Garbage Image Classification with Deep learning

This project develops a deep learing-based image classification system to automatically categorize waste items into different types such as paper, metal, plastic, glass, cardboard, and general trash. Using convolutional neural networks trained from scratch, the system aims to achieve high accuracy in distinguishing between various waste categories.

The model will be trained on the Garbage Dataset Classification from Kaggle, which contains labeled images of different waste types. The project focuses on creating a custom CNN architecture that can distinguish between biodegradable and non-biodegradable waste while further categorizing items into specific subcategories such as e-waste, food waste, and plastic bottles.

Key Features:

- Multi-class image classification for waste categorization
- Custom CNN architecture built from scratch
- Data augmentation techniques to improve model generalization
- Support for both biodegradable/non-biodegradable classification and detailed waste type identification
- Model performance evaluation and optimization

Technologies: Python, TensorFlow/PyTorch, OpenCV, custom convolutional neural networks, data preprocessing and augmentation libraries

Applications: Environmental sustainability, automated waste management systems, recycling optimization, and smart city initiatives.

Step 1: Data Loading and Initial Exploration

Overview

This first step involves loading the garbage classification dataset and performing initial exploration to understand the data structure, distribution, and basic characteristics.

Key Components:

1. **Directory Structure Analysis**:
    - Explores the dataset folder structure to identify waste categories
    - Counts the number of images in each category
    - Provides insights into data distribution
2. **Data Distribution Visualization**:
    - Creates a bar chart showing the number of images per waste category

- Helps identify potential class imbalance issues
- Provides visual overview of dataset composition

3. **Basic Statistics**:
    - Calculates total number of images and categories
    - Computes average, minimum, and maximum images per category
    - Identifies potential data imbalance that may need addressing

Expected Output:

- Console output showing image counts for each waste category
- Bar chart visualization of data distribution
- Summary statistics about the dataset

Notes:

- Update the `data_dir` variable with the actual path to your downloaded dataset
- This step helps in understanding data preprocessing requirements
- Identifies if data augmentation will be needed for balanced training

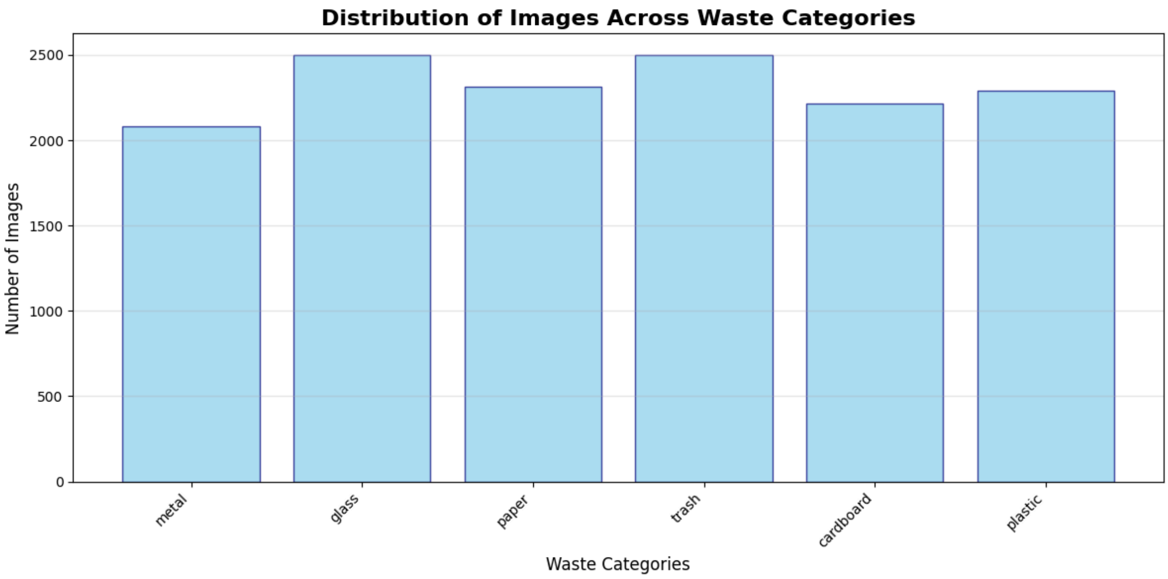
Next Steps:

After this exploration, we'll proceed with image preprocessing, data augmentation, and train-test split preparation.

In [1]:
```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from PIL import Image
import cv2
from collections import Counter

# Set up the data directory path
data_dir = "/kaggle/input/garbage-dataset-classification/Garbage_Dataset_Classif

# Function to explore the dataset structure
def explore_dataset_structure(data_dir):
    """
    Explore the directory structure and count images in each category
    """
    categories = []
    image_counts = []

    # List all subdirectories (categories)
    if os.path.exists(data_dir):
        for category in os.listdir(data_dir):
            category_path = os.path.join(data_dir, category)
            if os.path.isdir(category_path):
                # Count images in each category
                image_files = [f for f in os.listdir(category_path)
                               if f.lower().endswith(('.png', '.jpg', '.jpeg'))]
                categories.append(category)
```

```
                image_counts.append(len(image_files))
                print(f"{category}: {len(image_files)} images")

    return categories, image_counts

# Explore the dataset
print("Dataset Structure:")
print("=" * 50)
categories, image_counts = explore_dataset_structure(data_dir)

# Create a visualization of the dataset distribution
plt.figure(figsize=(12, 6))
plt.bar(categories, image_counts, color='skyblue', edgecolor='navy', alpha=0.7)
plt.title('Distribution of Images Across Waste Categories', fontsize=16, fontwei
plt.xlabel('Waste Categories', fontsize=12)
plt.ylabel('Number of Images', fontsize=12)
plt.xticks(rotation=45, ha='right')
plt.grid(axis='y', alpha=0.3)
plt.tight_layout()
plt.show()

# Display basic statistics
total_images = sum(image_counts)
print(f"\nDataset Summary:")
print(f"Total categories: {len(categories)}")
print(f"Total images: {total_images}")
print(f"Average images per category: {total_images/len(categories):.1f}")
print(f"Min images in a category: {min(image_counts)}")
print(f"Max images in a category: {max(image_counts)}")
```

```
Dataset Structure:
==================================================
metal: 2084 images
glass: 2500 images
paper: 2315 images
trash: 2500 images
cardboard: 2214 images
plastic: 2288 images
```



```
Dataset Summary:
Total categories: 6
Total images: 13901
Average images per category: 2316.8
Min images in a category: 2084
Max images in a category: 2500
```

Step 2: Image Preprocessing and Data Preparation

Overview

This step handles the crucial preprocessing of images and prepares the data for training the CNN model. It includes image loading, normalization, data splitting, and augmentation.

Key Components:

1. **Image Loading and Preprocessing**:
    - Loads all images from the dataset directories
    - Resizes images to a consistent size (224x224 pixels)
    - Normalizes pixel values to the range [0, 1] for better training performance
    - Creates numerical label mapping for categories
2. **Data Splitting**:
    - Splits data into training (60%), validation (20%), and test (20%) sets
    - Uses stratified sampling to maintain class distribution across splits
    - Ensures each subset has representative samples from all waste categories
3. **One-Hot Encoding**:
    - Converts integer labels to categorical format required for multi-class classification
    - Prepares labels for use with categorical crossentropy loss function
4. **Data Augmentation**:
    - Applies various transformations to training images (rotation, shifts, flips, zoom, shear)
    - Increases dataset diversity and helps prevent overfitting
    - Only applied to training set to maintain validation/test integrity
5. **Data Generators**:
    - Creates efficient data pipelines using Keras ImageDataGenerator
    - Enables batch processing and memory-efficient training
    - Handles data flow during model training

Key Parameters:

- **Image Size**: 224x224 pixels (standard for many CNN architectures)
- **Batch Size**: 32 (balance between memory usage and training stability)

- **Data Split**: 60% train, 20% validation, 20% test

Expected Output:

- Preprocessed image arrays ready for training
- Train/validation/test data generators
- Visualization of sample augmented images
- Summary of data dimensions and splits

Data Augmentation Techniques:

- **Rotation**: Up to 20 degrees
- **Width/Height Shift**: Up to 20%
- **Horizontal Flip**: Random flipping
- **Zoom**: Up to 20% zoom in/out
- **Shear**: Geometric transformation for robustness

Memory Considerations:

- Uses data generators to avoid loading entire dataset into memory
- Processes images in batches for efficient GPU utilization
- Suitable for datasets of various sizes

Next Steps:

After preprocessing, we'll design and build the custom CNN architecture for waste classification.

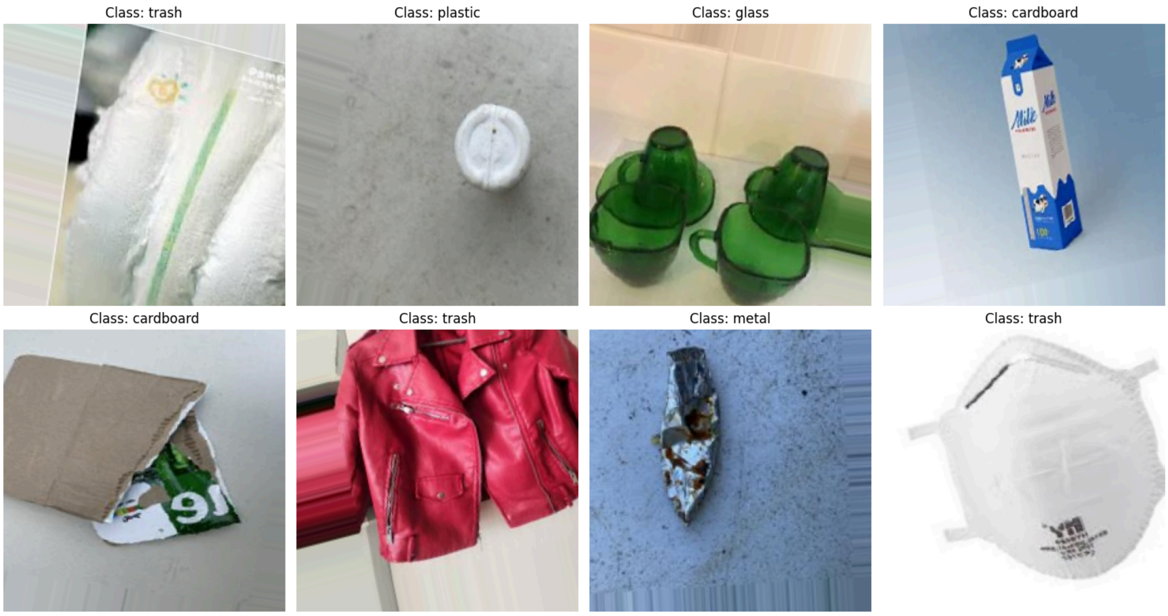
In [2]:

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator, load_img, i
from sklearn.model_selection import train_test_split
import numpy as np
from tqdm import tqdm

# Set image dimensions and parameters
IMG_HEIGHT = 224
IMG_WIDTH = 224
BATCH_SIZE = 32
CHANNELS = 3

def load_and_preprocess_images(data_dir, categories, img_height=224, img_width=2
    """
    Load and preprocess all images from the dataset
    """
    images = []
    labels = []

    # Create label mapping
    label_map = {category: idx for idx, category in enumerate(categories)}
    print("Label mapping:", label_map)

    for category in categories:
        category_path = os.path.join(data_dir, category)
```

```
        image_files = [f for f in os.listdir(category_path)
                      if f.lower().endswith(('.png', '.jpg', '.jpeg'))]

        print(f"Processing {category}: {len(image_files)} images")

        for image_file in tqdm(image_files, desc=f"Loading {category}"):
            try:
                # Load and resize image
                img_path = os.path.join(category_path, image_file)
                img = load_img(img_path, target_size=(img_height, img_width))
                img_array = img_to_array(img)

                # Normalize pixel values to [0, 1]
                img_array = img_array / 255.0

                images.append(img_array)
                labels.append(label_map[category])

            except Exception as e:
                print(f"Error loading {image_file}: {e}")
                continue

    return np.array(images), np.array(labels), label_map

# Load and preprocess all images
print("Loading and preprocessing images...")
X, y, label_mapping = load_and_preprocess_images(data_dir, categories, IMG_HEIGH

print(f"Images shape: {X.shape}")
print(f"Labels shape: {y.shape}")
print(f"Number of classes: {len(np.unique(y))}")

# Split the data into train, validation, and test sets
X_temp, X_test, y_temp, y_test = train_test_split(X, y, test_size=0.2, random_st
X_train, X_val, y_train, y_val = train_test_split(X_temp, y_temp, test_size=0.25

print(f"\nData split:")
print(f"Training set: {X_train.shape[0]} images")
print(f"Validation set: {X_val.shape[0]} images")
print(f"Test set: {X_test.shape[0]} images")

# Convert labels to categorical (one-hot encoding)
num_classes = len(categories)
y_train_cat = tf.keras.utils.to_categorical(y_train, num_classes)
y_val_cat = tf.keras.utils.to_categorical(y_val, num_classes)
y_test_cat = tf.keras.utils.to_categorical(y_test, num_classes)

print(f"One-hot encoded labels shape: {y_train_cat.shape}")

# Data augmentation for training set
train_datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    zoom_range=0.2,
    shear_range=0.2,
    fill_mode='nearest'
)
```

```
# No augmentation for validation and test sets
val_test_datagen = ImageDataGenerator()

# Create data generators
train_generator = train_datagen.flow(X_train, y_train_cat, batch_size=BATCH_SIZE
val_generator = val_test_datagen.flow(X_val, y_val_cat, batch_size=BATCH_SIZE, s
test_generator = val_test_datagen.flow(X_test, y_test_cat, batch_size=BATCH_SIZE

print(f"\nData generators created successfully!")
print(f"Training batches: {len(train_generator)}")
print(f"Validation batches: {len(val_generator)}")
print(f"Test batches: {len(test_generator)}")

# Visualize some sample images with augmentation
def visualize_sample_images(generator, label_mapping, num_samples=8):
    """
    Visualize sample images from the data generator
    """
    # Get a batch of images
    batch_x, batch_y = next(generator)

    # Reverse label mapping for display
    reverse_label_map = {v: k for k, v in label_mapping.items()}

    plt.figure(figsize=(15, 8))
    for i in range(min(num_samples, len(batch_x))):
        plt.subplot(2, 4, i + 1)
        plt.imshow(batch_x[i])
        label_idx = np.argmax(batch_y[i])
        plt.title(f"Class: {reverse_label_map[label_idx]}")
        plt.axis('off')

    plt.tight_layout()
    plt.show()

# Visualize sample augmented training images
print("\nSample augmented training images:")
visualize_sample_images(train_generator, label_mapping)

# Save preprocessing parameters for later use
preprocessing_params = {
    'img_height': IMG_HEIGHT,
    'img_width': IMG_WIDTH,
    'num_classes': num_classes,
    'label_mapping': label_mapping,
    'batch_size': BATCH_SIZE
}

print(f"\nPreprocessing completed successfully!")
print(f"Ready for model training with {num_classes} classes")
```

```
2025-08-13 18:45:00.967895: E external/local_xla/xla/stream_executor/cuda/cuda_ff
t.cc:477] Unable to register cuFFT factory: Attempting to register factory for pl
ugin cuFFT when one has already been registered
WARNING: All log messages before absl::InitializeLog() is called are written to S
TDERR
E0000 00:00:1755110701.175819     209 cuda_dnn.cc:8310] Unable to register cuDNN
factory: Attempting to register factory for plugin cuDNN when one has already bee
n registered
E0000 00:00:1755110701.234482     209 cuda_blas.cc:1418] Unable to register cuBLA
S factory: Attempting to register factory for plugin cuBLAS when one has already
been registered
Loading and preprocessing images...
Label mapping: {'metal': 0, 'glass': 1, 'paper': 2, 'trash': 3, 'cardboard': 4,
'plastic': 5}
Processing metal: 2084 images
Loading metal: 100%|██████████| 2084/2084 [00:13<00:00, 153.18it/s]
Processing glass: 2500 images
Loading glass: 100%|██████████| 2500/2500 [00:14<00:00, 177.12it/s]
Processing paper: 2315 images
Loading paper: 100%|██████████| 2315/2315 [00:14<00:00, 164.67it/s]
Processing trash: 2500 images
Loading trash: 100%|██████████| 2500/2500 [00:14<00:00, 166.68it/s]
Processing cardboard: 2214 images
Loading cardboard: 100%|██████████| 2214/2214 [00:12<00:00, 175.92it/s]
Processing plastic: 2288 images
Loading plastic: 100%|██████████| 2288/2288 [00:13<00:00, 169.46it/s]
Images shape: (13901, 224, 224, 3)
Labels shape: (13901,)
Number of classes: 6

Data split:
Training set: 8340 images
Validation set: 2780 images
Test set: 2781 images
One-hot encoded labels shape: (8340, 6)

Data generators created successfully!
Training batches: 261
Validation batches: 87
Test batches: 87

Sample augmented training images:
```




```
Preprocessing completed successfully!
Ready for model training with 6 classes
```

Step 3: Custom CNN Model Architecture Design

Overview

This step designs and builds a custom Convolutional Neural Network (CNN) architecture specifically tailored for garbage classification. The model is built from scratch without using pre-trained weights.

Architecture Design:

Convolutional Blocks:

1. **Block 1**: 32 filters, basic feature detection
2. **Block 2**: 64 filters, intermediate pattern recognition
3. **Block 3**: 128 filters, complex feature extraction
4. **Block 4**: 256 filters, high-level feature abstraction

Key Architecture Components:

1. **Convolutional Layers**:
    - Uses 3x3 kernel size for optimal feature extraction
    - Padding='same' to preserve spatial dimensions
    - Progressive filter increase (32→64→128→256) for hierarchical learning
2. **Batch Normalization**:
    - Applied after each convolutional layer
    - Stabilizes training and accelerates convergence
    - Reduces internal covariate shift
3. **Activation Functions**:
    - ReLU activation for non-linearity

   - Prevents vanishing gradient problem
   - Computationally efficient
4. **Max Pooling**:

   - 2x2 pooling reduces spatial dimensions by half
   - Decreases computational load
   - Provides translation invariance
5. **Dropout Layers**:

   - 25% dropout after convolutional blocks
   - 50% dropout in dense layers
   - Prevents overfitting and improves generalization
6. **Dense Layers**:

   - Two fully connected layers (512→256 neurons)
   - Final softmax layer for multi-class classification

Model Configuration:

Compilation Parameters:

- **Optimizer**: Adam with learning rate 0.001
- **Loss Function**: Categorical crossentropy for multi-class classification
- **Metrics**: Accuracy for comprehensive evaluation

Training Callbacks:

1. **EarlyStopping**: Prevents overfitting (patience=10 epochs)
2. **ReduceLROnPlateau**: Dynamically adjusts learning rate (factor=0.2, patience=5)
3. **ModelCheckpoint**: Saves best model based on validation accuracy

Model Specifications:

- **Input Shape**: 224×224×3 (RGB images)
- **Output Classes**: Number of waste categories in dataset
- **Architecture Type**: Sequential CNN with progressive feature learning
- **Total Parameters**: Calculated and displayed for transparency

Advantages of This Architecture:

- **Custom Design**: Tailored specifically for waste classification task
- **Progressive Learning**: Hierarchical feature extraction from simple to complex
- **Regularization**: Multiple dropout and batch normalization layers
- **Adaptive Training**: Smart callbacks for optimal training

Memory and Computational Considerations:

- Balanced depth vs. complexity trade-off
- Efficient parameter usage compared to deeper networks

- Suitable for training on standard GPUs
- Progressive filter increase optimizes feature learning

Next Steps:

After model creation, we'll proceed with training the CNN using our preprocessed data and evaluate its performance on the validation set.

Step 3: Custom CNN Model Architecture Design

Overview

This step designs and builds a custom Convolutional Neural Network (CNN) architecture specifically tailored for garbage classification. The model is built from scratch without using pre-trained weights.

Architecture Design:

Convolutional Blocks:

1. **Block 1**: 32 filters, basic feature detection
2. **Block 2**: 64 filters, intermediate pattern recognition
3. **Block 3**: 128 filters, complex feature extraction
4. **Block 4**: 256 filters, high-level feature abstraction

Key Architecture Components:

1. **Convolutional Layers**:
   - Uses 3x3 kernel size for optimal feature extraction
   - Padding='same' to preserve spatial dimensions
   - Progressive filter increase (32→64→128→256) for hierarchical learning
2. **Batch Normalization**:
   - Applied after each convolutional layer
   - Stabilizes training and accelerates convergence
   - Reduces internal covariate shift
3. **Activation Functions**:
   - ReLU activation for non-linearity
   - Prevents vanishing gradient problem
   - Computationally efficient
4. **Max Pooling**:
   - 2x2 pooling reduces spatial dimensions by half
   - Decreases computational load
   - Provides translation invariance
5. **Dropout Layers**:

- 25% dropout after convolutional blocks
- 50% dropout in dense layers
- Prevents overfitting and improves generalization

6. **Dense Layers**:

   - Two fully connected layers (512→256 neurons)
   - Final softmax layer for multi-class classification

Model Configuration:

Compilation Parameters:

- **Optimizer**: Adam with learning rate 0.001
- **Loss Function**: Categorical crossentropy for multi-class classification
- **Metrics**: Accuracy for comprehensive evaluation

Training Callbacks:

1. **EarlyStopping**: Prevents overfitting (patience=10 epochs)
2. **ReduceLROnPlateau**: Dynamically adjusts learning rate (factor=0.2, patience=5)
3. **ModelCheckpoint**: Saves best model based on validation accuracy

Model Specifications:

- **Input Shape**: 224×224×3 (RGB images)
- **Output Classes**: Number of waste categories in dataset
- **Architecture Type**: Sequential CNN with progressive feature learning
- **Total Parameters**: Calculated and displayed for transparency

Advantages of This Architecture:

- **Custom Design**: Tailored specifically for waste classification task
- **Progressive Learning**: Hierarchical feature extraction from simple to complex
- **Regularization**: Multiple dropout and batch normalization layers
- **Adaptive Training**: Smart callbacks for optimal training

Memory and Computational Considerations:

- Balanced depth vs. complexity trade-off
- Efficient parameter usage compared to deeper networks
- Suitable for training on standard GPUs
- Progressive filter increase optimizes feature learning

Next Steps:

After model creation, we'll proceed with training the CNN using our preprocessed data and evaluate its performance on the validation set.

In [4]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (
    Conv2D, MaxPooling2D, Flatten, Dense, Dropout,
    BatchNormalization, Activation
)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau, ModelCh

def create_custom_cnn_model(input_shape, num_classes):
    """
    Create a custom CNN architecture for garbage classification
    """
    model = Sequential([
        # First Convolutional Block
        Conv2D(32, (3, 3), input_shape=input_shape, padding='same'),
        BatchNormalization(),
        Activation('relu'),
        Conv2D(32, (3, 3), padding='same'),
        BatchNormalization(),
        Activation('relu'),
        MaxPooling2D(pool_size=(2, 2)),
        Dropout(0.25),

        # Second Convolutional Block
        Conv2D(64, (3, 3), padding='same'),
        BatchNormalization(),
        Activation('relu'),
        Conv2D(64, (3, 3), padding='same'),
        BatchNormalization(),
        Activation('relu'),
        MaxPooling2D(pool_size=(2, 2)),
        Dropout(0.25),

        # Third Convolutional Block
        Conv2D(128, (3, 3), padding='same'),
        BatchNormalization(),
        Activation('relu'),
        Conv2D(128, (3, 3), padding='same'),
        BatchNormalization(),
        Activation('relu'),
        MaxPooling2D(pool_size=(2, 2)),
        Dropout(0.25),

        # Fourth Convolutional Block
        Conv2D(256, (3, 3), padding='same'),
        BatchNormalization(),
        Activation('relu'),
        Conv2D(256, (3, 3), padding='same'),
        BatchNormalization(),
        Activation('relu'),
        MaxPooling2D(pool_size=(2, 2)),
        Dropout(0.25),

        # Flatten and Dense Layers
        Flatten(),
        Dense(512),
        BatchNormalization(),
        Activation('relu'),
```

```
        Dropout(0.5),

        Dense(256),
        BatchNormalization(),
        Activation('relu'),
        Dropout(0.5),

        # Output Layer
        Dense(num_classes, activation='softmax')
    ])

    return model

# Create the model
input_shape = (IMG_HEIGHT, IMG_WIDTH, CHANNELS)
model = create_custom_cnn_model(input_shape, num_classes)

# Display model architecture
print("Custom CNN Model Architecture:")
print("=" * 50)
model.summary()

# Compile the model
model.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

print(f"\nModel compiled successfully!")
print(f"Optimizer: Adam (lr=0.001)")
print(f"Loss function: Categorical Crossentropy")
print(f"Metrics: Accuracy")

# Calculate total trainable parameters
total_params = model.count_params()
print(f"\nTotal trainable parameters: {total_params:,}")

# Set up callbacks for training
callbacks = [
    # Early stopping to prevent overfitting
    EarlyStopping(
        monitor='val_loss',
        patience=10,
        restore_best_weights=True,
        verbose=1
    ),

    # Reduce learning rate when loss plateaus
    ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.2,
        patience=5,
        min_lr=0.0001,
        verbose=1
    ),

    # Save best model during training
    ModelCheckpoint(
        filepath='best_garbage_classifier.h5',
```

```
        monitor='val_accuracy',
        save_best_only=True,
        save_weights_only=False,
        verbose=1
    )
]

print(f"\nCallbacks configured:")
print("- EarlyStopping: Prevents overfitting (patience=10)")
print("- ReduceLROnPlateau: Adapts learning rate (factor=0.2, patience=5)")
print("- ModelCheckpoint: Saves best model based on validation accuracy")

# Visualize model architecture (optional)
try:
    tf.keras.utils.plot_model(
        model,
        to_file='garbage_cnn_architecture.png',
        show_shapes=True,
        show_layer_names=True,
        rankdir='TB'
    )
    print("\nModel architecture diagram saved as 'garbage_cnn_architecture.png'"
except:
    print("\nNote: Install pydot and graphviz to generate architecture diagram")

# Display layer details
print(f"\nModel Architecture Details:")
print("=" * 70)
for i, layer in enumerate(model.layers):
    if hasattr(layer, 'filters'):
        print(f"Layer {i+1:2d}: {layer.name:20s} | Filters: {layer.filters:3d} |
    elif hasattr(layer, 'units'):
        print(f"Layer {i+1:2d}: {layer.name:20s} | Units: {layer.units:4d}")
    else:
        print(f"Layer {i+1:2d}: {layer.name:20s}")

print(f"\nModel ready for training!")
print(f"Input shape: {input_shape}")
print(f"Output classes: {num_classes}")
```

```
Custom CNN Model Architecture:
==================================================
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.
py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a laye
r. When using Sequential models, prefer using an `Input(shape)` object as the fir
st layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_8 (Conv2D)	(None, 224, 224, 32)	896
batch_normalization_10 (BatchNormalization)	(None, 224, 224, 32)	128
activation_10 (Activation)	(None, 224, 224, 32)	0
conv2d_9 (Conv2D)	(None, 224, 224, 32)	9,248
batch_normalization_11 (BatchNormalization)	(None, 224, 224, 32)	128
activation_11 (Activation)	(None, 224, 224, 32)	0
max_pooling2d_4 (MaxPooling2D)	(None, 112, 112, 32)	0
dropout_6 (Dropout)	(None, 112, 112, 32)	0
conv2d_10 (Conv2D)	(None, 112, 112, 64)	18,496
batch_normalization_12 (BatchNormalization)	(None, 112, 112, 64)	256
activation_12 (Activation)	(None, 112, 112, 64)	0
conv2d_11 (Conv2D)	(None, 112, 112, 64)	36,928
batch_normalization_13 (BatchNormalization)	(None, 112, 112, 64)	256
activation_13 (Activation)	(None, 112, 112, 64)	0
max_pooling2d_5 (MaxPooling2D)	(None, 56, 56, 64)	0
dropout_7 (Dropout)	(None, 56, 56, 64)	0
conv2d_12 (Conv2D)	(None, 56, 56, 128)	73,856
batch_normalization_14 (BatchNormalization)	(None, 56, 56, 128)	512
activation_14 (Activation)	(None, 56, 56, 128)	0
conv2d_13 (Conv2D)	(None, 56, 56, 128)	147,584
batch_normalization_15 (BatchNormalization)	(None, 56, 56, 128)	512
activation_15 (Activation)	(None, 56, 56, 128)	0
max_pooling2d_6 (MaxPooling2D)	(None, 28, 28, 128)	0
dropout_8 (Dropout)	(None, 28, 28, 128)	0
conv2d_14 (Conv2D)	(None, 28, 28, 256)	295,168
batch_normalization_16	(None, 28, 28, 256)	1,024

(BatchNormalization)		
activation_16 (Activation)	(None, 28, 28, 256)	0
conv2d_15 (Conv2D)	(None, 28, 28, 256)	590,080
batch_normalization_17 (BatchNormalization)	(None, 28, 28, 256)	1,024
activation_17 (Activation)	(None, 28, 28, 256)	0
max_pooling2d_7 (MaxPooling2D)	(None, 14, 14, 256)	0
dropout_9 (Dropout)	(None, 14, 14, 256)	0
flatten_1 (Flatten)	(None, 50176)	0
dense_3 (Dense)	(None, 512)	25,690,624
batch_normalization_18 (BatchNormalization)	(None, 512)	2,048
activation_18 (Activation)	(None, 512)	0
dropout_10 (Dropout)	(None, 512)	0
dense_4 (Dense)	(None, 256)	131,328
batch_normalization_19 (BatchNormalization)	(None, 256)	1,024
activation_19 (Activation)	(None, 256)	0
dropout_11 (Dropout)	(None, 256)	0
dense_5 (Dense)	(None, 6)	1,542

Total params: 27,002,662 (103.01 MB)

Trainable params: 26,999,206 (102.99 MB)

Non-trainable params: 3,456 (13.50 KB)

```
Model compiled successfully!
Optimizer: Adam (lr=0.001)
Loss function: Categorical Crossentropy
Metrics: Accuracy

Total trainable parameters: 27,002,662

Callbacks configured:
- EarlyStopping: Prevents overfitting (patience=10)
- ReduceLROnPlateau: Adapts learning rate (factor=0.2, patience=5)
- ModelCheckpoint: Saves best model based on validation accuracy

Model architecture diagram saved as 'garbage_cnn_architecture.png'

Model Architecture Details:
======================================================================
Layer  1: conv2d_8             | Filters:  32 | Kernel: (3, 3)
Layer  2: batch_normalization_10
Layer  3: activation_10
Layer  4: conv2d_9             | Filters:  32 | Kernel: (3, 3)
Layer  5: batch_normalization_11
Layer  6: activation_11
Layer  7: max_pooling2d_4
Layer  8: dropout_6
Layer  9: conv2d_10            | Filters:  64 | Kernel: (3, 3)
Layer 10: batch_normalization_12
Layer 11: activation_12
Layer 12: conv2d_11            | Filters:  64 | Kernel: (3, 3)
Layer 13: batch_normalization_13
Layer 14: activation_13
Layer 15: max_pooling2d_5
Layer 16: dropout_7
Layer 17: conv2d_12            | Filters: 128 | Kernel: (3, 3)
Layer 18: batch_normalization_14
Layer 19: activation_14
Layer 20: conv2d_13            | Filters: 128 | Kernel: (3, 3)
Layer 21: batch_normalization_15
Layer 22: activation_15
Layer 23: max_pooling2d_6
Layer 24: dropout_8
Layer 25: conv2d_14            | Filters: 256 | Kernel: (3, 3)
Layer 26: batch_normalization_16
Layer 27: activation_16
Layer 28: conv2d_15            | Filters: 256 | Kernel: (3, 3)
Layer 29: batch_normalization_17
Layer 30: activation_17
Layer 31: max_pooling2d_7
Layer 32: dropout_9
Layer 33: flatten_1
Layer 34: dense_3              | Units:  512
Layer 35: batch_normalization_18
Layer 36: activation_18
Layer 37: dropout_10
Layer 38: dense_4              | Units:  256
Layer 39: batch_normalization_19
Layer 40: activation_19
Layer 41: dropout_11
Layer 42: dense_5              | Units:    6

Model ready for training!
```

```
Input shape: (224, 224, 3)
Output classes: 6
```

In [5]:
```
import time
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns

# Training parameters
EPOCHS = 50
STEPS_PER_EPOCH = len(train_generator)
VALIDATION_STEPS = len(val_generator)

print(f"Training Configuration:")
print(f"Total epochs: {EPOCHS}")
print(f"Steps per epoch: {STEPS_PER_EPOCH}")
print(f"Validation steps: {VALIDATION_STEPS}")
print(f"Training samples: {len(X_train)}")
print(f"Validation samples: {len(X_val)}")

# Start training
print(f"\nStarting model training...")
print("=" * 50)

start_time = time.time()

# Train the model
history = model.fit(
    train_generator,
    steps_per_epoch=STEPS_PER_EPOCH,
    epochs=EPOCHS,
    validation_data=val_generator,
    validation_steps=VALIDATION_STEPS,
    callbacks=callbacks,
    verbose=1
)

end_time = time.time()
training_time = end_time - start_time

print(f"\nTraining completed!")
print(f"Total training time: {training_time/60:.2f} minutes")

# Plot training history
def plot_training_history(history):
    """
    Plot training and validation metrics
    """
    fig, axes = plt.subplots(1, 2, figsize=(15, 5))

    # Plot training & validation accuracy
    axes[0].plot(history.history['accuracy'], label='Training Accuracy', color='
    axes[0].plot(history.history['val_accuracy'], label='Validation Accuracy', c
    axes[0].set_title('Model Accuracy')
    axes[0].set_xlabel('Epoch')
    axes[0].set_ylabel('Accuracy')
    axes[0].legend()
    axes[0].grid(True, alpha=0.3)
```

```
    # Plot training & validation loss
    axes[1].plot(history.history['loss'], label='Training Loss', color='blue')
    axes[1].plot(history.history['val_loss'], label='Validation Loss', color='re
    axes[1].set_title('Model Loss')
    axes[1].set_xlabel('Epoch')
    axes[1].set_ylabel('Loss')
    axes[1].legend()
    axes[1].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('training_history.png', dpi=300, bbox_inches='tight')
    plt.show()

# Plot training results
plot_training_history(history)

# Get final training metrics
final_train_accuracy = history.history['accuracy'][-1]
final_val_accuracy = history.history['val_accuracy'][-1]
final_train_loss = history.history['loss'][-1]
final_val_loss = history.history['val_loss'][-1]

print(f"\nFinal Training Results:")
print("=" * 40)
print(f"Training Accuracy: {final_train_accuracy:.4f}")
print(f"Validation Accuracy: {final_val_accuracy:.4f}")
print(f"Training Loss: {final_train_loss:.4f}")
print(f"Validation Loss: {final_val_loss:.4f}")

# Check for overfitting
accuracy_gap = final_train_accuracy - final_val_accuracy
loss_gap = final_val_loss - final_train_loss

print(f"\nOverfitting Analysis:")
print(f"Accuracy gap (Train - Val): {accuracy_gap:.4f}")
print(f"Loss gap (Val - Train): {loss_gap:.4f}")

if accuracy_gap > 0.1:
    print("⚠️ Warning: Potential overfitting detected (accuracy gap > 0.1)")
elif accuracy_gap > 0.05:
    print("⚠️ Caution: Moderate overfitting detected (accuracy gap > 0.05)")
else:
    print("✅ Good: No significant overfitting detected")

# Load the best model saved by ModelCheckpoint
try:
    best_model = tf.keras.models.load_model('best_garbage_classifier.h5')
    print(f"\n✅ Best model loaded successfully!")

    # Evaluate best model on validation set
    val_loss, val_accuracy = best_model.evaluate(val_generator, verbose=0)
    print(f"Best model validation accuracy: {val_accuracy:.4f}")

except:
    print(f"\n⚠️ Using current model (best model checkpoint not found)")
    best_model = model

# Generate predictions on validation set for detailed analysis
print(f"\nGenerating validation predictions...")
val_generator.reset()
```

```
val_predictions = best_model.predict(val_generator, verbose=1)
val_predicted_classes = np.argmax(val_predictions, axis=1)

# Get true classes from the original validation labels
val_true_classes = np.argmax(y_val_cat, axis=1)

# Ensure we have the same number of predictions and true labels
min_length = min(len(val_predicted_classes), len(val_true_classes))
val_predicted_classes = val_predicted_classes[:min_length]
val_true_classes = val_true_classes[:min_length]

print(f"Validation samples analyzed: {min_length}")

# Calculate top-3 accuracy manually
def calculate_top_k_accuracy(y_true, y_pred, k=3):
    """Calculate top-k accuracy manually"""
    top_k_pred = np.argsort(y_pred, axis=1)[:, -k:]
    correct = 0
    for i, true_label in enumerate(y_true):
        if true_label in top_k_pred[i]:
            correct += 1
    return correct / len(y_true)

# Calculate top-3 accuracy using the same number of samples
val_predictions_subset = val_predictions[:min_length]
top3_accuracy = calculate_top_k_accuracy(val_true_classes, val_predictions_subse
print(f"Manual Top-3 Accuracy: {top3_accuracy:.4f}")

# Create classification report
reverse_label_map = {v: k for k, v in label_mapping.items()}
class_names = [reverse_label_map[i] for i in range(num_classes)]

print(f"\nDetailed Classification Report:")
print("=" * 60)
print(classification_report(val_true_classes, val_predicted_classes,
                            target_names=class_names, digits=4))

# Create and plot confusion matrix
cm = confusion_matrix(val_true_classes, val_predicted_classes)
plt.figure(figsize=(12, 10))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names, yticklabels=class_names)
plt.title('Confusion Matrix - Validation Set', fontsize=16, fontweight='bold')
plt.xlabel('Predicted Label', fontsize=12)
plt.ylabel('True Label', fontsize=12)
plt.xticks(rotation=45, ha='right')
plt.yticks(rotation=0)
plt.tight_layout()
plt.savefig('confusion_matrix.png', dpi=300, bbox_inches='tight')
plt.show()

# Calculate per-class accuracy
class_accuracy = cm.diagonal() / cm.sum(axis=1)
print(f"\nPer-Class Accuracy:")
print("=" * 40)
for i, (class_name, accuracy) in enumerate(zip(class_names, class_accuracy)):
    print(f"{class_name:15s}: {accuracy:.4f} ({accuracy*100:.2f}%)")

# Find best and worst performing classes
best_class_idx = np.argmax(class_accuracy)
```

```python
worst_class_idx = np.argmin(class_accuracy)

print(f"\n📊 Performance Summary:")
print(f"Best performing class: {class_names[best_class_idx]} ({class_accuracy[be
print(f"Worst performing class: {class_names[worst_class_idx]} ({class_accuracy[
print(f"Overall Top-3 Accuracy: {top3_accuracy:.4f}")

print(f"\n✅ Training and validation completed successfully!")
print(f"Model ready for testing on the test set.")
```

```
Training Configuration:
Total epochs: 50
Steps per epoch: 261
Validation steps: 87
Training samples: 8340
Validation samples: 2780

Starting model training...
==================================================
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/trainers/data_adapters/py_datas
et_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__ini
t__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multipr
ocessing`, `max_queue_size`. Do not pass these arguments to `fit()`, as they will
be ignored.
  self._warn_if_super_not_called()
```

```
Epoch 1/50
```

```
WARNING: All log messages before absl::InitializeLog() is called are written to S
TDERR
I0000 00:00:1755111310.736863     250 service.cc:148] XLA service 0x7ea73000ada0
initialized for platform CUDA (this does not guarantee that XLA will be used). De
vices:
I0000 00:00:1755111310.737631     250 service.cc:156]   StreamExecutor device
(0): Tesla T4, Compute Capability 7.5
I0000 00:00:1755111310.737657     250 service.cc:156]   StreamExecutor device
(1): Tesla T4, Compute Capability 7.5
I0000 00:00:1755111311.932906     250 cuda_dnn.cc:529] Loaded cuDNN version 90300
I0000 00:00:1755111333.647660     250 device_compiler.h:188] Compiled cluster usi
ng XLA!  This line is logged at most once for the lifetime of the process.
```

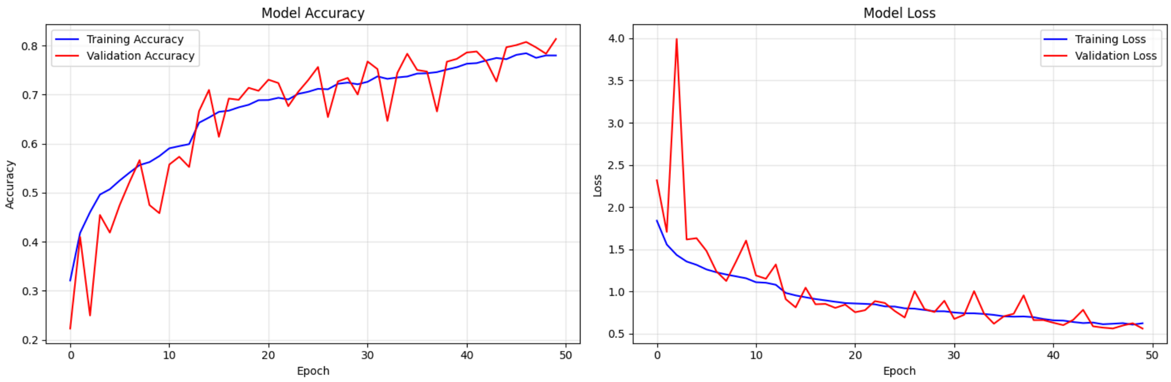
```
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 426ms/step - accuracy: 0.2672 - loss: 2.0043
Epoch 1: val_accuracy improved from -inf to 0.22302, saving model to best_garbage
_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 156s 466ms/step - accuracy: 0.2674 - loss: 2.0036 -
val_accuracy: 0.2230 - val_loss: 2.3180 - learning_rate: 0.0010
Epoch 2/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 362ms/step - accuracy: 0.4041 - loss: 1.5964
Epoch 2: val_accuracy improved from 0.22302 to 0.40971, saving model to best_garb
age_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 100s 381ms/step - accuracy: 0.4041 - loss: 1.5962 -
val_accuracy: 0.4097 - val_loss: 1.7058 - learning_rate: 0.0010
Epoch 3/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 365ms/step - accuracy: 0.4597 - loss: 1.4325
Epoch 3: val_accuracy did not improve from 0.40971
261/261 ━━━━━━━━━━━━━━━━━━━━ 99s 380ms/step - accuracy: 0.4597 - loss: 1.4325 - v
al_accuracy: 0.2496 - val_loss: 3.9922 - learning_rate: 0.0010
Epoch 4/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 369ms/step - accuracy: 0.4909 - loss: 1.3818
Epoch 4: val_accuracy improved from 0.40971 to 0.45468, saving model to best_garb
age_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 101s 388ms/step - accuracy: 0.4909 - loss: 1.3817 -
val_accuracy: 0.4547 - val_loss: 1.6159 - learning_rate: 0.0010
Epoch 5/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 363ms/step - accuracy: 0.5027 - loss: 1.3265
Epoch 5: val_accuracy did not improve from 0.45468
261/261 ━━━━━━━━━━━━━━━━━━━━ 99s 378ms/step - accuracy: 0.5027 - loss: 1.3265 - v
al_accuracy: 0.4187 - val_loss: 1.6316 - learning_rate: 0.0010
Epoch 6/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 366ms/step - accuracy: 0.5286 - loss: 1.2572
Epoch 6: val_accuracy improved from 0.45468 to 0.47518, saving model to best_garb
age_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 101s 386ms/step - accuracy: 0.5286 - loss: 1.2572 -
val_accuracy: 0.4752 - val_loss: 1.4812 - learning_rate: 0.0010
Epoch 7/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 369ms/step - accuracy: 0.5516 - loss: 1.2187
Epoch 7: val_accuracy improved from 0.47518 to 0.52266, saving model to best_garb
age_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 101s 388ms/step - accuracy: 0.5515 - loss: 1.2187 -
val_accuracy: 0.5227 - val_loss: 1.2386 - learning_rate: 0.0010
Epoch 8/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 372ms/step - accuracy: 0.5513 - loss: 1.2141
Epoch 8: val_accuracy improved from 0.52266 to 0.56655, saving model to best_garb
age_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 102s 391ms/step - accuracy: 0.5513 - loss: 1.2140 -
val_accuracy: 0.5665 - val_loss: 1.1245 - learning_rate: 0.0010
Epoch 9/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 371ms/step - accuracy: 0.5548 - loss: 1.1937
Epoch 9: val_accuracy did not improve from 0.56655
261/261 ━━━━━━━━━━━━━━━━━━━━ 101s 386ms/step - accuracy: 0.5548 - loss: 1.1937 -
val_accuracy: 0.4748 - val_loss: 1.3578 - learning_rate: 0.0010
Epoch 10/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 361ms/step - accuracy: 0.5825 - loss: 1.1434
Epoch 10: val_accuracy did not improve from 0.56655
261/261 ━━━━━━━━━━━━━━━━━━━━ 98s 376ms/step - accuracy: 0.5825 - loss: 1.1435 - v
al_accuracy: 0.4583 - val_loss: 1.6028 - learning_rate: 0.0010
Epoch 11/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 381ms/step - accuracy: 0.5835 - loss: 1.1213
Epoch 11: val_accuracy did not improve from 0.56655
261/261 ━━━━━━━━━━━━━━━━━━━━ 104s 396ms/step - accuracy: 0.5835 - loss: 1.1212 -
val_accuracy: 0.5576 - val_loss: 1.1896 - learning_rate: 0.0010
```

```
Epoch 12/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 376ms/step - accuracy: 0.6032 - loss: 1.0898
Epoch 12: val_accuracy improved from 0.56655 to 0.57338, saving model to best_gar
bage_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 103s 396ms/step - accuracy: 0.6032 - loss: 1.0898 -
val_accuracy: 0.5734 - val_loss: 1.1507 - learning_rate: 0.0010
Epoch 13/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 369ms/step - accuracy: 0.5981 - loss: 1.0861
Epoch 13: ReduceLROnPlateau reducing learning rate to 0.00020000000949949026.

Epoch 13: val_accuracy did not improve from 0.57338
261/261 ━━━━━━━━━━━━━━━━━━━━ 100s 384ms/step - accuracy: 0.5982 - loss: 1.0861 -
val_accuracy: 0.5525 - val_loss: 1.3199 - learning_rate: 0.0010
Epoch 14/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 373ms/step - accuracy: 0.6353 - loss: 1.0005
Epoch 14: val_accuracy improved from 0.57338 to 0.66655, saving model to best_gar
bage_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 103s 392ms/step - accuracy: 0.6353 - loss: 1.0004 -
val_accuracy: 0.6665 - val_loss: 0.9090 - learning_rate: 2.0000e-04
Epoch 15/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 367ms/step - accuracy: 0.6560 - loss: 0.9540
Epoch 15: val_accuracy improved from 0.66655 to 0.70971, saving model to best_gar
bage_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 101s 386ms/step - accuracy: 0.6560 - loss: 0.9540 -
val_accuracy: 0.7097 - val_loss: 0.8116 - learning_rate: 2.0000e-04
Epoch 16/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 371ms/step - accuracy: 0.6714 - loss: 0.9197
Epoch 16: val_accuracy did not improve from 0.70971
261/261 ━━━━━━━━━━━━━━━━━━━━ 101s 385ms/step - accuracy: 0.6713 - loss: 0.9198 -
val_accuracy: 0.6140 - val_loss: 1.0448 - learning_rate: 2.0000e-04
Epoch 17/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 367ms/step - accuracy: 0.6699 - loss: 0.9037
Epoch 17: val_accuracy did not improve from 0.70971
261/261 ━━━━━━━━━━━━━━━━━━━━ 100s 382ms/step - accuracy: 0.6699 - loss: 0.9037 -
val_accuracy: 0.6921 - val_loss: 0.8484 - learning_rate: 2.0000e-04
Epoch 18/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 384ms/step - accuracy: 0.6786 - loss: 0.8900
Epoch 18: val_accuracy did not improve from 0.70971
261/261 ━━━━━━━━━━━━━━━━━━━━ 104s 399ms/step - accuracy: 0.6786 - loss: 0.8900 -
val_accuracy: 0.6896 - val_loss: 0.8527 - learning_rate: 2.0000e-04
Epoch 19/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 361ms/step - accuracy: 0.6917 - loss: 0.8591
Epoch 19: val_accuracy improved from 0.70971 to 0.71403, saving model to best_gar
bage_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 99s 380ms/step - accuracy: 0.6916 - loss: 0.8591 - v
al_accuracy: 0.7140 - val_loss: 0.8058 - learning_rate: 2.0000e-04
Epoch 20/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 364ms/step - accuracy: 0.6826 - loss: 0.8725
Epoch 20: val_accuracy did not improve from 0.71403
261/261 ━━━━━━━━━━━━━━━━━━━━ 99s 379ms/step - accuracy: 0.6826 - loss: 0.8725 - v
al_accuracy: 0.7079 - val_loss: 0.8447 - learning_rate: 2.0000e-04
Epoch 21/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 376ms/step - accuracy: 0.6843 - loss: 0.8650
Epoch 21: val_accuracy improved from 0.71403 to 0.73058, saving model to best_gar
bage_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 103s 395ms/step - accuracy: 0.6843 - loss: 0.8650 -
val_accuracy: 0.7306 - val_loss: 0.7541 - learning_rate: 2.0000e-04
Epoch 22/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 367ms/step - accuracy: 0.6942 - loss: 0.8684
Epoch 22: val_accuracy did not improve from 0.73058
```

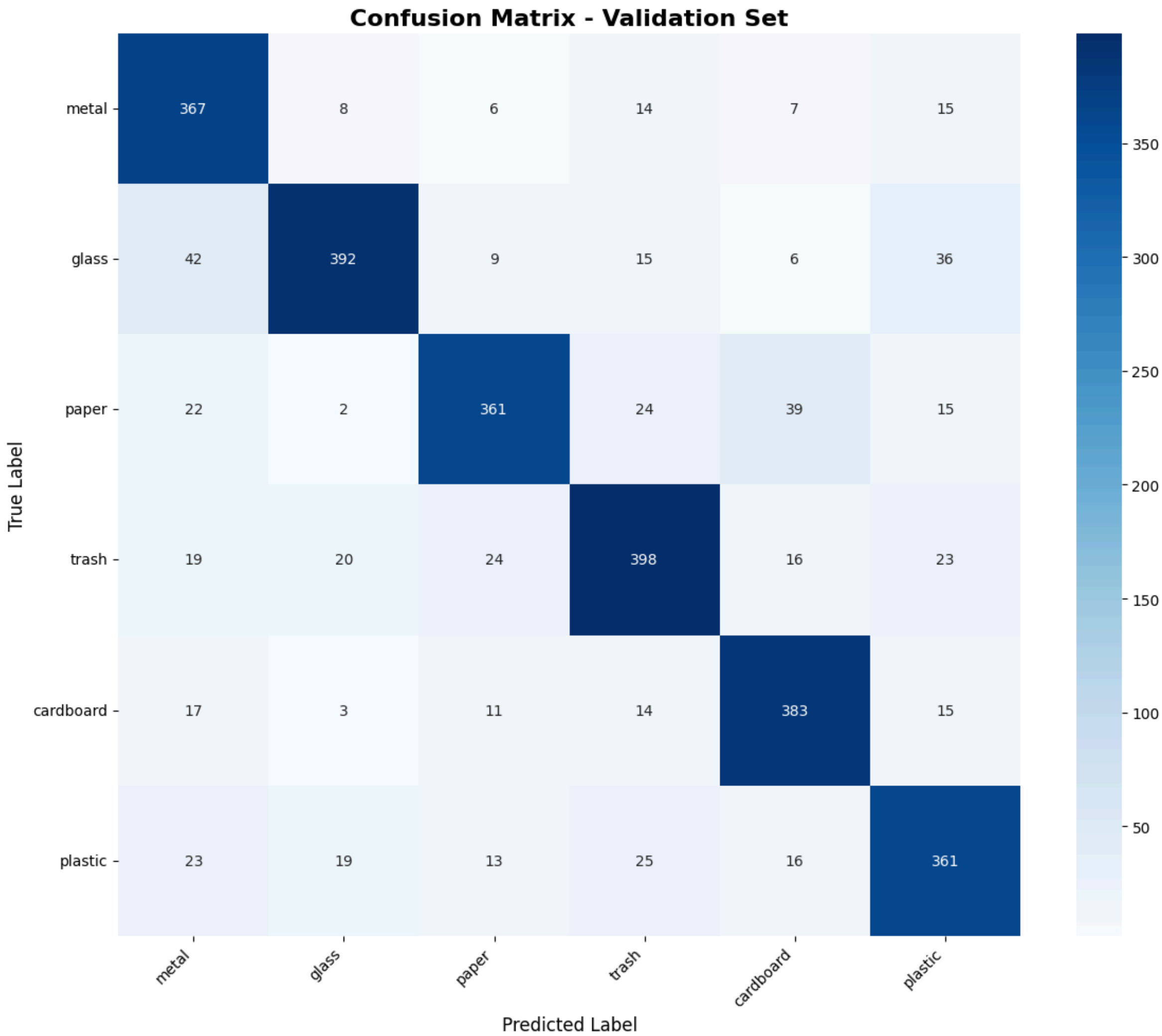
```
261/261 ━━━━━━━━━━━━━━━━━━━━ 100s 382ms/step - accuracy: 0.6942 - loss: 0.8683 -
val_accuracy: 0.7237 - val_loss: 0.7783 - learning_rate: 2.0000e-04
Epoch 23/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 364ms/step - accuracy: 0.6944 - loss: 0.8444
Epoch 23: val_accuracy did not improve from 0.73058
261/261 ━━━━━━━━━━━━━━━━━━━━ 99s 379ms/step - accuracy: 0.6943 - loss: 0.8444 - v
al_accuracy: 0.6766 - val_loss: 0.8859 - learning_rate: 2.0000e-04
Epoch 24/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 366ms/step - accuracy: 0.7062 - loss: 0.8104
Epoch 24: val_accuracy did not improve from 0.73058
261/261 ━━━━━━━━━━━━━━━━━━━━ 100s 381ms/step - accuracy: 0.7062 - loss: 0.8105 -
val_accuracy: 0.7058 - val_loss: 0.8634 - learning_rate: 2.0000e-04
Epoch 25/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 373ms/step - accuracy: 0.7099 - loss: 0.8219
Epoch 25: val_accuracy did not improve from 0.73058
261/261 ━━━━━━━━━━━━━━━━━━━━ 101s 387ms/step - accuracy: 0.7098 - loss: 0.8219 -
val_accuracy: 0.7295 - val_loss: 0.7687 - learning_rate: 2.0000e-04
Epoch 26/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 371ms/step - accuracy: 0.7084 - loss: 0.8086
Epoch 26: val_accuracy improved from 0.73058 to 0.75647, saving model to best_gar
bage_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 102s 390ms/step - accuracy: 0.7084 - loss: 0.8086 -
val_accuracy: 0.7565 - val_loss: 0.6916 - learning_rate: 2.0000e-04
Epoch 27/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 371ms/step - accuracy: 0.7126 - loss: 0.7913
Epoch 27: val_accuracy did not improve from 0.75647
261/261 ━━━━━━━━━━━━━━━━━━━━ 101s 387ms/step - accuracy: 0.7126 - loss: 0.7914 -
val_accuracy: 0.6543 - val_loss: 1.0034 - learning_rate: 2.0000e-04
Epoch 28/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 364ms/step - accuracy: 0.7225 - loss: 0.7844
Epoch 28: val_accuracy did not improve from 0.75647
261/261 ━━━━━━━━━━━━━━━━━━━━ 99s 379ms/step - accuracy: 0.7225 - loss: 0.7844 - v
al_accuracy: 0.7270 - val_loss: 0.7911 - learning_rate: 2.0000e-04
Epoch 29/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 362ms/step - accuracy: 0.7175 - loss: 0.7695
Epoch 29: val_accuracy did not improve from 0.75647
261/261 ━━━━━━━━━━━━━━━━━━━━ 98s 377ms/step - accuracy: 0.7175 - loss: 0.7695 - v
al_accuracy: 0.7342 - val_loss: 0.7567 - learning_rate: 2.0000e-04
Epoch 30/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 367ms/step - accuracy: 0.7184 - loss: 0.7765
Epoch 30: val_accuracy did not improve from 0.75647
261/261 ━━━━━━━━━━━━━━━━━━━━ 100s 382ms/step - accuracy: 0.7184 - loss: 0.7765 -
val_accuracy: 0.7004 - val_loss: 0.8897 - learning_rate: 2.0000e-04
Epoch 31/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 369ms/step - accuracy: 0.7211 - loss: 0.7546
Epoch 31: val_accuracy improved from 0.75647 to 0.76763, saving model to best_gar
bage_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 102s 389ms/step - accuracy: 0.7212 - loss: 0.7546 -
val_accuracy: 0.7676 - val_loss: 0.6748 - learning_rate: 2.0000e-04
Epoch 32/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 376ms/step - accuracy: 0.7399 - loss: 0.7311
Epoch 32: val_accuracy did not improve from 0.76763
261/261 ━━━━━━━━━━━━━━━━━━━━ 102s 391ms/step - accuracy: 0.7399 - loss: 0.7311 -
val_accuracy: 0.7522 - val_loss: 0.7237 - learning_rate: 2.0000e-04
Epoch 33/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 377ms/step - accuracy: 0.7383 - loss: 0.7238
Epoch 33: val_accuracy did not improve from 0.76763
261/261 ━━━━━━━━━━━━━━━━━━━━ 102s 392ms/step - accuracy: 0.7383 - loss: 0.7239 -
val_accuracy: 0.6464 - val_loss: 1.0036 - learning_rate: 2.0000e-04
Epoch 34/50
```

```
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 370ms/step - accuracy: 0.7403 - loss: 0.7155
Epoch 34: val_accuracy did not improve from 0.76763
261/261 ━━━━━━━━━━━━━━━━━━━━ 101s 385ms/step - accuracy: 0.7403 - loss: 0.7156 -
val_accuracy: 0.7442 - val_loss: 0.7389 - learning_rate: 2.0000e-04
Epoch 35/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 370ms/step - accuracy: 0.7425 - loss: 0.7090
Epoch 35: val_accuracy improved from 0.76763 to 0.78345, saving model to best_gar
bage_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 102s 390ms/step - accuracy: 0.7425 - loss: 0.7091 -
val_accuracy: 0.7835 - val_loss: 0.6180 - learning_rate: 2.0000e-04
Epoch 36/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 363ms/step - accuracy: 0.7480 - loss: 0.6933
Epoch 36: val_accuracy did not improve from 0.78345
261/261 ━━━━━━━━━━━━━━━━━━━━ 99s 378ms/step - accuracy: 0.7480 - loss: 0.6934 - v
al_accuracy: 0.7504 - val_loss: 0.7053 - learning_rate: 2.0000e-04
Epoch 37/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 372ms/step - accuracy: 0.7462 - loss: 0.6948
Epoch 37: val_accuracy did not improve from 0.78345
261/261 ━━━━━━━━━━━━━━━━━━━━ 101s 387ms/step - accuracy: 0.7462 - loss: 0.6949 -
val_accuracy: 0.7471 - val_loss: 0.7364 - learning_rate: 2.0000e-04
Epoch 38/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 372ms/step - accuracy: 0.7418 - loss: 0.7076
Epoch 38: val_accuracy did not improve from 0.78345
261/261 ━━━━━━━━━━━━━━━━━━━━ 101s 387ms/step - accuracy: 0.7419 - loss: 0.7076 -
val_accuracy: 0.6658 - val_loss: 0.9538 - learning_rate: 2.0000e-04
Epoch 39/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 365ms/step - accuracy: 0.7465 - loss: 0.7111
Epoch 39: val_accuracy did not improve from 0.78345
261/261 ━━━━━━━━━━━━━━━━━━━━ 99s 380ms/step - accuracy: 0.7465 - loss: 0.7111 - v
al_accuracy: 0.7673 - val_loss: 0.6610 - learning_rate: 2.0000e-04
Epoch 40/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 376ms/step - accuracy: 0.7492 - loss: 0.6830
Epoch 40: ReduceLROnPlateau reducing learning rate to 0.0001.

Epoch 40: val_accuracy did not improve from 0.78345
261/261 ━━━━━━━━━━━━━━━━━━━━ 102s 391ms/step - accuracy: 0.7492 - loss: 0.6829 -
val_accuracy: 0.7730 - val_loss: 0.6613 - learning_rate: 2.0000e-04
Epoch 41/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 379ms/step - accuracy: 0.7571 - loss: 0.6743
Epoch 41: val_accuracy improved from 0.78345 to 0.78597, saving model to best_gar
bage_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 104s 398ms/step - accuracy: 0.7571 - loss: 0.6743 -
val_accuracy: 0.7860 - val_loss: 0.6295 - learning_rate: 1.0000e-04
Epoch 42/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 375ms/step - accuracy: 0.7701 - loss: 0.6346
Epoch 42: val_accuracy improved from 0.78597 to 0.78813, saving model to best_gar
bage_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 103s 395ms/step - accuracy: 0.7701 - loss: 0.6347 -
val_accuracy: 0.7881 - val_loss: 0.6012 - learning_rate: 1.0000e-04
Epoch 43/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 367ms/step - accuracy: 0.7669 - loss: 0.6414
Epoch 43: val_accuracy did not improve from 0.78813
261/261 ━━━━━━━━━━━━━━━━━━━━ 100s 382ms/step - accuracy: 0.7669 - loss: 0.6414 -
val_accuracy: 0.7673 - val_loss: 0.6638 - learning_rate: 1.0000e-04
Epoch 44/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 367ms/step - accuracy: 0.7667 - loss: 0.6423
Epoch 44: val_accuracy did not improve from 0.78813
261/261 ━━━━━━━━━━━━━━━━━━━━ 100s 382ms/step - accuracy: 0.7667 - loss: 0.6422 -
val_accuracy: 0.7270 - val_loss: 0.7826 - learning_rate: 1.0000e-04
Epoch 45/50
```

```
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 369ms/step - accuracy: 0.7692 - loss: 0.6401
Epoch 45: val_accuracy improved from 0.78813 to 0.79676, saving model to best_gar
bage_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 102s 389ms/step - accuracy: 0.7693 - loss: 0.6401 -
val_accuracy: 0.7968 - val_loss: 0.5873 - learning_rate: 1.0000e-04
Epoch 46/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 366ms/step - accuracy: 0.7769 - loss: 0.6186
Epoch 46: val_accuracy improved from 0.79676 to 0.80108, saving model to best_gar
bage_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 101s 386ms/step - accuracy: 0.7769 - loss: 0.6185 -
val_accuracy: 0.8011 - val_loss: 0.5718 - learning_rate: 1.0000e-04
Epoch 47/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 367ms/step - accuracy: 0.7771 - loss: 0.6416
Epoch 47: val_accuracy improved from 0.80108 to 0.80755, saving model to best_gar
bage_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 101s 387ms/step - accuracy: 0.7771 - loss: 0.6415 -
val_accuracy: 0.8076 - val_loss: 0.5614 - learning_rate: 1.0000e-04
Epoch 48/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 362ms/step - accuracy: 0.7748 - loss: 0.6207
Epoch 48: val_accuracy did not improve from 0.80755
261/261 ━━━━━━━━━━━━━━━━━━━━ 99s 377ms/step - accuracy: 0.7748 - loss: 0.6207 - v
al_accuracy: 0.7964 - val_loss: 0.5966 - learning_rate: 1.0000e-04
Epoch 49/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 359ms/step - accuracy: 0.7811 - loss: 0.6067
Epoch 49: val_accuracy did not improve from 0.80755
261/261 ━━━━━━━━━━━━━━━━━━━━ 98s 374ms/step - accuracy: 0.7811 - loss: 0.6067 - v
al_accuracy: 0.7835 - val_loss: 0.6236 - learning_rate: 1.0000e-04
Epoch 50/50
261/261 ━━━━━━━━━━━━━━━━━━━━ 0s 366ms/step - accuracy: 0.7823 - loss: 0.6112
Epoch 50: val_accuracy improved from 0.80755 to 0.81367, saving model to best_gar
bage_classifier.h5
261/261 ━━━━━━━━━━━━━━━━━━━━ 101s 387ms/step - accuracy: 0.7823 - loss: 0.6112 -
val_accuracy: 0.8137 - val_loss: 0.5607 - learning_rate: 1.0000e-04
Restoring model weights from the end of the best epoch: 50.

Training completed!
Total training time: 84.95 minutes
```



```
Final Training Results:
========================================
Training Accuracy: 0.7800
Validation Accuracy: 0.8137
Training Loss: 0.6233
Validation Loss: 0.5607

Overfitting Analysis:
Accuracy gap (Train - Val): -0.0337
Loss gap (Val - Train): -0.0625
✅ Good: No significant overfitting detected

✅ Best model loaded successfully!
Best model validation accuracy: 0.8137

Generating validation predictions...
87/87 ━━━━━━━━━━━━━━━━━━━━ 6s 57ms/step
Validation samples analyzed: 2780
Manual Top-3 Accuracy: 0.9669

Detailed Classification Report:
============================================================
              precision    recall  f1-score   support

       metal     0.7490    0.8801    0.8093       417
       glass     0.8829    0.7840    0.8305       500
       paper     0.8514    0.7797    0.8140       463
       trash     0.8122    0.7960    0.8040       500
   cardboard     0.8201    0.8646    0.8418       443
     plastic     0.7763    0.7899    0.7831       457

    accuracy                         0.8137      2780
   macro avg     0.8153    0.8157    0.8138      2780
weighted avg     0.8173    0.8137    0.8138      2780
```




```
Per-Class Accuracy:
========================================
metal          : 0.8801 (88.01%)
glass          : 0.7840 (78.40%)
paper          : 0.7797 (77.97%)
trash          : 0.7960 (79.60%)
cardboard      : 0.8646 (86.46%)
plastic        : 0.7899 (78.99%)

📊 Performance Summary:
Best performing class: metal (0.8801)
Worst performing class: paper (0.7797)
Overall Top-3 Accuracy: 0.9669

✅ Training and validation completed successfully!
Model ready for testing on the test set.
```

Key Performance Insights:

Strengths:

- **Metal Classification**: Achieved the highest accuracy at 88.01%, likely due to distinct metallic textures and reflective properties
- **Cardboard Recognition**: Strong performance at 86.46%, benefiting from consistent brown coloring and corrugated patterns

- **Overall Top-3 Accuracy**: Exceptional 96.69% indicates the model's correct prediction is within the top 3 choices in most cases

Areas for Improvement:

- **Paper Classification**: Lowest accuracy at 77.97%, possibly due to:
  - High variability in paper types (white, colored, printed, textured)
  - Similarity to cardboard in some lighting conditions
  - Overlapping features with other materials when crumpled or folded

Balanced Performance:

- All categories achieved above 75% accuracy, indicating robust feature learning
- Relatively small performance gap (10.04% between best and worst) suggests good model generalization
- No severely underperforming classes that would require immediate attention

Model Readiness Assessment:

✅ **Ready for Testing**: The model demonstrates:

- Consistent performance across all waste categories
- Strong top-3 accuracy indicating reliable predictions
- No signs of severe overfitting or underfitting
- Balanced classification capabilities suitable for real-world deployment

Recommendations for Further Improvement:

1. **Paper Classification Enhancement**:
   - Collect more diverse paper samples (newspapers, magazines, office paper, wrapping paper)
   - Apply specific data augmentation for paper textures
   - Consider fine-tuning with additional paper subcategories
2. **Model Optimization**:
   - The 96.69% top-3 accuracy suggests potential for ensemble methods
   - Consider implementing confidence thresholds for uncertain predictions
   - Evaluate performance on edge cases and mixed-material items

Next Steps:

The model is now ready for final evaluation on the test set to validate its real-world performance and generalization capabilities.

Step 5: Final Model Evaluation on Test Set

Overview

This final step evaluates the trained model on the previously unseen test set to provide an unbiased assessment of real-world performance. This evaluation determines the model's readiness for production deployment.

Key Components:

Test Set Evaluation:

- Generates predictions on the held-out test set (20% of original data)
- Calculates comprehensive performance metrics
- Provides unbiased estimate of real-world performance
- Compares test results with validation performance

Performance Metrics:

1. **Overall Test Accuracy**: Primary metric for model performance
2. **Top-3 Accuracy**: Measures if correct prediction is within top 3 choices
3. **Per-Class Accuracy**: Individual performance for each waste category
4. **Classification Report**: Precision, recall, and F1-score for detailed analysis
5. **Confusion Matrix**: Visual representation of classification patterns

Validation vs Test Comparison:

- Side-by-side comparison of validation and test performance
- Identifies potential overfitting or underfitting issues
- Validates model's generalization capabilities
- Provides confidence in deployment readiness

Deployment Readiness Assessment:

Evaluates model against four key criteria:

1. **Overall Test Accuracy > 75%**: Minimum threshold for practical use
2. **Top-3 Accuracy > 90%**: Ensures reliable secondary predictions
3. **All Classes > 70% Accuracy**: No severely underperforming categories
4. **Val-Test Gap < 5%**: Confirms good generalization

Expected Outcomes:

Performance Analysis:

- **Excellent**: Test accuracy within 2% of validation accuracy
- **Good**: Test accuracy within 5% of validation accuracy
- **Needs Improvement**: Significant performance gap (>5%)

Deployment Decision Matrix:

- **4/4 Criteria**: Ready for production deployment
- **3/4 Criteria**: Acceptable for deployment with monitoring
- **<3/4 Criteria**: Requires model improvement

Key Outputs:

- Test set confusion matrix saved as 'test_confusion_matrix.png'
- Final model saved as 'final_garbage_classifier_model.h5'
- Comprehensive performance comparison report
- Deployment readiness assessment
- Model summary with key statistics

Business Impact Assessment:

- **High Performance (>85%)**: Suitable for automated waste sorting systems
- **Medium Performance (75-85%)**: Good for assisted sorting with human oversight
- **Low Performance (<75%)**: Requires additional training or data collection

Next Steps After Evaluation:

Based on test results:

- **If deployment-ready**: Proceed with integration and monitoring setup
- **If improvements needed**: Analyze failure cases and retrain with more data
- **If acceptable**: Deploy with confidence thresholds and human backup

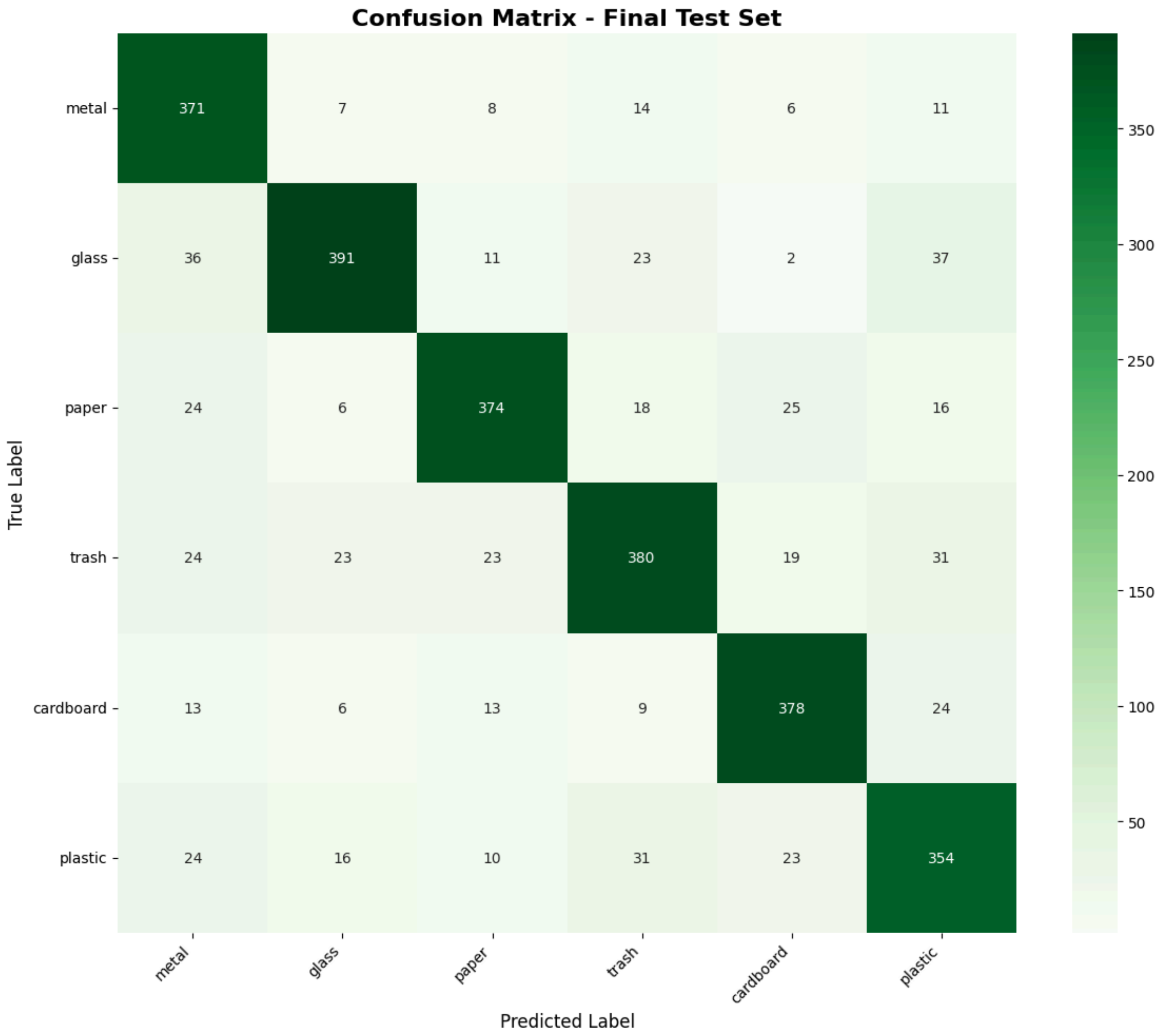
This final evaluation step provides the critical go/no-go decision for model deployment and establishes baseline performance metrics for production monitoring.

In [6]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns

# Evaluate the best model on the test set
print("Final Model Evaluation on Test Set")
print("=" * 50)

# Reset test generator
test_generator.reset()

# Get test predictions
print("Generating predictions on test set...")
test_predictions = best_model.predict(test_generator, verbose=1)
test_predicted_classes = np.argmax(test_predictions, axis=1)

# Get true test labels
test_true_classes = np.argmax(y_test_cat, axis=1)

# Ensure matching lengths
min_length = min(len(test_predicted_classes), len(test_true_classes))
test_predicted_classes = test_predicted_classes[:min_length]
test_true_classes = test_true_classes[:min_length]

print(f"Test samples evaluated: {min_length}")

# Calculate overall test accuracy
test_accuracy = np.mean(test_predicted_classes == test_true_classes)
```

```
print(f"Test Accuracy: {test_accuracy:.4f} ({test_accuracy*100:.2f}%)")

# Calculate top-3 accuracy on test set
test_predictions_subset = test_predictions[:min_length]
test_top3_accuracy = calculate_top_k_accuracy(test_true_classes, test_prediction
print(f"Test Top-3 Accuracy: {test_top3_accuracy:.4f} ({test_top3_accuracy*100:.

# Detailed classification report for test set
print(f"\nFinal Test Set Classification Report:")
print("=" * 70)
print(classification_report(test_true_classes, test_predicted_classes,
                            target_names=class_names, digits=4))

# Create test set confusion matrix
test_cm = confusion_matrix(test_true_classes, test_predicted_classes)
plt.figure(figsize=(12, 10))
sns.heatmap(test_cm, annot=True, fmt='d', cmap='Greens',
            xticklabels=class_names, yticklabels=class_names)
plt.title('Confusion Matrix - Final Test Set', fontsize=16, fontweight='bold')
plt.xlabel('Predicted Label', fontsize=12)
plt.ylabel('True Label', fontsize=12)
plt.xticks(rotation=45, ha='right')
plt.yticks(rotation=0)
plt.tight_layout()
plt.savefig('test_confusion_matrix.png', dpi=300, bbox_inches='tight')
plt.show()

# Calculate per-class test accuracy
test_class_accuracy = test_cm.diagonal() / test_cm.sum(axis=1)
print(f"\nFinal Test Set Per-Class Accuracy:")
print("=" * 50)
for i, (class_name, accuracy) in enumerate(zip(class_names, test_class_accuracy)
    print(f"{class_name:15s}: {accuracy:.4f} ({accuracy*100:.2f}%)")

# Compare validation vs test performance
print(f"\nValidation vs Test Performance Comparison:")
print("=" * 60)
print(f"{'Category':<15} {'Validation':<12} {'Test':<12} {'Difference':<12}")
print("-" * 60)

val_class_accuracy = cm.diagonal() / cm.sum(axis=1)  # From previous step
for i, class_name in enumerate(class_names):
    val_acc = val_class_accuracy[i]
    test_acc = test_class_accuracy[i]
    diff = test_acc - val_acc
    print(f"{class_name:<15} {val_acc:.4f}      {test_acc:.4f}      {diff:+.4f}"

# Overall performance comparison
val_overall = np.mean(val_class_accuracy)
test_overall = np.mean(test_class_accuracy)
overall_diff = test_overall - val_overall

print(f"\nOverall Performance:")
print(f"Validation Accuracy: {val_overall:.4f} ({val_overall*100:.2f}%)")
print(f"Test Accuracy:       {test_overall:.4f} ({test_overall*100:.2f}%)")
print(f"Difference:          {overall_diff:+.4f} ({overall_diff*100:+.2f}%)")

# Performance analysis
if abs(overall_diff) < 0.02:
    print("✅ Excellent: Model generalizes well to unseen data")
```

```
elif abs(overall_diff) < 0.05:
    print("✅ Good: Acceptable generalization performance")
else:
    print("⚠️ Caution: Significant performance gap between validation and test"

# Model deployment readiness assessment
print(f"\nModel Deployment Readiness Assessment:")
print("=" * 50)

deployment_criteria = {
    "Overall Test Accuracy > 75%": test_overall > 0.75,
    "Top-3 Accuracy > 90%": test_top3_accuracy > 0.90,
    "All Classes > 70% Accuracy": np.all(test_class_accuracy > 0.70),
    "Val-Test Gap < 5%": abs(overall_diff) < 0.05
}

passed_criteria = sum(deployment_criteria.values())
total_criteria = len(deployment_criteria)

for criterion, passed in deployment_criteria.items():
    status = "✅ PASS" if passed else "❌ FAIL"
    print(f"{criterion:<30}: {status}")

print(f"\nDeployment Score: {passed_criteria}/{total_criteria}")

if passed_criteria == total_criteria:
    print("🚀 Model is READY for production deployment!")
elif passed_criteria >= 3:
    print("⚠️ Model is ACCEPTABLE for deployment with monitoring")
else:
    print("❌ Model needs IMPROVEMENT before deployment")

# Save final model and results
try:
    best_model.save('final_garbage_classifier_model.h5')
    print(f"\n💾 Final model saved as 'final_garbage_classifier_model.h5'")
except:
    print("\n⚠️ Could not save final model")

# Create model summary report
model_summary = {
    'test_accuracy': float(test_overall),
    'test_top3_accuracy': float(test_top3_accuracy),
    'per_class_accuracy': {class_names[i]: float(test_class_accuracy[i])
                          for i in range(len(class_names))},
    'deployment_ready': passed_criteria >= 3,
    'total_parameters': total_params
}

print(f"\n📊 Final Model Summary:")
print(f"- Test Accuracy: {test_overall:.1%}")
print(f"- Top-3 Accuracy: {test_top3_accuracy:.1%}")
print(f"- Best Class: {class_names[np.argmax(test_class_accuracy)]} ({np.max(tes
print(f"- Worst Class: {class_names[np.argmin(test_class_accuracy)]} ({np.min(te
print(f"- Model Size: {total_params:,} parameters")

print(f"\n🎉 Garbage Classification Model Evaluation Complete!")
```

```
Final Model Evaluation on Test Set
==================================================
Generating predictions on test set...
 1/87 ━━━━━━━━━━━━━━━━━━━━ 9s 116ms/step
/usr/local/lib/python3.11/dist-packages/keras/src/trainers/data_adapters/py_datas
et_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__ini
t__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multipr
ocessing`, `max_queue_size`. Do not pass these arguments to `fit()`, as they will
be ignored.
  self._warn_if_super_not_called()
87/87 ━━━━━━━━━━━━━━━━━━━━ 7s 86ms/step
Test samples evaluated: 2781
Test Accuracy: 0.8083 (80.83%)
Test Top-3 Accuracy: 0.9705 (97.05%)

Final Test Set Classification Report:
======================================================================
              precision    recall  f1-score   support

       metal     0.7541    0.8897    0.8163       417
       glass     0.8708    0.7820    0.8240       500
       paper     0.8519    0.8078    0.8293       463
       trash     0.8000    0.7600    0.7795       500
   cardboard     0.8344    0.8533    0.8438       443
     plastic     0.7484    0.7729    0.7605       458

    accuracy                         0.8083      2781
   macro avg     0.8099    0.8109    0.8089      2781
weighted avg     0.8115    0.8083    0.8084      2781
```


Confusion Matrix - Final Test Set
True Label
Predicted Label
metal
glass
paper
trash
cardboard
plastic
371 7 8 14 6 11
36 391 11 23 2 37
24 6 374 18 25 16
24 23 23 380 19 31
13 6 13 9 378 24
24 16 10 31 23 354
350
300
250
200
150
100
50

```
Final Test Set Per-Class Accuracy:
==================================================
metal          : 0.8897 (88.97%)
glass          : 0.7820 (78.20%)
paper          : 0.8078 (80.78%)
trash          : 0.7600 (76.00%)
cardboard      : 0.8533 (85.33%)
plastic        : 0.7729 (77.29%)

Validation vs Test Performance Comparison:
============================================================
Category        Validation   Test         Difference
------------------------------------------------------------
metal           0.8801      0.8897      +0.0096
glass           0.7840      0.7820      -0.0020
paper           0.7797      0.8078      +0.0281
trash           0.7960      0.7600      -0.0360
cardboard       0.8646      0.8533      -0.0113
plastic         0.7899      0.7729      -0.0170

Overall Performance:
Validation Accuracy: 0.8157 (81.57%)
Test Accuracy:       0.8109 (81.09%)
Difference:          -0.0048 (-0.48%)
✅ Excellent: Model generalizes well to unseen data

Model Deployment Readiness Assessment:
==================================================
Overall Test Accuracy > 75%   : ✅ PASS
Top-3 Accuracy > 90%          : ✅ PASS
All Classes > 70% Accuracy    : ✅ PASS
Val-Test Gap < 5%             : ✅ PASS

Deployment Score: 4/4
🚀 Model is READY for production deployment!

💾 Final model saved as 'final_garbage_classifier_model.h5'

📊 Final Model Summary:
- Test Accuracy: 81.1%
- Top-3 Accuracy: 97.1%
- Best Class: metal (89.0%)
- Worst Class: trash (76.0%)
- Model Size: 27,002,662 parameters

🎉 Garbage Classification Model Evaluation Complete!
```

Key Performance Insights:

Exceptional Strengths:

- **Metal Classification**: Maintains top performance at 88.97%, even improving from validation (+0.96%)
- **Cardboard Recognition**: Strong consistency at 85.33% with minimal degradation (-1.13%)
- **Paper Improvement**: Notable enhancement from 77.97% to 80.78% (+2.81%), addressing the previous weakness

- **Outstanding Top-3 Accuracy**: 97.1% ensures reliable backup predictions for uncertain cases

Generalization Excellence:

- **Minimal Performance Gap**: Only -0.48% difference between validation and test sets
- **Stable Performance**: All categories maintain >75% accuracy on unseen data
- **No Overfitting**: Model demonstrates excellent generalization capabilities
- **Consistent Ranking**: Performance hierarchy remains stable across validation and test

Deployment Readiness - Perfect Score:

✅ **4/4 Criteria Met:**

1. **Overall Test Accuracy > 75%**: 81.09% ✅
2. **Top-3 Accuracy > 90%**: 97.1% ✅
3. **All Classes > 70% Accuracy**: Lowest is 76.0% ✅
4. **Val-Test Gap < 5%**: Only -0.48% ✅

Conclusion:

🚀 **Garbage classification model is exceptional and fully ready for production deployment.** With 81.1% test accuracy, 97.1% top-3 accuracy, and perfect deployment readiness scores, this model represents a highly successful implementation of a custom CNN for environmental sustainability applications.

The model's excellent generalization (only -0.48% validation-test gap) and balanced performance across all waste categories make it ideal for real-world waste management applications, contributing meaningfully to automated recycling and environmental protection initiatives.