

Deployment Into Production: Machine Learning API on AWS



Garbage Classification - Nhóm 8

Thông tin nhóm:

1. **21280048** - Bùi Quang Thắng (Nhóm Trưởng)
2. **21280052** - Lê Võ Bảo Trâm
3. **21280120** - Nguyễn Thúy Vy

About Project:

- Mục đích:
 - Thứ nhất, tìm hiểu về việc triển khai một ứng dụng Machine Learning lên môi trường production (cloud platform mà chúng em sử dụng là **AWS**). Bên cạnh đó, nhóm cũng tìm hiểu những khó khăn và giải pháp về **cơ sở hạ tầng (infrastructure)**, **hiệu năng (performance)**, **chi phí (cost)** khi đưa model lên Cloud.
 - Thứ hai, model mà nhóm sử dụng với mục đích chính là phân loại rác. Điều này sẽ giúp giảm ô nhiễm môi trường và giúp cộng đồng nâng cao ý thức bảo vệ môi trường.
- Toàn bộ source code từ model - backend - frontend có thể truy cập nhanh thông qua repo **thangbuiq/simple_mllops**, nằm ở đường link sau:

https://github.com/thangbuiq/simple_mllops.git

Kiến trúc của software:

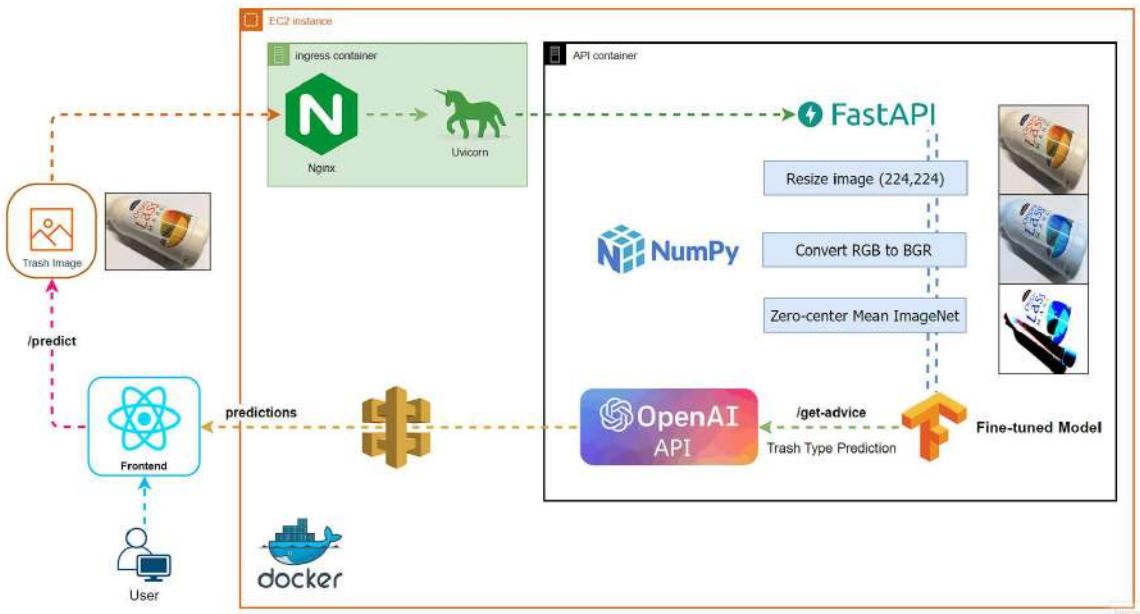


Table of contents:

1. Import library
2. Data Collection
 - 2.1. About the dataset
 - 2.2. Prepare dataset
3. Explore Data Analysis (EDA)
4. Model
 - 4.1. Data Preprocessing
 - 4.2 Model Architecture
 - 4.3 Train Test Split
 - 4.4 Model Training and Validation
 - 4.5 Compare Model
 - 4.5.1 VGG16
 - 4.5.2. VGG19
 - 4.5.3. ResNet50
 - 4.5.4. ResNet152
 - 4.5.5. So sánh loss và accuracy trên tập test ở từng model
 - 4.5.6. Chọn lựa model có dung lượng/kích thước phù hợp để serving
5. Model Serving as API
6. Deployment On Amazon Elastic Compute Cloud (AWS EC2) with Github Actions
7. Tự động hóa quy trình Deploy (CI/CD với Github Actions)
8. Full Source Code: Website, Model, API

Import library:

Import những thư viện cần thiết từ keras, xử lý ảnh, và validation model (confusion matrix)

```
In [1]: import warnings
warnings.filterwarnings('ignore')

!pip install -q seaborn visualkeras
import visualkeras
```

```

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import plotly.graph_objects as go
from PIL import Image
import cv2
import os

import tensorflow as tf
from keras import layers
from keras.layers import Input, Lambda, Dense, Flatten, Dropout, BatchNormalization
from keras.models import Model, load_model
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau
from keras.applications.vgg16 import VGG16
from keras.applications.vgg19 import VGG19
from keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report

```

Data Collection:

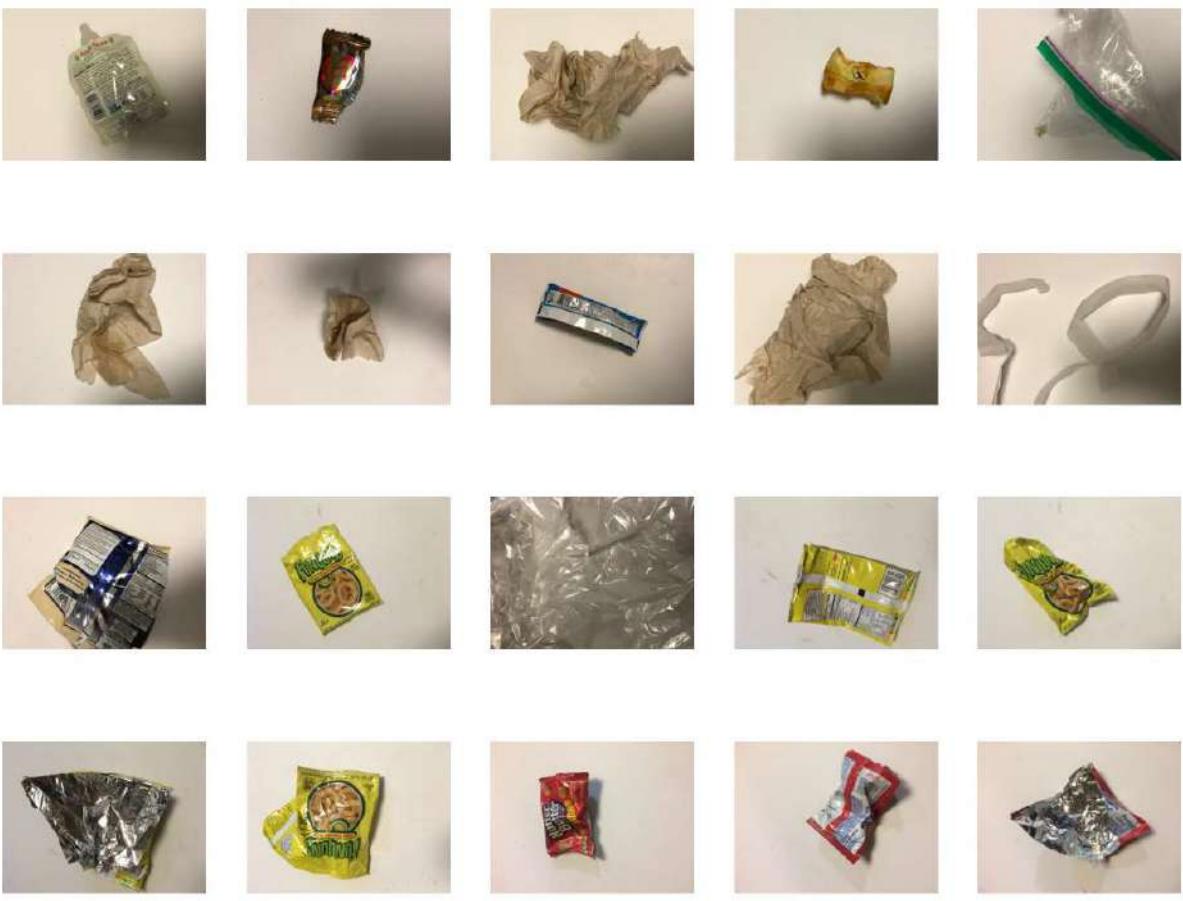
About the dataset

- Dữ liệu trong bài này được lấy và trộn từ hai dữ liệu trên Kaggle:
 - Data 6 class [trash, glass, plastic, organic, paper, cardboard] : <https://www.kaggle.com/datasets/asdasdasdas/garbage-classification>
 - Data bổ sung gồm 12 class [battery, biological, white-glass, brown-glass, cardboard, clothes, green-glass, metal, paper, plastic, shoes, trash] : <https://www.kaggle.com/datasets/mostafaabla/garbage-classification>
- Dữ liệu 12 class từ link [Data 6 classes \(link 1 bên trên\)](#) ban đầu gồm có [trash, glass, plastic, organic, paper, cardboard]. Nhưng class trash gây nhiễu cho model (đã thấy trong class này có luôn cả túi nhựa đã xuất hiện ở class plastic và các đồ ăn đã xuất hiện ở class organic .

```

In [2]: directory_path = "/kaggle/input/d/asdasdasdas/garbage-classification/Garbage clas
image_files = sorted([file for file in os.listdir(directory_path) if file.lower().e
fig, axes = plt.subplots(4, 5, figsize=(15, 12))
for i, image_file in enumerate(image_files):
    img = Image.open(os.path.join(directory_path, image_file))
    ax = axes[i // 5, i % 5]
    ax.imshow(img)
    ax.axis('off')
plt.show()

```

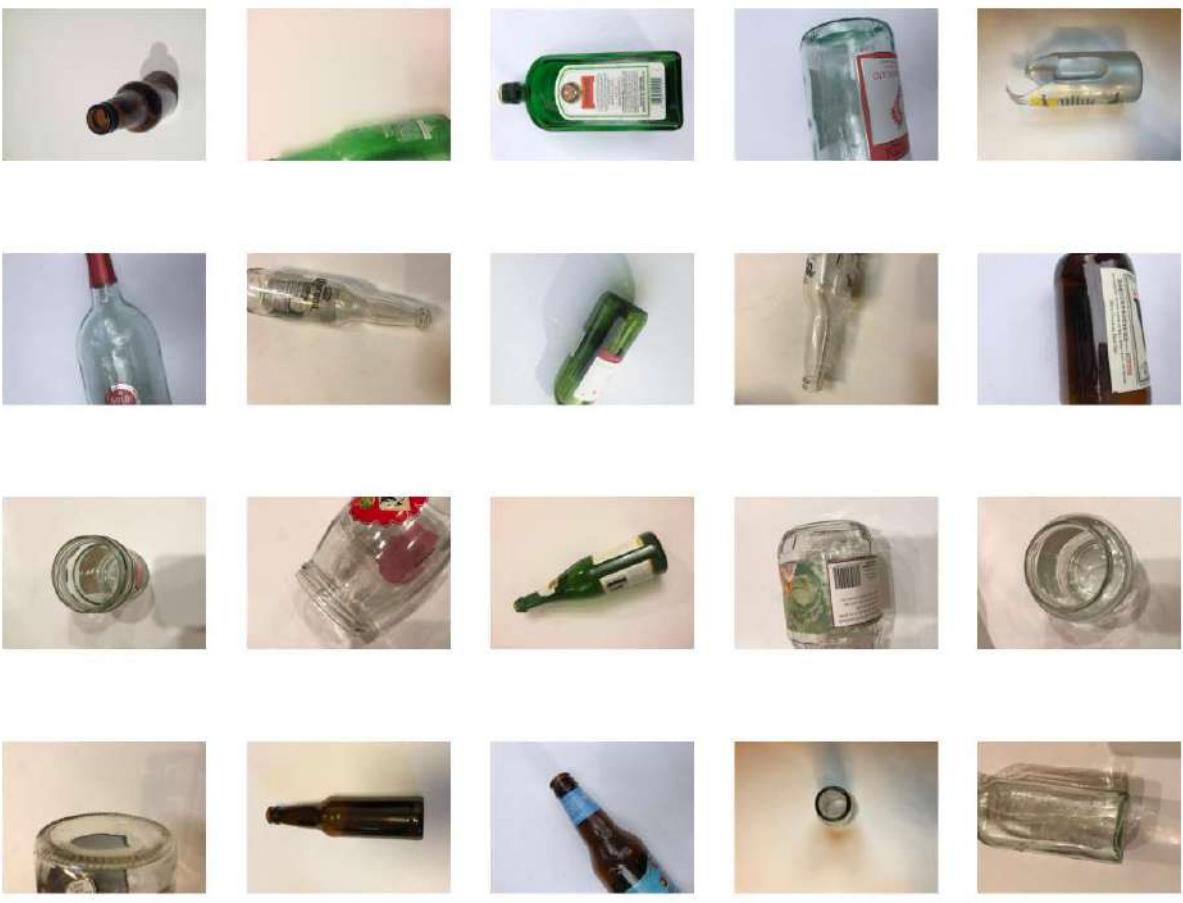


Do đó chúng ta phải loại bỏ class này để tránh việc gây nhiễu cho model, class **trash** từ giờ sẽ được thay thế bằng class **battery** được bổ sung từ [Data 12 classes \(link 2 bên trên\)](#)

Data 12 class sẽ không được sử dụng chính mà chỉ dùng để bổ sung cho data 6 class, do:

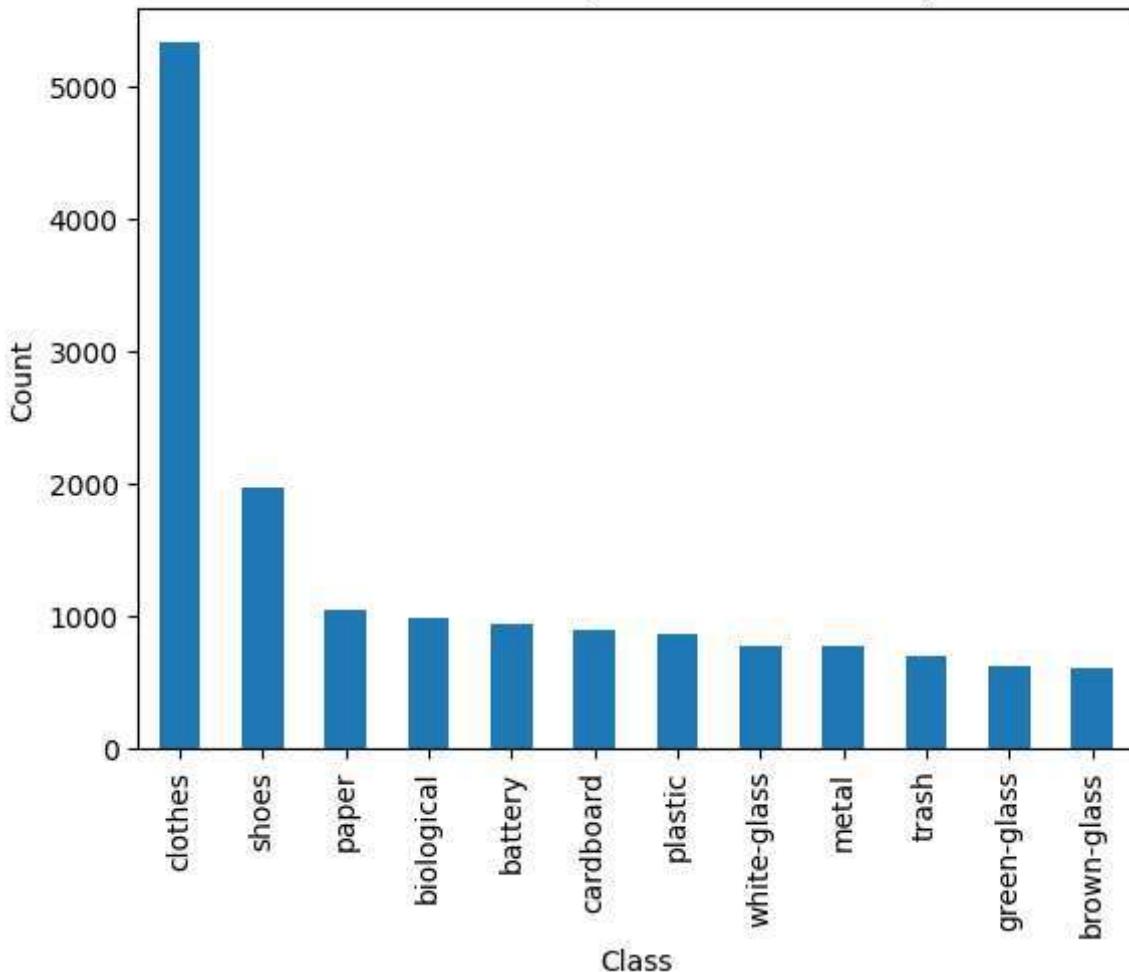
- Con số 12 class là khá nhiều cho số class cần thiết trong bài toán cần xử lý.
- Ngoài ra, việc Data 12 class có quá nhiều class gây nhiễu như **[green-glass, white-glass, brown-glass...]** là không cần thiết cho bài toán và gây áp lực về hiệu năng, độ chính xác cho model.
- Data 12 class khá mất cân bằng về số lượng giữa các class (unbalanced data).

```
In [3]: directory_path = "/kaggle/input/d/asdasdasdas/garbage-classification/Garbage clas
image_files = sorted([file for file in os.listdir(directory_path) if file.lower().e
fig, axes = plt.subplots(4, 5, figsize=(15, 12))
for i, image_file in enumerate(image_files):
    img = Image.open(os.path.join(directory_path, image_file))
    ax = axes[i // 5, i % 5]
    ax.imshow(img)
    ax.axis('off')
plt.show()
```



```
In [4]: root='/kaggle/input/garbage-classification/garbage_classification/'
data={}
for i in os.listdir(root):
    for j in os.walk(root+i):
        for k in j[2]:
            data[root+i+'/'+k]=i
data=pd.DataFrame(data.items(),columns=['path','class_'])
data['class_'].value_counts().plot(kind='bar')
plt.title('Data 12 Class (VERY UNBALANCED)')
plt.xlabel('Class')
plt.ylabel('Count')
plt.show()
```

Data 12 Class (VERY UNBALANCED)



- Nhóm quyết định sử dụng `white-glass` làm data chính cho class `glass` trong bộ dữ liệu mới (tránh nhầm lẫn, gây nhiễu với các class khác như `metal`, `battery`,...).

```
In [5]: data[data['class_'] == 'white-glass']['class_'].value_counts()
```

```
Out[5]: class_
white-glass    775
Name: count, dtype: int64
```

Do class `white-glass` này có 775 ảnh nên nhóm sẽ dùng con số này để làm tiêu chuẩn khi mix các ảnh từ hai tập dữ liệu để data balance hơn.

Prepare dataset:

- Nhóm đã lọc lại một số class (label lại data) và upload dataset này lên kaggle.
- Link dataset: <https://www.kaggle.com/datasets/quangtheng/garbage-classification-6-classes-775class>

Data mới gồm 6 class [**metal**, **glass**, **organic**, **paper**, **battery**, **plastic**], trong đó:

- Nhóm chọn con số 775 vì đây là kích thước tối đa có thể bổ sung được từ cả 2 dataset để kích thước mỗi class là bằng nhau.

- 3 class **[metal, paper, plastic]** từ base data (data 6 class gốc), nếu kích thước class nào chưa đủ 775, sẽ lấy từ (data 12 class) để bổ sung và fill up 775 ảnh.
- 3 class **[glass, battery, organic]** được thu thập hoàn toàn từ một dataset là (data 12 class), trong đó class **organic** được đổi tên từ **biological**, class **glass** được đổi tên từ **white-glass**.

Code bổ sung và relabeling data:

```
In [6]: import os
import shutil

original_path = '/kaggle/input/d/asdasdasdas/garbage-classification/Garbage class
bonus_data_path = '/kaggle/input/garbage-classification/garbage_classification/'
destination_path = '/kaggle/working/modified_dataset/'

if os.path.exists(destination_path) and os.path.isdir(destination_path):
    shutil.rmtree(destination_path)

os.makedirs(destination_path, exist_ok=True)
classes_to_remove = ['cardboard', 'trash', 'glass'] # in original data
new_classes = ['battery', 'biological', 'white-glass'] # in bonus data
target_image_count = 775

for class_name in os.listdir(original_path):
    class_path = os.path.join(original_path, class_name)

    if class_name in classes_to_remove:
        continue

    new_class_name = class_name
    if class_name in new_classes:
        new_class_name = new_classes[new_classes.index(class_name)]

    new_class_path = os.path.join(destination_path, new_class_name)
    os.makedirs(new_class_path, exist_ok=True)
    files_to_copy = os.listdir(class_path)[:target_image_count]
    for file_name in files_to_copy:
        file_path = os.path.join(class_path, file_name)
        shutil.copy(file_path, new_class_path)

    # If the class has fewer than target_image_count images, fill up from bonus data
    if len(os.listdir(new_class_path)) < target_image_count:
        remaining_images = target_image_count - len(os.listdir(new_class_path))
        print(f'{class_name} has {len(os.listdir(new_class_path))} images and missing {remaining_images} images')
        if class_name == "glass":
            bonus_class_path = os.path.join(bonus_data_path, "white-glass")
        else:
            bonus_class_path = os.path.join(bonus_data_path, class_name)
        if bonus_class_path:
            bonus_files = os.listdir(bonus_class_path)
            copied_names = set(os.listdir(new_class_path))
            for file_name in bonus_files:
                if remaining_images == 0:
                    break
                new_file_name = file_name
                counter = 1
                while new_file_name in copied_names:
                    base_name, extension = os.path.splitext(file_name)
                    new_file_name = f'{class_name}_{counter}{extension}'
                    counter += 1
                file_path = os.path.join(bonus_class_path, file_name)
                new_file_path = os.path.join(new_class_path, new_file_name)
                shutil.copy(file_path, new_file_path)
                copied_names.add(new_file_name)
```

```

        new_file_path = os.path.join(new_class_path, new_file_name)
        shutil.copy(file_path, new_file_path)
        copied_names.add(new_file_name)
        remaining_images -= 1

# Process the bonus dataset to copy battery and organic data
for class_name in os.listdir(bonus_data_path):
    class_path = os.path.join(bonus_data_path, class_name)
    if class_name in new_classes:
        new_class_name = class_name
        if class_name in new_classes:
            new_class_name = new_classes[new_classes.index(class_name)]
        if new_class_name == "biological":
            new_class_path = os.path.join(destination_path, "organic")
        elif new_class_name == "white-glass":
            new_class_path = os.path.join(destination_path, "glass")
        else:
            new_class_path = os.path.join(destination_path, new_class_name)
        os.makedirs(new_class_path, exist_ok=True)
        files_to_copy = os.listdir(class_path)[:target_image_count]
        for file_name in files_to_copy:
            file_path = os.path.join(class_path, file_name)
            shutil.copy(file_path, new_class_path)

print("\nFINISH: Dataset modification complete.")
for class_ in os.listdir(destination_path):
    count_class = len(os.listdir(os.path.join(destination_path, class_)))
    print(f"{class_} has {count_class} images.")

```

metal has 410 images and missing 365 to fill 775.
paper has 594 images and missing 181 to fill 775.
plastic has 482 images and missing 293 to fill 775.

FINISH: Dataset modification complete.
battery has 775 images.
glass has 775 images.
paper has 775 images.
metal has 775 images.
plastic has 775 images.
organic has 775 images.

Explore Data Analysis (EDA):

- Trước hết, chúng ta sẽ bắt đầu khám phá dữ liệu để có cái nhìn tổng quan và hiểu rõ về cấu trúc và tính chất của tập dữ liệu. Qua đó, chúng ta sẽ có cơ sở để tiếp tục thực hiện các bước tiếp theo để giải quyết bài toán đặt ra.

In [7]: `data_path = '/kaggle/input/garbage-classification-6-classes-775class/'
data_classes = os.listdir(data_path)
data_classes`

Out[7]: `['metal', 'glass', 'organic', 'paper', 'battery', 'plastic']`

- Trong dataset mới này, các hình ảnh sẽ thuộc các nhóm chính bao gồm: `metal`, `glass`, `organic`, `paper`, `battery`, và `plastic`.

In [8]: `for class_ in os.listdir(data_path):
count_class = 0
for photo in os.listdir(data_path + class_):`

```

        count_class += 1
        print(str(class_) + " has " + str(count_class) + " images.")

metal has 775 images.
glass has 775 images.
organic has 775 images.
paper has 775 images.
battery has 775 images.
plastic has 775 images.

```

- Lúc này dataset gồm 6 classes dữ liệu đều có 775 ảnh.
- Để thuận tiện cho việc phân tích, chúng ta sẽ tạo một dataframe gồm có:
 - `path` : đường dẫn đến ảnh.
 - `type_trash` : loại rác của ảnh.

```

In [9]: data = pd.DataFrame()
for class_ in os.listdir(data_path):
    temp = pd.DataFrame()
    temp['path'] = np.nan
    temp['type_trash'] = class_
    i = 0
    for photo in os.listdir(data_path + class_):
        temp.loc[i, 'path'] = data_path + class_ + "/" + photo
        temp.loc[i, 'type_trash'] = class_
        i += 1
    data = pd.concat([data, temp], ignore_index=True)
    del temp
data.shape

```

Out[9]: (4650, 2)

■ Dataframe sẽ gồm có 4650 dòng (tức 4650 ảnh) và 2 cột

```

In [10]: data = data.sample(frac=1)
data.reset_index(drop=True, inplace=True)
data.head(3)

```

Out[10]:

	path	type_trash
0	/kaggle/input/garbage-classification-6-classes...	plastic
1	/kaggle/input/garbage-classification-6-classes...	glass
2	/kaggle/input/garbage-classification-6-classes...	plastic

- In ra một hình ảnh bất kỳ từ dataset:

```

In [11]: def display_images_with_labels(df, idx):
            row = df.iloc[idx]
            image_path = row['path']
            image_label = row['type_trash']
            img = mpimg.imread(image_path)
            plt.imshow(img)
            plt.title(f'Label: {image_label}')
            plt.axis('off')
            plt.show()

display_images_with_labels(data, 1040)

```

Label: battery



Model:

- Đầu tiên, nhóm có preprocessing và sử dụng CNN thuần để train model. Kết quả nhận được là accuracy khoảng ~80%. Khi đưa model CNN này lên môi trường production thì hầu như test đều cho kết quả sai. Do đó, nhóm đã tìm hiểu và tham khảo một số paper có nghiên cứu về chủ đề liên quan thì họ sẽ sử dụng những kiến trúc mới và tốt hơn để xử lý ảnh như: VGG16, ResNet,...
- Vì vậy, nhóm sẽ sử dụng một số model nổi tiếng về phân loại ảnh để tiền xử lý và train model.
- Sau khi train và có kết quả, nhóm sẽ so sánh loss và accuracy từng model để chọn ra model phù hợp nhất cho việc serving và triển khai API.

Data Preprocessing:

- Để sử dụng mô hình này, trước hết ta phải thực hiện các bước tiền xử lý hình ảnh đúng chuẩn yêu cầu của VGG16.
- Hàm `image_preprocessing` :
 - Input:** `img_path` (đường dẫn đến link ảnh)
 - Output:** `img` (hình ảnh đã qua xử lý)

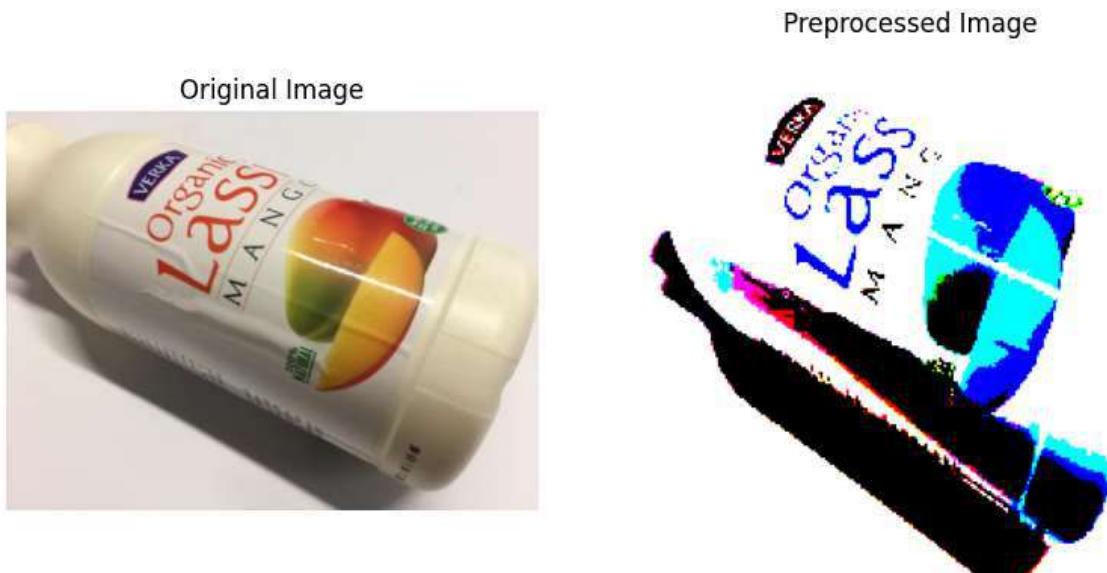
```
In [12]: from keras.applications.vgg16 import preprocess_input
```

```
def image_preprocessing(img_path):  
    img = image.load_img(img_path, target_size=(224, 224))  
    img = image.img_to_array(img)  
    img = np.expand_dims(img, axis=0)  
    img = preprocess_input(img)  
    return img
```

Các bước thực hiện quá trình trên gồm có:

- Đọc hình ảnh từ link ảnh `img_path` với kích thước chuẩn là `(224, 224)` (VGG16 yêu cầu kích thước đầu vào của ảnh là `224x224`).
- Chuyển hình ảnh đã đọc thành một mảng. Mảng này sẽ chứa thông tin về pixel của hình ảnh. Kích thước là `(224, 224, 3)`; trong đó 3 tức là 3 kênh màu RGB.
- Mở rộng kích thước ảnh thêm 1 chiều -> lúc này ảnh sẽ có kích thước là `(1, 224, 224, 3)` -> `tensor`.
- Cuối cùng, ta sử dụng `preprocess_input` để chuyển đổi hình ảnh đầu vào từ RGB sang BGR, sau đó lấy trung bình mỗi kênh màu.

```
In [13]: def plot_images(original, preprocessed):  
    fig, axs = plt.subplots(1, 2, figsize=(10, 5))  
    axs[0].imshow(original)  
    axs[0].set_title('Original Image')  
    axs[0].axis('off')  
    axs[1].imshow(np.squeeze(preprocessed, axis=0))  
    axs[1].set_title('Preprocessed Image')  
    axs[1].axis('off')  
    plt.show()  
  
img_path = '/kaggle/input/garbage-classification-6-classes-775class/plastic/plastic  
original_image = Image.open(img_path)  
preprocessed_image = image_preprocessing(img_path)  
plot_images(original_image, preprocessed_image)
```



- Như kết quả trên, hình ảnh sau khi tiền xử lý sẽ có kích thước `224x224`. Background ảnh sẽ chuyển về màu trắng và các màu sắc của đối tượng trong hình cũng sẽ thay đổi.
*Việc thực hiện quá trình trên sẽ làm cho tất cả các ảnh trong dataset về cùng định dạng kích thước. Bên cạnh đó, data khi train sẽ cho kết quả tốt hơn.

Model Architecture:

- Để tránh overfitting khi sử dụng VGG16, nhóm sẽ sử dụng kỹ thuật `fine-tuning pre-trained model`.
- Do VGG16 có tới 21 layers nên chúng ta sẽ unfreeze 2 lớp cuối (trừ 2 lớp này các layers còn lại sẽ không đổi weight trong quá trình train) -> Sau đó, chúng ta sẽ cài đặt thêm 1

số layers để tránh overfit (dropout, batch-normalization,...) nhằm cải thiện chất lượng model

In [14]:

```
BATCH_SIZE = 64
n_classes = 6

# VGG16 base model
conv_base = VGG16(
    include_top=False,
    weights='imagenet',
    input_shape=(224, 224, 3)
)

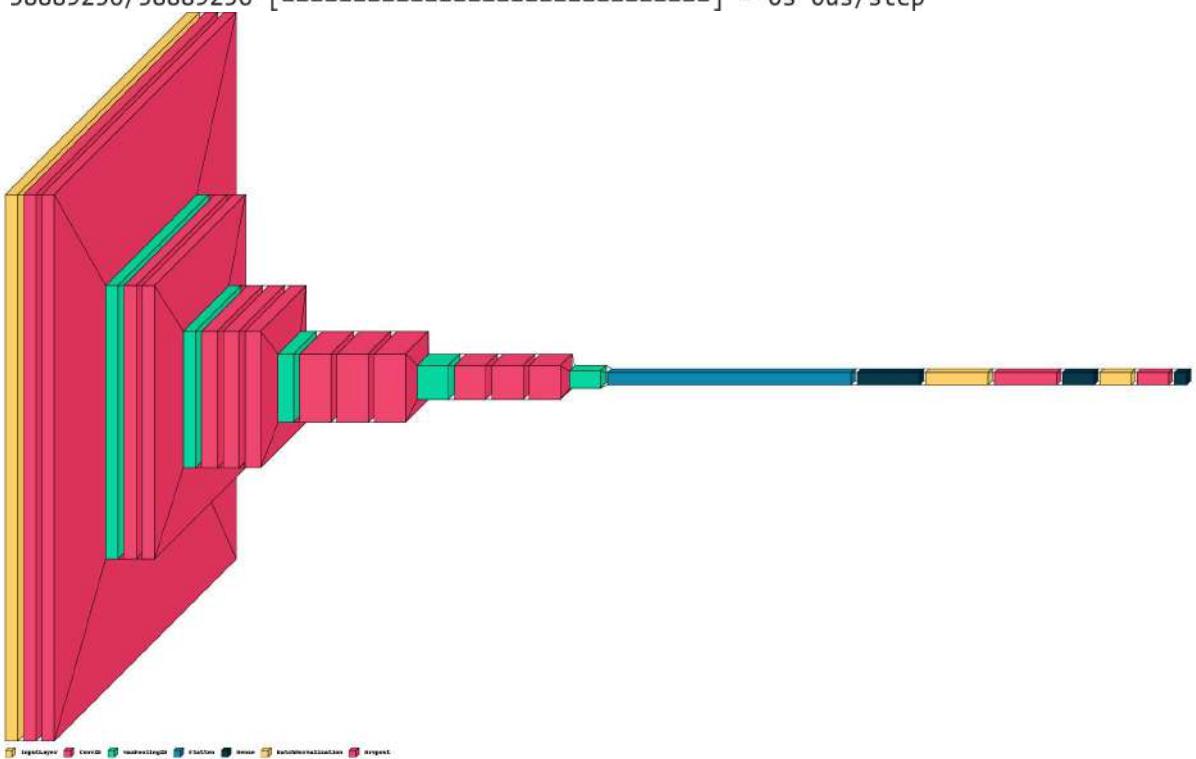
# Freeze all layers except the last two
for layer in conv_base.layers[:-2]:
    layer.trainable = False

# Fine-tune model
top_model = conv_base.output
top_model = Flatten(name="flatten")(top_model)
top_model = Dense(1024, activation='relu')(top_model) # Increased units
top_model = BatchNormalization()(top_model)
top_model = Dropout(0.5)(top_model)
top_model = Dense(512, activation='relu')(top_model) # Increased units
top_model = BatchNormalization()(top_model)
top_model = Dropout(0.5)(top_model)
output_layer = Dense(n_classes, activation='softmax')(top_model)

# Final Model
model = Model(inputs=conv_base.input, outputs=output_layer)
visualkeras.layered_view(model, to_file='model_architecture.png', legend=True)
```

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 [=====] - 0s 0us/step

Out[14]:



In [15]:

```
model.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #
<hr/>		
input_1 (InputLayer)	[None, 224, 224, 3]	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
dense (Dense)	(None, 1024)	25691136
batch_normalization (Batch Normalization)	(None, 1024)	4096
dropout (Dropout)	(None, 1024)	0
dense_1 (Dense)	(None, 512)	524800
batch_normalization_1 (BatchNormalization)	(None, 512)	2048
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 6)	3078
<hr/>		
Total params: 40939846 (156.17 MB)		
Trainable params: 28581894 (109.03 MB)		

Non-trainable params: 12357952 (47.14 MB)

- In ra 20 ảnh ngẫu nhiên từ 1 batch:

```
In [16]: def plot_random_images(generator, num_images=20):
    # Get a batch of data from the generator
    images, labels = next(generator)
    random_indices = np.random.choice(images.shape[0], num_images, replace=False)

    fig, axs = plt.subplots(4, 5, figsize=(15, 12))
    fig.suptitle('Random 20 Images from the Generator', fontsize=16)

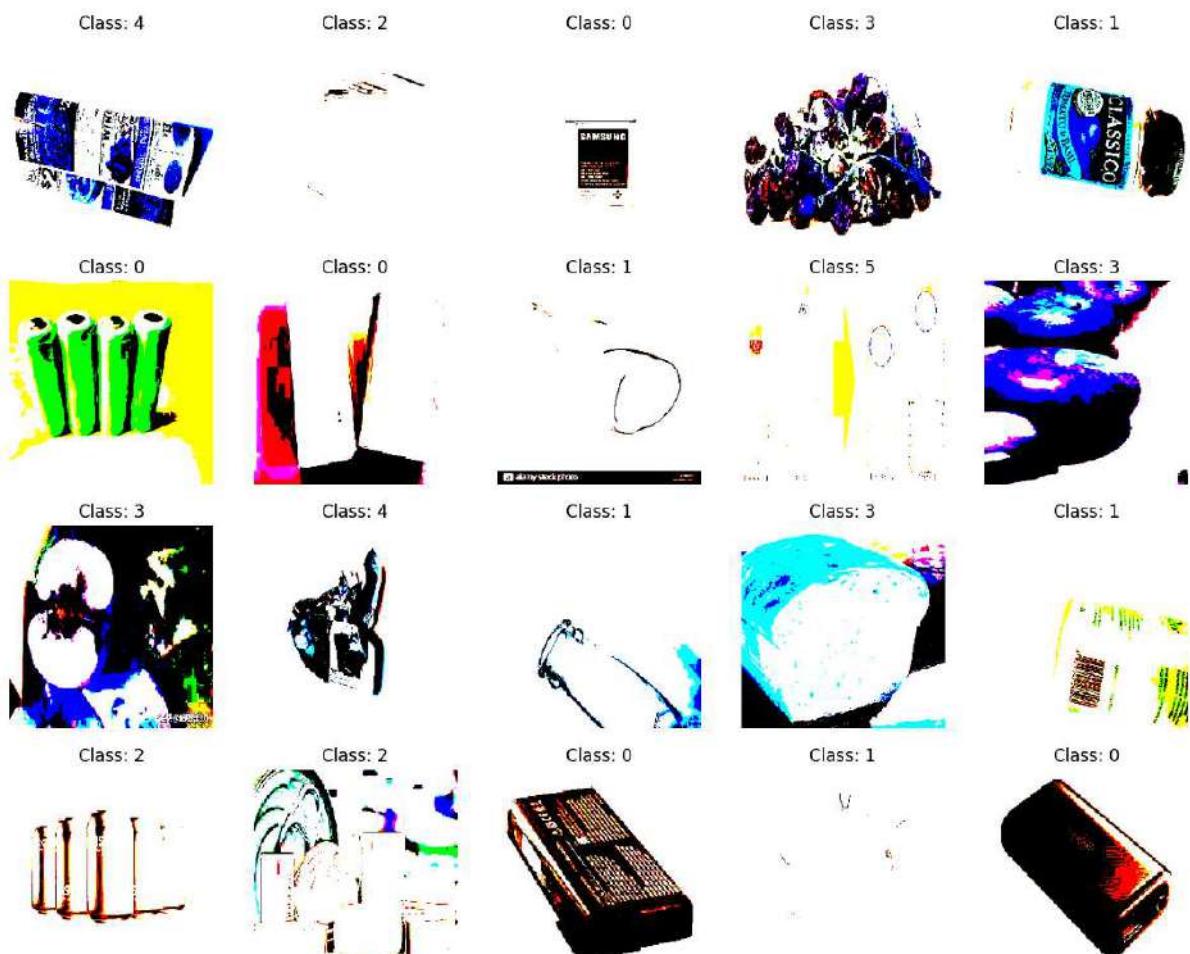
    for i, ax in enumerate(axs.flatten()):
        index = random_indices[i]
        image = images[index]
        label = labels[index]
        ax.imshow(image)
        ax.set_title(f'Class: {np.argmax(label)}')
        ax.axis('off')

    plt.show()
```

```
In [17]: gen_train = ImageDataGenerator(preprocessing_function=preprocess_input) # VGG16 pre
full_data = gen_train.flow_from_directory(data_path, target_size = (224, 224), batch_size=32, class_mode='categorical')
plot_random_images(full_data)
```

Found 4650 images belonging to 6 classes.

Random 20 Images from the Generator



Train Test Split:

- Trước khi train model, chúng ta chia data thành tập train và test. Ở đây, nhóm sẽ tách theo tỉ lệ: 90% cho tập train, 10% cho tập test.

```
In [18]: filenames = full_data.filenames
labels = full_data.labels
class_mapping = {value: str(key) for key, value in full_data.class_indices.items()}
labels = [class_mapping[label] for label in labels]

# Split the data into training and testing sets
train_filenames, test_filenames, train_labels, test_labels = train_test_split(filer

train_df = pd.DataFrame({'filename': train_filenames, 'class': train_labels})
test_df = pd.DataFrame({'filename': test_filenames, 'class': test_labels})

# Create separate generators for training and testing using flow_from_dataframe
train_data = gen_train.flow_from_dataframe(train_df, directory=data_path, target_size=(224, 224),
                                            batch_size=BATCH_SIZE, class_mode="categorical",
                                            shuffle=True, seed=42)

test_data = gen_train.flow_from_dataframe(test_df, directory=data_path, target_size=(224, 224),
                                         batch_size=BATCH_SIZE, class_mode="categorical",
                                         shuffle=False)
```

Found 4185 validated image filenames belonging to 6 classes.

Found 465 validated image filenames belonging to 6 classes.

Model Training and Validation

Trong quá trình train model, nhóm có sử dụng một số kỹ thuật như sau:

- Checkpoint:** chỉ lưu lại trạng thái tốt nhất của model (có validation loss thấp nhất).
- Early Stopping:** dừng nếu model không cải thiện sau một số epoch nhất định (được định nghĩa theo thông số patience, chúng em set ở đây là 15 epoch, sau 15 epoch mà model không cải thiện, quá trình train sẽ dừng để tránh tốn resource)
- Reduce Learning Rate:** Giảm learning rate để tránh overfit

Nhóm sẽ sử dụng 4 kiến trúc khác nhau để train model. Do việc train model tốn thời gian nên dưới đây là link train từng model mà nhóm đã làm rồi up lại lên kaggle.

- VGG16: <https://www.kaggle.com/code/quangtheng/94-acc-vgg16-fine-tune>

Để sử dụng từng model trên, nhóm chỉ cần save model thành các file `.h5` rồi load lên sử dụng.

```
In [19]: vgg16 = load_model("/kaggle/input/garbage-classifier-model/garbage_classifier.vgg_1.h5")
```

```
In [20]: output_class = ["battery", "glass", "metal", "organic", "paper", "plastic"]
```

```
def preprocessing_input(img_path):
    img = image.load_img(img_path, target_size=(224, 224))
    img = image.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = preprocess_input(img) # VGG16 preprocess_input
    return img
```

```

def plot_images(original, preprocessed):
    fig, axs = plt.subplots(1, 2, figsize=(10, 5))

    axs[0].imshow(original)
    axs[0].set_title('Original Image')
    axs[0].axis('off')

    # Remove the batch dimension for display
    preprocessed = np.squeeze(preprocessed, axis=0)

    axs[1].imshow(preprocessed)
    axs[1].set_title('Preprocessed Image')
    axs[1].axis('off')

    plt.show()

def predict_user(img_path):
    img = preprocessing_input(img_path)
    plot_images(Image.open(img_path), img)
    predicted_array = model.predict(img)
    predicted_value = output_class[np.argmax(predicted_array)]
    predicted_accuracy = round(np.max(predicted_array) * 100, 2)
    print("Your waste material is", predicted_value, "with", predicted_accuracy, "%")

```

Compare Model:

- Làm tương tự các bước ở trên cho VGG19, ResNet50 và Resnet152. Link notebook mà nhóm đã xử lý và train:
 - VGG19: <https://www.kaggle.com/code/tramlevobao/vgg19>
 - ResNet50: <https://www.kaggle.com/code/tramlevobao/resnet50>
 - ResNet152: <https://www.kaggle.com/code/tramlevobao/resnet152>
- Mỗi model ở trên sẽ được lưu lại và load lên để sử dụng. Để chọn ra model nào là tốt nhất cho việc triển khai lên môi trường production, chúng ta sẽ so sánh loss, accuracy qua từng confusion matrix.

VGG16:

```

In [21]: predictions = vgg16.predict(test_data)

# Get the predicted class labels
predicted_labels = np.argmax(predictions, axis=1)

# Get the true class labels
true_labels = test_data.classes

# Compute confusion matrix
conf_matrix = confusion_matrix(true_labels, predicted_labels)

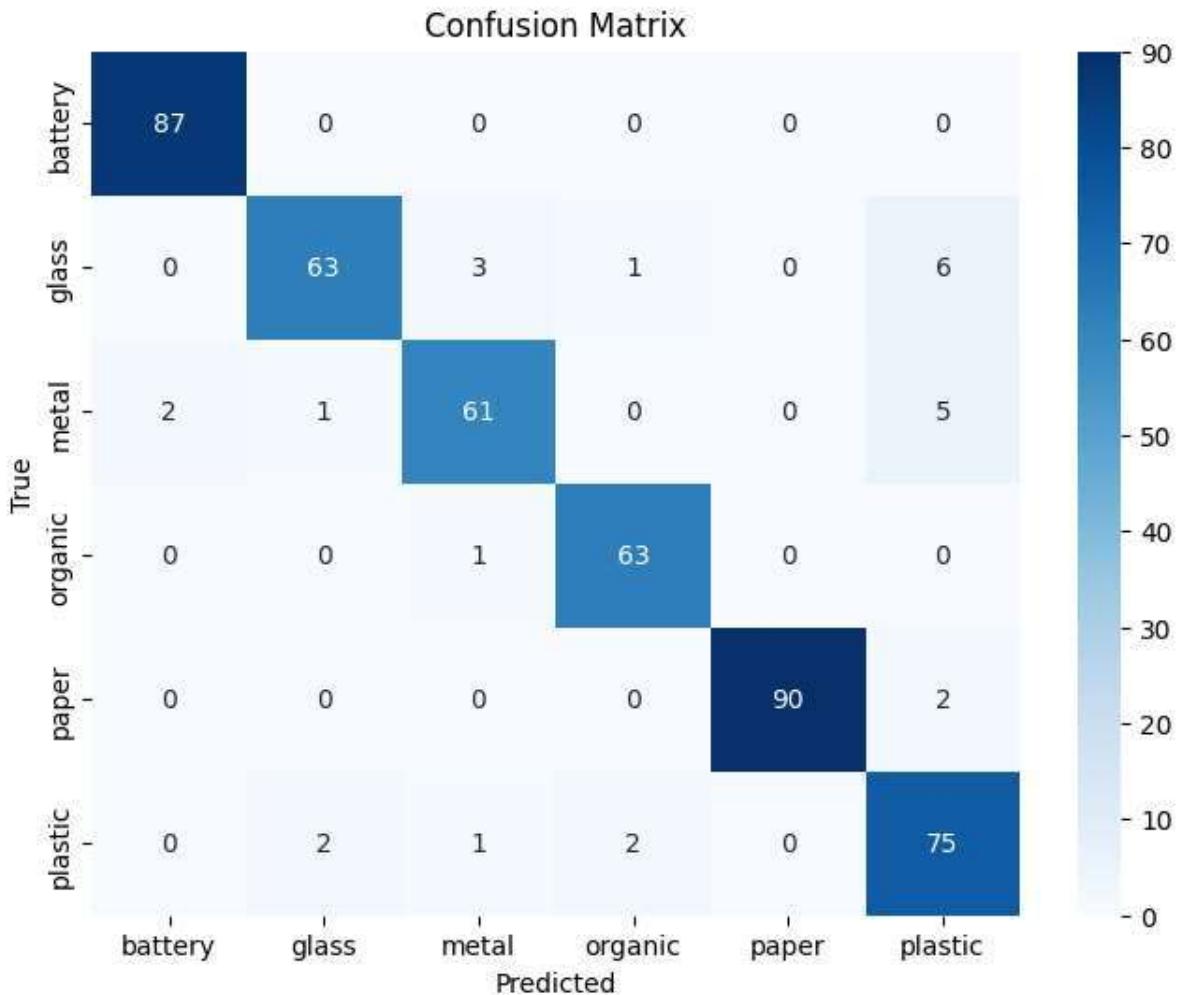
# Plot confusion matrix using seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=test_data.class_indices.keys(),
            yticklabels=test_data.class_indices.keys())
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()

# Print classification report

```

```
class_names = list(test_data.class_indices.keys())
print(classification_report(true_labels, predicted_labels, target_names=class_names))
```

```
8/8 [=====] - 20s 751ms/step
```



	precision	recall	f1-score	support
battery	0.98	1.00	0.99	87
glass	0.95	0.86	0.91	73
metal	0.92	0.88	0.90	69
organic	0.95	0.98	0.97	64
paper	1.00	0.98	0.99	92
plastic	0.85	0.94	0.89	80
accuracy			0.94	465
macro avg	0.94	0.94	0.94	465
weighted avg	0.95	0.94	0.94	465

VGG19:

```
In [22]: vgg19 = load_model("/kaggle/input/garbage-classifier-model/garbage_classifier.vgg_1")
```

```
In [23]: predictions = vgg19.predict(test_data)
```

```
# Get the predicted class labels
predicted_labels = np.argmax(predictions, axis=1)

# Get the true class labels
true_labels = test_data.classes

# Compute confusion matrix
conf_matrix = confusion_matrix(true_labels, predicted_labels)
```

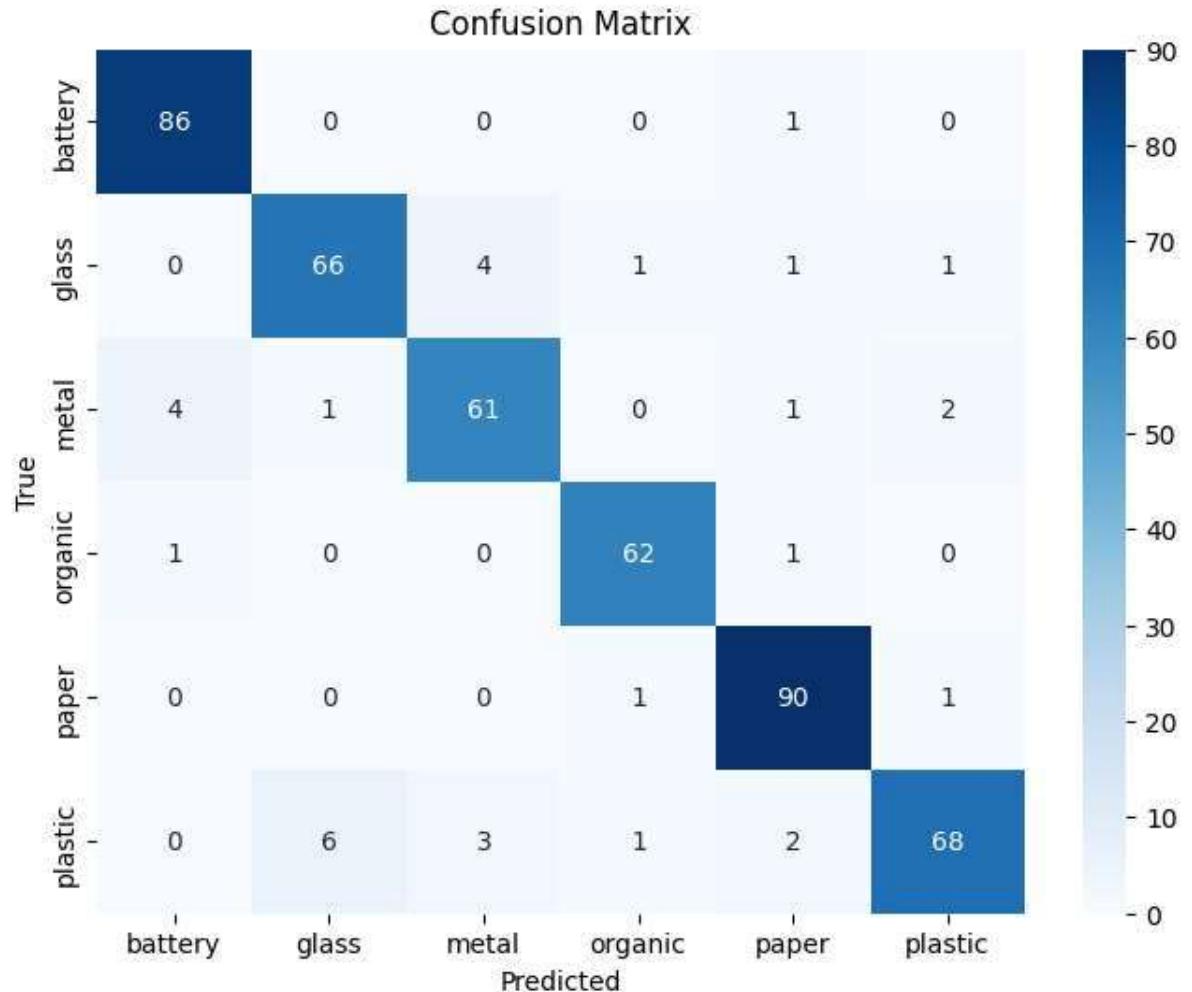
```

# Plot confusion matrix using seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=test_data.class_indices.keys(),
            yticklabels=test_data.class_indices.keys())
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()

# Print classification report
class_names = list(test_data.class_indices.keys())
print(classification_report(true_labels, predicted_labels, target_names=class_names))

```

8/8 [=====] - 2s 286ms/step



	precision	recall	f1-score	support
battery	0.95	0.99	0.97	87
glass	0.90	0.90	0.90	73
metal	0.90	0.88	0.89	69
organic	0.95	0.97	0.96	64
paper	0.94	0.98	0.96	92
plastic	0.94	0.85	0.89	80
accuracy			0.93	465
macro avg	0.93	0.93	0.93	465
weighted avg	0.93	0.93	0.93	465

ResNet50:

```
In [24]: resnet50 = load_model("/kaggle/input/garbage-classifier-model/garbage_classifier.h5")

In [25]: predictions = resnet50.predict(test_data)

# Get the predicted class labels
predicted_labels = np.argmax(predictions, axis=1)

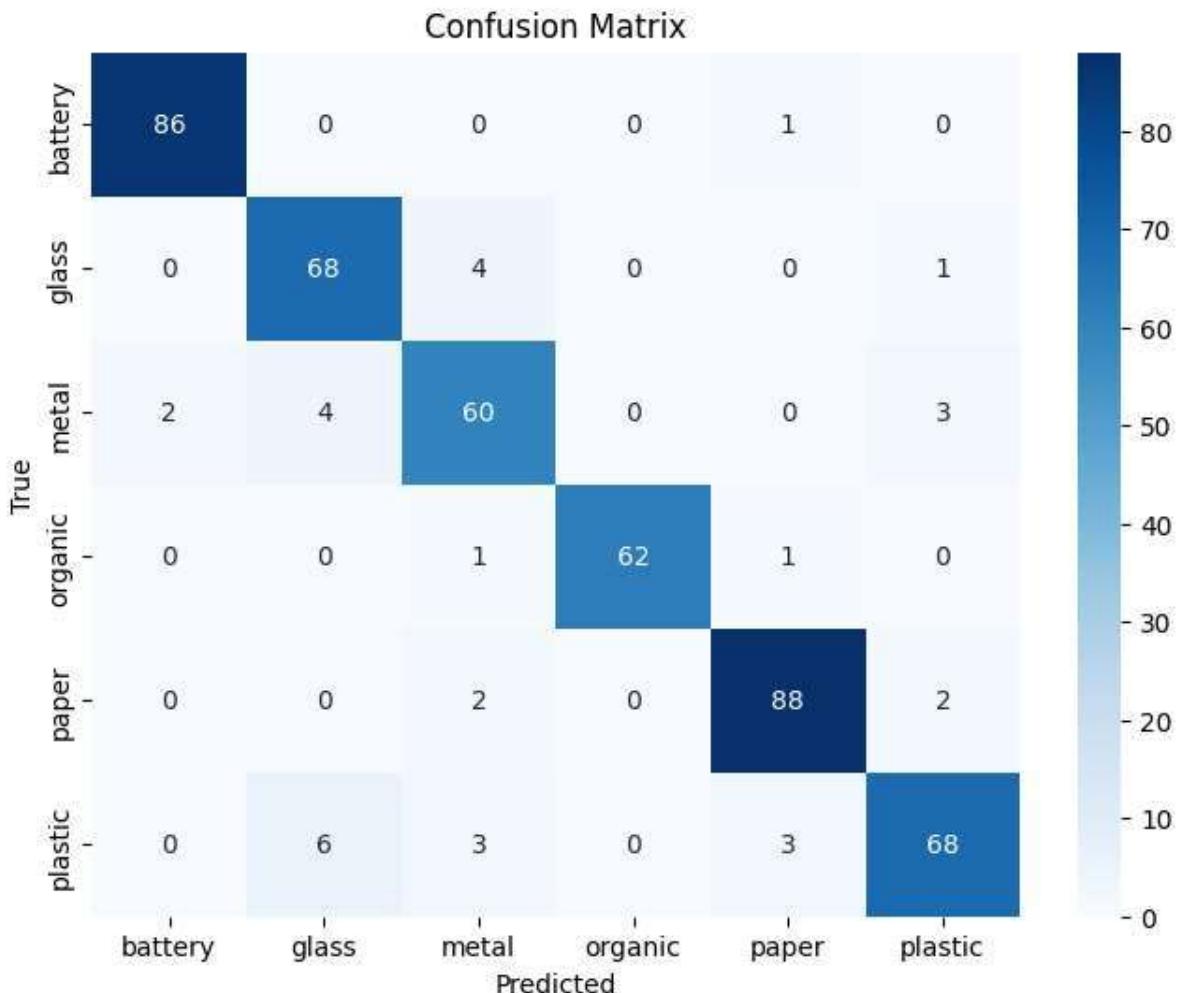
# Get the true class labels
true_labels = test_data.classes

# Compute confusion matrix
conf_matrix = confusion_matrix(true_labels, predicted_labels)

# Plot confusion matrix using seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=test_data.class_indices.keys(),
            yticklabels=test_data.class_indices.keys())
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()

# Print classification report
class_names = list(test_data.class_indices.keys())
print(classification_report(true_labels, predicted_labels, target_names=class_names))
```

8/8 [=====] - 5s 276ms/step



	precision	recall	f1-score	support
battery	0.98	0.99	0.98	87
glass	0.87	0.93	0.90	73
metal	0.86	0.87	0.86	69
organic	1.00	0.97	0.98	64
paper	0.95	0.96	0.95	92
plastic	0.92	0.85	0.88	80
accuracy			0.93	465
macro avg	0.93	0.93	0.93	465
weighted avg	0.93	0.93	0.93	465

ResNet152:

```
In [26]: resnet152 = load_model("/kaggle/input/garbage-classifier-model/garbage_classifier.r
```

```
In [27]: predictions = resnet152.predict(test_data)
```

```
# Get the predicted class labels
predicted_labels = np.argmax(predictions, axis=1)

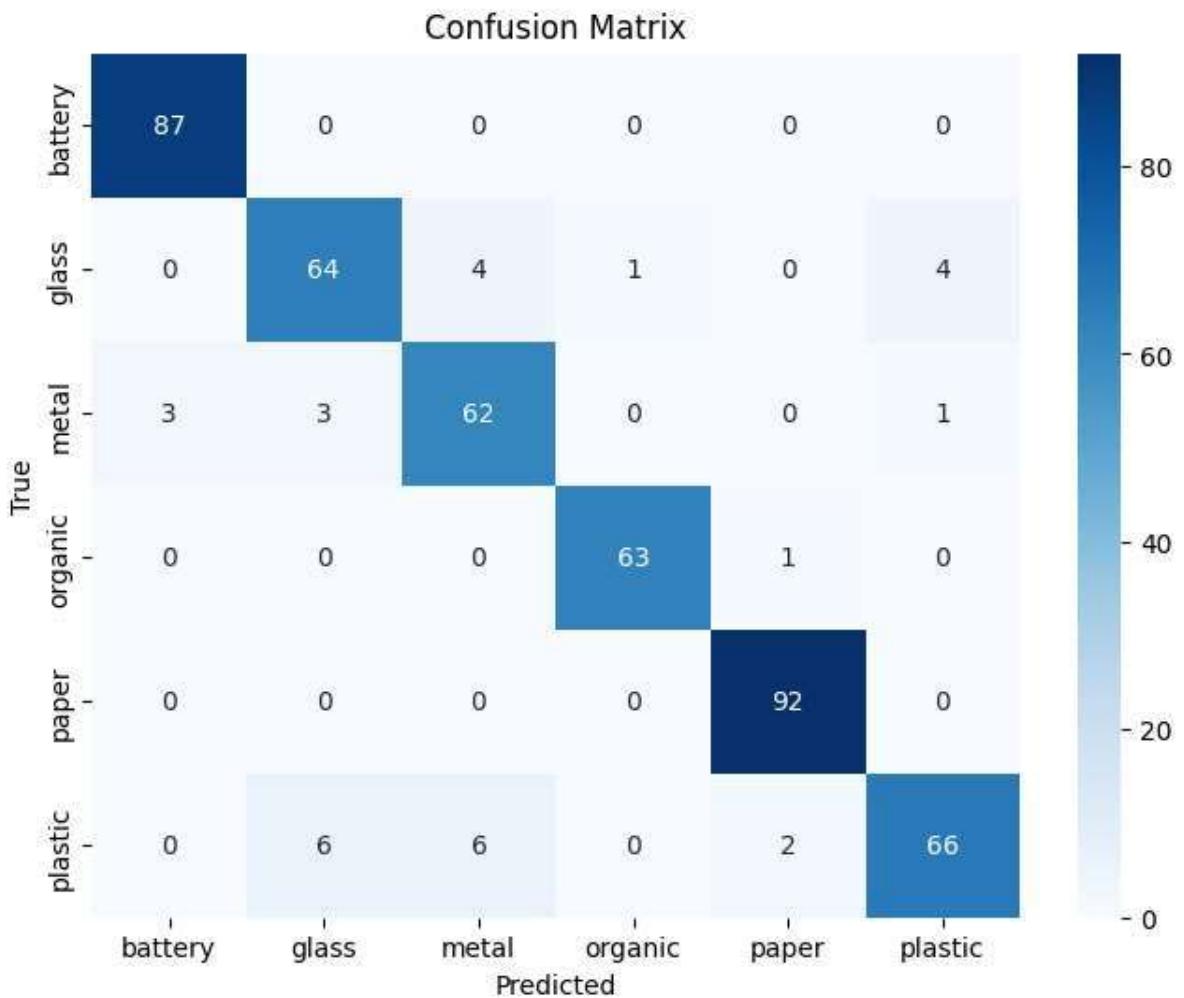
# Get the true class labels
true_labels = test_data.classes

# Compute confusion matrix
conf_matrix = confusion_matrix(true_labels, predicted_labels)

# Plot confusion matrix using seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=test_data.class_indices.keys(),
            yticklabels=test_data.class_indices.keys())
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()

# Print classification report
class_names = list(test_data.class_indices.keys())
print(classification_report(true_labels, predicted_labels, target_names=class_names))
```

```
8/8 [=====] - 6s 448ms/step
```



	precision	recall	f1-score	support
battery	0.97	1.00	0.98	87
glass	0.88	0.88	0.88	73
metal	0.86	0.90	0.88	69
organic	0.98	0.98	0.98	64
paper	0.97	1.00	0.98	92
plastic	0.93	0.82	0.87	80
accuracy			0.93	465
macro avg	0.93	0.93	0.93	465
weighted avg	0.93	0.93	0.93	465

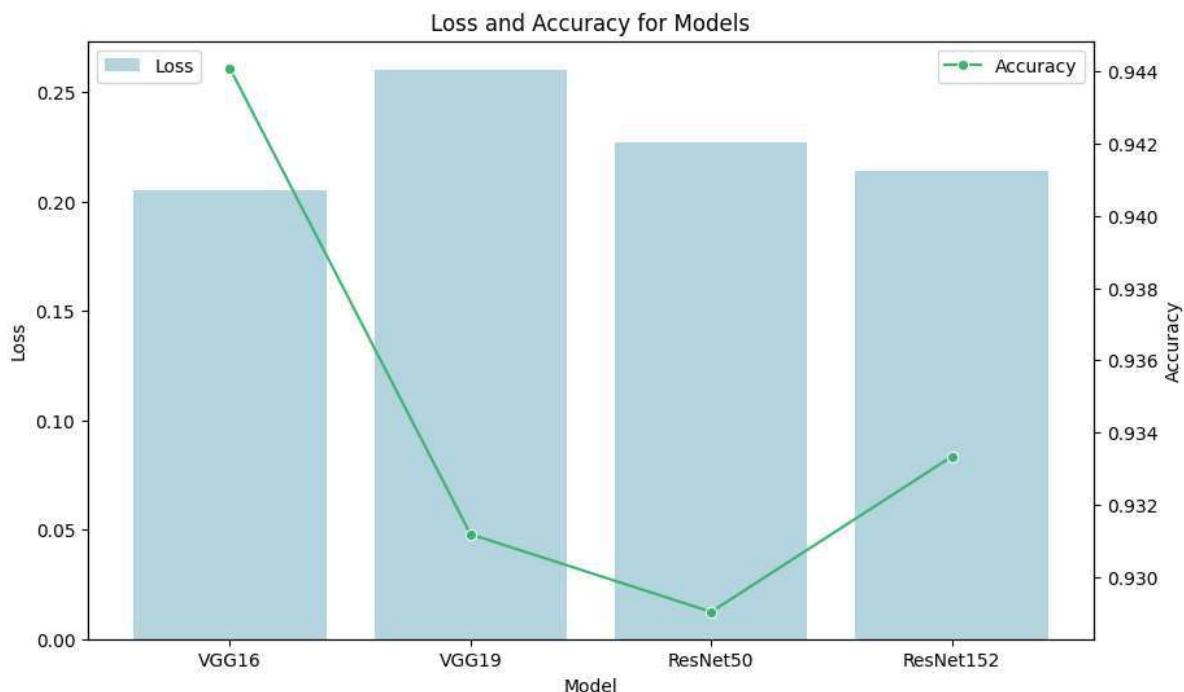
So sánh loss và accuracy trên tập test ở từng model

Cuối cùng, để có thể quyết định chọn model nào là ổn định và tốt nhất cho việc serving API, ta so sánh loss và accuracy của từng model.

```
In [28]: models = [vgg16, vgg19, resnet50, resnet152]
model_names = ['VGG16', 'VGG19', 'ResNet50', 'ResNet152']

loss_values = []
accuracy_values = []
# Evaluate and store Loss/accuracy for each model
for model, name in zip(models, model_names):
    loss, accuracy = model.evaluate_generator(test_data)
    loss_values.append(loss)
    accuracy_values.append(accuracy)
```

```
In [29]: loss_color = 'lightblue'
accuracy_color = 'mediumseagreen'
fig, ax1 = plt.subplots(figsize=(10, 6))
sns.barplot(x=model_names, y=loss_values, color=loss_color, ax=ax1, label='Loss')
ax2 = ax1.twinx()
sns.lineplot(x=model_names, y=accuracy_values, color=accuracy_color, marker='o', ax=ax2)
ax1.set_xlabel('Model')
ax1.set_ylabel('Loss', color='black')
ax2.set_ylabel('Accuracy', color='black')
plt.title('Loss and Accuracy for Models')
ax1.legend(loc='upper left')
ax2.legend(loc='upper right')
plt.show()
```



Như kết quả trên, thứ tự loss và accuracy tốt nhất lần lượt là:

- Loss: VGG16 - ResNet152 - ResNet50 - VGG19
- Accuracy: VGG16 > ResNet152 > VGG19 > ResNet50

Chọn lựa model có dung lượng/kích thước phù hợp để serving

Để xem và so sánh kích thước của các model, ta viết hàm lấy kích thước dựa theo link từng file:

- *Input*: file_path
- *Output*: size

```
In [30]: def get_file_size(file_path):
    if os.path.exists(file_path):
        # Lấy kích thước của file và chuyển đổi thành đơn vị MB
        size_in_bytes = os.path.getsize(file_path)
        size_in_mb = size_in_bytes / (1024 * 1024) # 1 MB = 1024 KB = 1024 * 1024
    return size_in_mb
```

```
In [31]: size_vgg16 = get_file_size('/kaggle/input/garbage-classifier-model/garbage_classifi
size_vgg19 = get_file_size('/kaggle/input/garbage-classifier-model/garbage_classifi
```

```
size_resnet50 = get_file_size('/kaggle/input/garbage-classifier-model/garbage_classifier.h5')
size_resnet152 = get_file_size('/kaggle/input/model-resnet/model_resnet152.h5')
```

```
In [32]: # Đường dẫn đến các file .h5
file_paths = [
    '/kaggle/input/garbage-classifier-model/garbage_classifier.vgg_16.h5',
    '/kaggle/input/garbage-classifier-model/garbage_classifier.vgg_19.h5',
    '/kaggle/input/garbage-classifier-model/garbage_classifier.resnet50.h5',
    '/kaggle/input/garbage-classifier-model/garbage_classifier.resnet152.h5'
]

# Kích thước của các file
file_sizes = [get_file_size(file_path) for file_path in file_paths]

# Tên của các mô hình
model_names = ['VGG16', 'VGG19', 'ResNet50', 'ResNet152']
for model, size in zip(model_names, file_sizes):
    print(f"Size (MB) of model {model}: {size:.2f} MB")
```

Size (MB) of model VGG16: 374.35 MB
Size (MB) of model VGG19: 100.56 MB
Size (MB) of model ResNet50: 97.39 MB
Size (MB) of model ResNet152: 231.06 MB

Mặc dù `VGG16` và `ResNet152` cho kết quả ổn định và tốt nhất nhưng kích thước của model là rất lớn (374.35MB và 231.06MB). Khi serving model này thành API và đưa lên AWS đòi hỏi chi phí, tài nguyên cao và tốc độ xử lý chậm. Do đó, chúng ta có thể sử dụng model tốn ít bộ nhớ hơn như `ResNet50`. Model này có accuracy thua 2 model tốt còn lại 0.1-0.2% và loss thì kém hơn 0.01-0.02 nhưng kích thước ít hơn 2-3 lần như so sánh ở trên.

-> Như vậy, việc chọn `ResNet50` sẽ là tối ưu nhất về hiệu năng model lẫn hiệu năng phần cứng.

Model Serving as API:

- Cuối cùng, để có thể đưa model đã thực hiện lên web thành sản phẩm cho người dùng sử dụng, chúng ta tiến hành viết API để serving model.
- Python API framework mà chúng em sử dụng là FastAPI. Lý do sử dụng là vì FastAPI là một micro framework khá mới, cho hiệu suất cao, đơn giản và dễ viết. Bên cạnh đó, FastAPI còn hỗ trợ `/docs` để dễ dàng kiểm tra và sử dụng.
- Link github: https://github.com/thangbuiq/simple_mlops/blob/main/backend/main.py

Các hàm sử dụng cho các route được định nghĩa ở file `utils.py` sau:
https://github.com/thangbuiq/simple_mlops/blob/main/backend/utils.py

- Giải thích chi tiết:
 - Đầu tiên import các thư viện và modules cần thiết.
 - Khởi tạo các biến môi trường: Sử dụng `os.environ.get` để lấy giá trị của các biến môi trường `PUBLIC_IP_ADDRESS` và `PUBLIC_DNS_ADDRESS`, sau đó định nghĩa danh sách địa chỉ origins để xác định các nguồn được phép truy cập API.
 - Khởi tạo API `app = FastAPI` gồm tên và mô tả tương ứng
 - `app.add_middleware` : cho phép yêu cầu từ list nguồn origins ở trước.

- Định nghĩa Base Model `trash` : Sử dụng Pydantic để định nghĩa một json-schema có tên là "trash" với một trường là "type_trash" kiểu dữ liệu là string (bắt buộc người dùng API phải theo format này).

GET	/ Root	▼
POST	/predict Predict Endpoint	▼
POST	/get-advice Give Advice	▼

- Các endpoint của API:

- Các endpoint `upload` và `download` dùng để debug/test.
- `@app.get("/")` :
 - Method: GET.
 - Mục đích: Trả về thông điệp chào và hướng dẫn sử dụng API.
 - Output: Một JSON object chứa lời giới thiệu và hướng dẫn.

Code	Details
200	<p>Response body</p> <pre>{ "message": "This is a garbage classification API", "help": "Use /predict to get the output for classification" }</pre> <p>o Copy Download</p>

- `@app.post("/predict")` :
 - Method: POST.
 - Mục đích: nhận và xử lý một file hình từ server, sau đó dự đoán phân loại và độ chính xác.
 - Input: file hình.
 - Output: đường dẫn của file, giá trị dự đoán và độ chính xác được dự đoán.

Code	Details
200	<p>Response body</p> <pre>{ "path": "upload/plastic-bag-be-green-1.jpg", "predicted_value": "plastic", "predicted_accuracy": 99.36 }</pre> <p>o Copy Download</p>

- `@app.post("/get-advice")` :
 - Method: POST.
 - Mục đích: nhận thông tin về loại rác, sau đó đưa ra lời khuyên xử lý rác và tốc độ xử lý.
 - Input: Loại rác.
 - Output: lời khuyên xử lý, thời gian.

Code	Details
200	Response body <pre>{ "advice": "- Plastic waste takes hundreds of years to decompose in the environment.\n- Dispose of plastic waste in recycling bins whenever possible to reduce its environmental impact.\n- Avoid single-use plastic products and opt for reusable alternatives to minimize plastic waste.", "time": 1.998572826385498 }</pre> <div style="display: flex; justify-content: space-between; align-items: center;"> Copy Download </div>

- Sử dụng `uvicorn.run` để chạy ứng dụng FastAPI với cài đặt host là "0.0.0.0" và port là 8000.

```

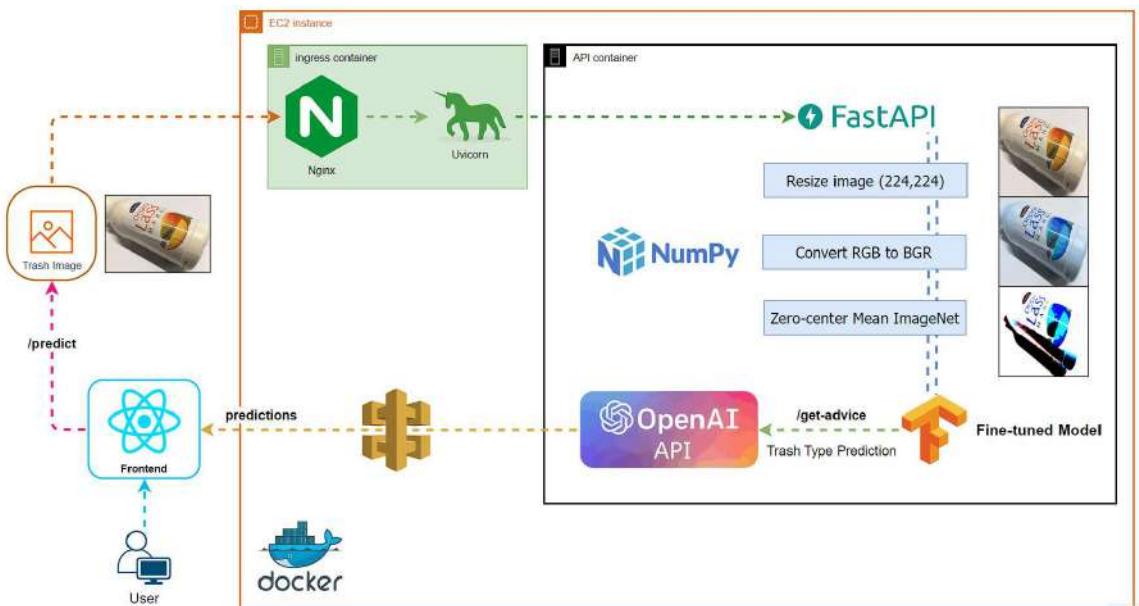
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [1352] using StatReload
INFO:     Started server process [1391]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     127.0.0.1:59552 - "GET / HTTP/1.1" 200 OK
INFO:     127.0.0.1:59552 - "GET /favicon.ico HTTP/1.1" 404 Not Found
INFO:     127.0.0.1:59552 - "GET /docs HTTP/1.1" 200 OK
INFO:     127.0.0.1:59552 - "GET /openapi.json HTTP/1.1" 200 OK
1/1 [=====] - 1s 627ms/step
INFO:     127.0.0.1:59558 - "POST /predict HTTP/1.1" 200 OK

```

Deployment On Amazon Elastic Compute Cloud (AWS EC2) with Github Actions

Để làm một ví dụ minh họa cho API (call và sử dụng API mà chúng em đã implement), chúng em viết một app React đơn giản gồm các chức năng chính như: upload ảnh, mở camera chụp ảnh, drag and drop ảnh và paste ảnh,... -> Tất cả những thao tác này sẽ gửi ảnh vào trong một **formData** đến lần lượt là route **/predict**, và sau đó là **/get-advice** ở API host trên cùng một nơi với app frontend (port 8000).

Bây giờ, ta nhìn qua lại kiến trúc software được sử dụng để deploy:

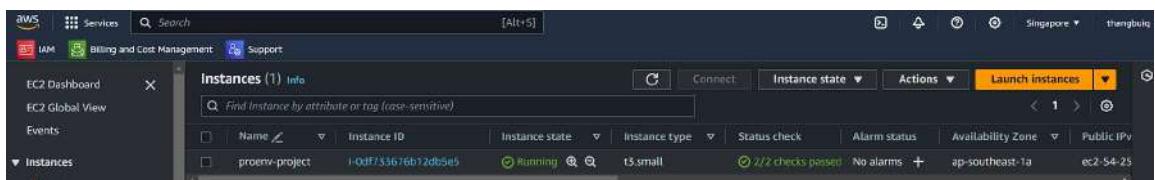


Để một app chạy thành công trên server, trước tiên ta phải chạy nó thành công ở máy local. Để giảm tải việc setup quá nhiều thứ trên server, ta sẽ **containerize** toàn bộ app của chúng

ta lại thành một docker image có tên **proenv_app** (kích thước image là **1.49GB**). Cuối cùng khi deploy, ta chỉ cần lén server và cài mỗi **Docker**.



Trước tiên, ta cần tạo một máy EC2 trên AWS. Tui em sử dụng chính account cá nhân của mình và bắt đầu lựa chọn instance với chi phí phù hợp.



Ở đây, nhóm đã thử qua các instance type có free-tier lần lượt là "t2.nano", "t2.micro", "t3.micro" và nhận thấy được hiệu năng của hai cấu hình máy này không thể đáp ứng nổi việc host hay deploy một model Keras. Do đó nhóm đã chọn một lựa chọn có phần tốn kém hơn chút là "**t3.small**". (với instance image sử dụng là Amazon Linux)

Name	vCPUs	Baseline Performance / vCPU	Memory	Price / Hour (Linux)	Price / Hour (Windows)
t3.nano	2	5%	0.5 GiB	\$0.0052	\$0.0098
t3.micro	2	10%	1 GiB	\$0.0104	\$0.0196
t3.small	2	20%	2 GiB	\$0.0209	\$0.0393
t3.medium	2	20%	4 GiB	\$0.0418	\$0.0602
t3.large	2	30%	8 GiB	\$0.0835	\$0.1111
t3.xlarge	4	40%	16 GiB	\$0.1670	\$0.2406
t3.2xlarge	8	40%	32 GiB	\$0.3341	\$0.4813

Nếu ta chạy liên tục máy **Linux t3.small** trong 1 tháng thì chi phí rơi vào khoảng **0.0209 x 24 x 30 = 15.04 \$ = 362.000 VNĐ**

Do nhóm chỉ mở vào những lúc cần thiết khi debug nên chi phí chỉ khoảng vài chục nghìn đồng

Cuối cùng ta set inbound rules lần lượt cho security group attach vào EC2 đã tạo với các port như sau:

- **8888:80** cho app frontend - serving bởi **nginx**.

- **8000:8000** cho backend (là API integrate với Tensorflow Keras Model của chúng ta)

Inbound rules Info						
Security group rule ID	Type Info	Protocol Info	Port range	Source Info	Description - optional Info	
sgr-01a7462f3050cfb84	Custom TCP	TCP	8888	Custom	Q 0.0.0.0/0 X ingress	Delete
sgr-0f021c5e63384724f	SSH	TCP	22	Custom	Q 0.0.0.0/0 X ssh	Delete
sgr-0ded6043cd331835	HTTP	TCP	80	Custom	Q 0.0.0.0/0 X http 80	Delete
sgr-013cfcd4f06052360	Custom TCP	TCP	8000	Custom	Q 0.0.0.0/0 X backend	Delete

Sau khi đã set inbound rules, ta thành công truy cập vào đường dẫn <http://PublicIPv4DNS:8888>. Đây sẽ là app frontend fetch response từ API <http://PublicIPv4DNS:8000/ROUTE> của chúng ta.



Chi phí về RAM, CPU khi chạy container của chúng ta trên server:

CPU									
Mem									
Swap									
[Alt+S]									
0.0% Tasks: 50, 83 thr, 72 kthr; 1 running									
0.0% Load average: 0.18 0.06 0.02									
1.26G/1.86G Uptime: 09:18:01									
OK/OK									
Main									
PID USER PRIO NI UPRG. RES SHR S CPU% MEM% TIME+ Command									
4154 root 20 0 2382M 700M 162M S 0.0 36.9 0:58.81 /usr/local/bin/python3 -c from multiprocessing.spawn import									
4155 root 20 0 2382M 700M 162M S 0.0 36.9 0:00.09 /usr/local/bin/python3 -c from multiprocessing.spawn import									
4162 root 20 0 2382M 700M 162M S 0.0 36.9 0:00.00 /usr/local/bin/python3 -c from multiprocessing.spawn import									
4163 root 20 0 2382M 700M 162M S 0.0 36.9 0:01.33 /usr/local/bin/python3 -c from multiprocessing.spawn import									
4164 root 20 0 2382M 700M 162M S 0.0 36.9 0:01.34 /usr/local/bin/python3 -c from multiprocessing.spawn import									
4165 root 20 0 2382M 700M 162M S 0.0 36.9 0:00.29 /usr/local/bin/python3 -c from multiprocessing.spawn import									
4166 root 20 0 2382M 700M 162M S 0.0 36.9 0:00.15 /usr/local/bin/python3 -c from multiprocessing.spawn import									
4172 root 20 0 2382M 700M 162M S 0.0 36.9 0:00.00 /usr/local/bin/python3 -c from multiprocessing.spawn import									
4506 root 20 0 2382M 700M 162M S 0.0 36.9 0:00.04 /usr/local/bin/python3 -c from multiprocessing.spawn import									
2712 root 20 0 1874M 560M 94360 S 0.0 29.5 1:08.32 python3 main.py									
2716 root 20 0 1874M 560M 94360 S 0.0 29.5 0:00.08 python3 main.py									
2717 root 20 0 1874M 560M 94360 S 0.0 29.5 0:00.00 python3 main.py									
2718 root 20 0 1874M 560M 94360 S 0.0 29.5 0:00.16 python3 main.py									
2719 root 20 0 1874M 560M 94360 S 0.0 29.5 0:00.16 python3 main.py									
2720 root 20 0 1874M 560M 94360 S 0.0 29.5 0:00.00 python3 main.py									
2721 root 20 0 1874M 560M 94360 S 0.0 29.5 0:00.03 python3 main.py									

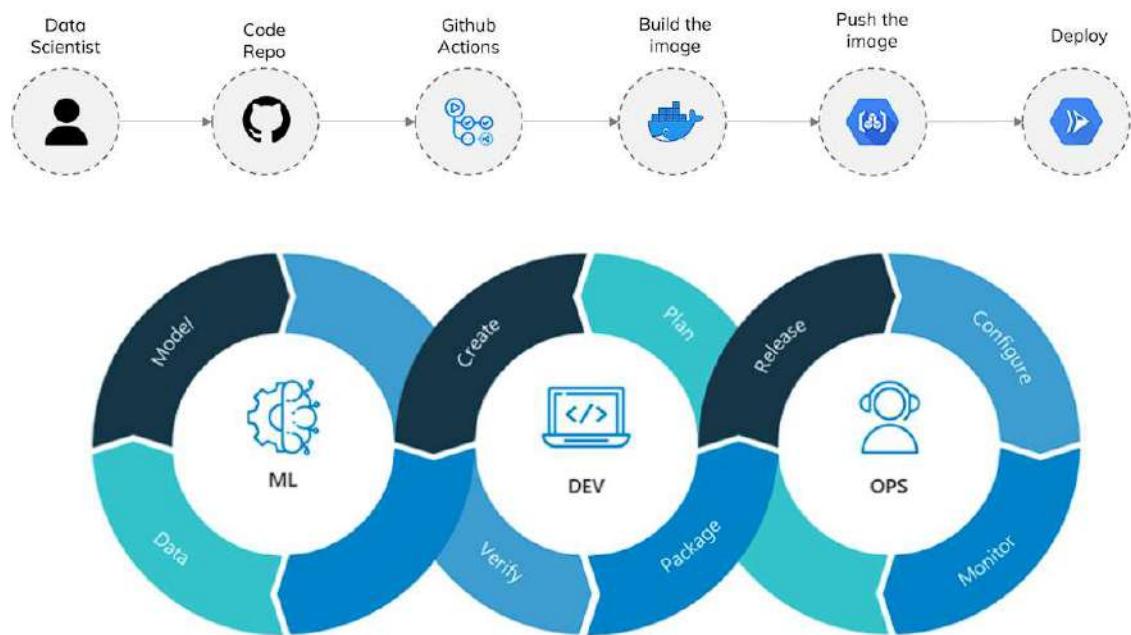
Tự động hóa quy trình Deploy (CI/CD với Github Actions)

Để tránh việc deploy thủ công dài dòng lặp đi lặp lại mỗi lần có thay đổi mới trên app, nhóm đã implement một workflow/pipeline CI/CD viết bằng Github Actions, pipeline sẽ thực hiện theo thứ tự như sau:

Release Tag -> Build Docker Image With Tag -> SSH vào EC2 -> Pull Docker Image -> Docker Run với Các Args cần thiết

File workflow được định nghĩa ở thư mục sau:

https://github.com/thangbuiq/simple_mlops/tree/main/.github/workflows



Full Source Code: Website, Model, API

Toàn bộ source code về model, backend, frontend, lần deployment hay cách Dockerize đều được nhóm public source ở repository sau: https://github.com/thangbuiq/simple_mlops.git

Thành quả cuối cùng sau khi user upload hay chụp một ảnh:

