# Model Training

```
pip install tensorflow
import tensorflow as tf

w = tf.constant(2.0)
b = tf.constant(3.0)
x = tf.constant(4.0)

#y = tf.multiply(w,x)
#y = tf.add(y,b)

print("Output:",w)
print("Output:",b)
print("Output:",x)
import os
import cv2
import numpy as np
from glob import glob
import matplotlib.pyplot as plt

# Root directory
dataset_dir = "dataset"

base_dir = "dataset"
train_dir = os.path.join(base_dir, "train")
test_dir = os.path.join(base_dir, "test")
```

```
# Check subfolders
print("Train subfolders:", os.listdir(train_dir))
print("Test subfolders:", os.listdir(test_dir))
# Collect all image paths from train and test
image_paths = glob(os.path.join(dataset_dir, "*/*/*.jpg"))  # Change *.jpg if needed

# Print the total number of images
print(f"Total images in dataset: {len(image_paths)}")

# Resize image to a specific size
def resize_image(image, size=(224, 224)):
    return cv2.resize(image, size)

# Normalize pixel values to range [0, 1]
def normalize_image(image):
    return image / 255.0

# Display an image (for verification)
def show_image(image, title="Image"):
    plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
    plt.title(title)
    plt.axis("off")
    plt.show()

def preprocess_and_save_all_images(image_paths, output_base_dir):
    for path in image_paths:
        # Read the image
        image = cv2.imread(path)
        if image is None:  # Handle unreadable images
```

```
            print(f"Warning: Unable to read image {path}. Skipping...")
            continue

        # Resize and normalize the image
        resized_image = resize_image(image)
        normalized_image = normalize_image(resized_image)

        # Reconstruct the new save path
        relative_path = os.path.relpath(path, dataset_dir)  # Get relative path
        save_path = os.path.join(output_base_dir, relative_path)  # Add base output directory
        os.makedirs(os.path.dirname(save_path), exist_ok=True)  # Create class-specific folder

        # Save the processed image
        cv2.imwrite(save_path, (normalized_image * 255).astype(np.uint8))

# Display a sample preprocessed image
sample_image_path = glob(os.path.join(output_dir, "**", "*.jpg"), recursive=True)  # Get preprocessed images
if sample_image_path:
    sample_image = cv2.imread(sample_image_path[0])  # Read the first preprocessed image
    show_image(sample_image, title="Sample Preprocessed Image")
else:
    print("No preprocessed images found.")

import os
import numpy as np
import matplotlib.pyplot as plt
import cv2
from keras.models import Sequential
```

```
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, Activation
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.utils import plot_model
from keras.layers import BatchNormalization
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.optimizers import Adam

train_path = "./preprocessed/train"  # Update with the dataset's train folder path
test_path = "./preprocessed/test"   # Update with the dataset's test folder path

batch_size = 32  # Adjust based on your system's GPU memory

# Augment and normalize training data
train_datagen = ImageDataGenerator(
    rescale=1.0/255.0,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)

# Normalize test data
test_datagen = ImageDataGenerator(rescale=1.0/255.0)

# Create generators
train_generator = train_datagen.flow_from_directory(
```

```
    train_path,
    target_size=(224, 224),
    batch_size=batch_size,
    class_mode='categorical'  # For multi-class classification
)

test_generator = test_datagen.flow_from_directory(
    test_path,
    target_size=(224, 224),
    batch_size=batch_size,
    class_mode='categorical'
)

from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam

# Model Architecture
model = Sequential()

# Convolutional Layers with Batch Normalization
model.add(Conv2D(32, (3, 3), input_shape=(224, 224, 3), activation='relu'))
model.add(BatchNormalization())  # Normalizing layer
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
```

```
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(256, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))

# Fully Connected Layers
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.6))  # Dropout to prevent overfitting
model.add(Dense(2, activation='softmax'))  # Output layer with two classes (Organic and Recyclable)

# Compile the model
model.compile(
    optimizer=Adam(learning_rate=0.0001),  # Learning rate
    loss='categorical_crossentropy',  # Cross-entropy loss for classification
    metrics=['accuracy']  # Accuracy as evaluation metric
)
train_path = './preprocessed/train'  # Directory for training data
test_path = './preprocessed/test'    # Directory for test data

from tensorflow.keras.preprocessing.image import ImageDataGenerator



# Augment and normalize training data
train_datagen = ImageDataGenerator(
    rescale=1.0/255.0,  # Normalize images
```

```
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)

# Normalize test data
test_datagen = ImageDataGenerator(rescale=1.0/255.0)

# Create the training and testing data generators
train_generator = train_datagen.flow_from_directory(
    train_path,
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical'  # For multi-class classification
)

test_generator = test_datagen.flow_from_directory(
    test_path,
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical'
)

batch_size = 32  # Adjust based on your system's GPU memory

# Augment and normalize training data
```

```
train_datagen = ImageDataGenerator(
    rescale=1.0/255.0,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)

# Normalize test data
test_datagen = ImageDataGenerator(rescale=1.0/255.0)

# Create generators
train_generator = train_datagen.flow_from_directory(
    train_path,
    target_size=(224, 224),
    batch_size=batch_size,
    class_mode='categorical'  # For multi-class classification
)

test_generator = test_datagen.flow_from_directory(
    test_path,
    target_size=(224, 224),
    batch_size=batch_size,
    class_mode='categorical'
)

from keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau
```

```
# Callbacks
callbacks = [
    ModelCheckpoint('best_waste_segregation_model.h5', save_best_only=True, monitor='val_loss', mode='min'),
    EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True),
    ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3, verbose=1)  # Reduce learning rate when loss plateaus
]

# Train the model
history = model.fit(
    train_generator,
    steps_per_epoch=len(train_generator),
    epochs=10,  # Adjust as per your needs
    validation_data=test_generator,
    validation_steps=len(test_generator),
    callbacks=callbacks  # Adding callbacks for early stopping and checkpointing
)

# Evaluate model on test data
test_loss, test_accuracy = model.evaluate(test_generator)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
print(f"Test Loss: {test_loss:.4f}")

# Plot Accuracy
import matplotlib.pyplot as plt

plt.plot(history.history['accuracy'], label='Train Accuracy')
```

```
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.title('Accuracy')
plt.show()

# Plot Loss
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.title('Loss')
plt.show()

model.save('waste_segregation_model.h5')  # Save the model

import numpy as np
import cv2

def preprocess_image(image_path):
    """Preprocess a single image for prediction."""
    img = cv2.imread(image_path)
    img = cv2.resize(img, (224, 224)) / 255.0  # Normalize
    img = np.expand_dims(img, axis=0)  # Add batch dimension
    return img

def predict_image(image_path, model, class_names):
    img = preprocess_image(image_path)
    prediction = model.predict(img)
    predicted_class = np.argmax(prediction)
    print(f"The image is classified as: {class_names[predicted_class]} "
```

```
        f"(Confidence: {np.max(prediction)*100:.2f}%)")

# Class names from the train generator
class_names = list(train_generator.class_indices.keys())

# Test predictions with sample images
predict_image('./preprocessed/test/O/O_12573.jpg', model, class_names)
predict_image('./preprocessed/test/R/R_10753.jpg', model, class_names)

test_datagen = ImageDataGenerator(rescale=1.0/255.0)

model.add(Dropout(0.5))  # Already present, try increasing to 0.6

def predict_image(image_path, model, class_names):
    img = cv2.imread(image_path)
    img = cv2.resize(img, (224, 224)) / 255.0  # Normalize
    img = np.expand_dims(img, axis=0)  # Add batch dimension

    prediction = model.predict(img)
    predicted_index = np.argmax(prediction)  # Get predicted class index

    # Ensure index matches class names correctly
    predicted_label = class_names[predicted_index]

    print(f"The image is classified as: {predicted_label} (Confidence: {np.max(prediction)*100:.2f}%)")

print(train_generator.class_indices)

from tensorflow.keras.models import load_model
```

```
from tensorflow.keras.optimizers import Adam


# Load and recompile the model

model = load_model('best_waste_segregation_model.h5')

model.compile(optimizer=Adam(learning_rate=0.0001), loss='categorical_crossentropy',
metrics=['accuracy'])


class_names = list(train_generator.class_indices.keys())  # Ensure correct order


test_images = ['./preprocessed/test/O/O_12573.jpg', './preprocessed/test/R/R_10753.jpg']


for img_path in test_images:
    predict_image(img_path, model, class_names)


import cv2

import matplotlib.pyplot as plt

import os


# Function to display an image

def show_image(image_path, title):
    img = cv2.imread(image_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)  # Convert BGR to RGB for correct colors
    plt.imshow(img)
    plt.title(title)
    plt.axis("off")
    plt.show()


# Paths for test dataset

test_organic_path = "./preprocessed/test/O"
```

```
test_recyclable_path = "./preprocessed/test/R"


# Get image filenames from the directories
test_organic_images = os.listdir(test_organic_path) if os.path.exists(test_organic_path) else []
test_recyclable_images = os.listdir(test_recyclable_path) if os.path.exists(test_recyclable_path) else []


# Display first 10 images from each category (if available)
def show_multiple_images(images, title):
    for i, image_name in enumerate(images[:20]):  # Display first 10 images
        image_path = os.path.join(test_organic_path if title == "Organic Waste" else test_recyclable_path, image_name)
        show_image(image_path, f"{title} - Image {i + 1}")


if test_organic_images:
    show_multiple_images(test_organic_images, "Organic Waste")
else:
    print("No Organic images found in the test set!")


if test_recyclable_images:
    show_multiple_images(test_recyclable_images, "Recyclable Waste")
else:
    print("No Recyclable images found in the test set!")
```

# APP.PY

```
import streamlit as st
import tensorflow as tf
import numpy as np
from keras.models import Sequential
```

```
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam
from PIL import Image
import base64

# Function to Set Background Image
def set_bg(image_path):
    with open(image_path, "rb") as img_file:
        encoded_string = base64.b64encode(img_file.read()).decode()
    st.markdown(
        f"""
        <style>
        .stApp {{
            background-image: url("{encoded_string}");
            background-size: cover;
            background-position: center;
            background-attachment: fixed;
        }}
        </style>
        """,
        unsafe_allow_html=True
    )

# Load Background Image
set_bg("iws.png")  # Ensure "iws.png" is in the same folder

# Define Model Architecture
def create_model():
    model = Sequential([
```

```
        Conv2D(32, (3, 3), input_shape=(224, 224, 3), activation='relu'),
        BatchNormalization(),
        MaxPooling2D(pool_size=(2, 2)),

        Conv2D(64, (3, 3), activation='relu'),
        BatchNormalization(),
        MaxPooling2D(pool_size=(2, 2)),

        Conv2D(128, (3, 3), activation='relu'),
        BatchNormalization(),
        MaxPooling2D(pool_size=(2, 2)),

        Conv2D(256, (3, 3), activation='relu'),
        BatchNormalization(),
        MaxPooling2D(pool_size=(2, 2)),

        Flatten(),
        Dense(256, activation='relu'),
        Dropout(0.6),
        Dense(2, activation='softmax')
    ])

    model.compile(optimizer=Adam(learning_rate=0.0001), loss='categorical_crossentropy',
metrics=['accuracy'])
    return model

# Load Model
model = create_model()
model.load_weights('best_waste_segregation_model.h5')
```

```
# Custom CSS for Styling
st.markdown(
    """
    <style>
    /* File Uploader Styling */
    .file-uploader {
        border: 2px dashed #ffffff;
        padding: 20px;
        text-align: center;
        font-size: 18px;
        font-family: 'Times New Roman', serif;
        background: rgba(255, 255, 255, 0.2);
        border-radius: 10px;
        color: white;
        margin-bottom: 20px;

    }

    /* Centering Text */
    .center-text {
        text-align: center;
        font-family: 'Times New Roman', serif;
        color: white;
    }

    /* Prediction Box */
    .prediction-box {
        text-align: center;
```

```
        font-size: 22px;
        font-weight: bold;
        color: white;
        background: linear-gradient(to right, #00ff00, #00aaff);
        padding: 10px;
        border-radius: 10px;
        display: inline-block;
        margin-top: 20px;
    }
    </style>
    """,
    unsafe_allow_html=True
)

# Title
st.markdown("<h1 class='center-text'>Waste Segregation Model</h1>", unsafe_allow_html=True)

# Styled File Uploader Box
st.markdown("<div class='file-uploader'>Drag and drop an image here or click to upload</div>",
unsafe_allow_html=True)
uploaded_file = st.file_uploader("", type=["jpg", "png", "jpeg"])

# Process Image if Uploaded
if uploaded_file is not None:
    image = Image.open(uploaded_file)

    # Display Uploaded Image with Styling
    st.markdown("<h3 class='center-text'>Uploaded Image:</h3>", unsafe_allow_html=True)
    st.image(image, caption="", use_container_width=True)
```

```
    # Preprocess Image
    image = image.resize((224, 224))
    image = np.array(image) / 255.0
    image = np.expand_dims(image, axis=0)

    # Make Prediction
    prediction = model.predict(image)
    predicted_class = np.argmax(prediction)

    # Class labels
    class_labels = ["Organic", "Recyclable"]

    # Display Prediction
    st.markdown(f"<div class='prediction-box'>Predicted Class: {class_labels[predicted_class]}</div>", unsafe_allow_html=True)
```

# Frontend.py

```
pip install streamlit h5py pandas

import streamlit as st
import h5py
import numpy as np
from PIL import Image
import tensorflow as tf

# Load the model once when the app starts (outside the image upload process)
@st.cache_resource
```

```
def load_model():
    model_path = 'best_waste_segregation_model.h5'
    model = tf.keras.models.load_model(model_path)
    return model

# Function to display the uploaded image
def display_image(uploaded_file):
    # Open the uploaded image file
    image = Image.open(uploaded_file)
    st.image(image, caption="Uploaded Image", use_container_width=True)
    return image

# Load the model once
model = load_model()

# Step 1: Upload an image file
uploaded_image = st.file_uploader("Upload an image", type=["jpg", "jpeg", "png"])

if uploaded_image is not None:
    # Step 2: Display the uploaded image
    image = display_image(uploaded_image)

    # Step 3: Preprocess the uploaded image for the model (assuming the model requires resizing)
    image = image.resize((224, 224))  # Resize to match model input size (if required by your model)
    image = np.array(image)  # Convert image to numpy array
    image = np.expand_dims(image, axis=0)  # Add batch dimension
    image = image / 255.0  # Normalize the image if required (depends on model)

    # Show a progress bar to inform the user while prediction is happening
```

```
    with st.spinner('Making prediction...'):
        # Step 4: Make a prediction using the model
        prediction = model.predict(image)

    # Step 5: Display the prediction result (modify this based on your model's output)
    st.write("Prediction Result:")
    st.write(prediction)  # Display the raw prediction, you can modify this depending on your model

    # Optional: Show predicted class or result based on your model's output
    predicted_class = np.argmax(prediction, axis=1)  # If it's a classification model
    st.write(f"Predicted Class: {predicted_class[0]}")  # Show the predicted class

!streamlit run app.py
```