirement	Moderate	Higher GPU memory and compute power required