**REPÚBLICA BOLIVARIANA DE VENEZUELA**

**MINISTERIO DEL PODER POPULAR PARA LA EDUCACIÓN UNIVERSITARIA**

**SAMSUNG INNOVATION CAMPUS**

**AULA VET-13**

# ECOCLASS: Clasificador de Residuos de Reciclaje.

**Tutores:**

Jenny Remolina

Alvaro Arauz

**Estudiantes:**

Julio Zambrano C.I. 30.720.044

Andrea Ruiz C.I. 30.708.907

Angel Villegas C.I. 29.698.892

**VENEZUELA, MARZO DE 2025.**

## Índice General:

**Contenido:** **pp.**

## Introducción:

En el presente informe se plasmará de manera informativa lo referente a los detalles técnicos de lo más relevante sobre el funcionamiento de la arquitectura del modelo del Proyecto de Inteligencia Artificial EcoClass: Clasificador de Residuos de reciclaje. En este breve informe tipo documentación, se logrará comprender de forma integral parte de las bases y criterio, que permiten el funcionamiento y utilización óptima de la aplicación, expresándose en palabras los resultados acerca de la precisión alcanzada.

Para que durante su despliegue quienes hagan uso de esta herramienta, puedan plantearse su mejora y mantenimiento, ajustándola a los futuros requerimiento en abordaje de esta problemática que engloba el reciclaje, así como guía personal respecto a nuestro trabajo. En este sentido funciona como contribución global en la cual nos aseguramos que lo más importante referente a la programación y el modelo utilizado, contribuya al empoderamiento e iniciativa propia de la comunidad donde se le implemente. Proporcionándoles todo lo necesario y así puedan dotarlo del alcance renovado que ocupe de forma personalizada su escalabilidad.

Se entiende que la clasificación automatizada de materiales reciclables mediante inteligencia artificial es crucial para optimizar procesos de gestión de residuos. Concretamente en este proyecto se implementa un clasificador de imágenes utilizando MobileNetV2, una red neuronal convolucional pre-entrenada, para distinguir entre cinco categorías: cartón (cardboard), vidrio (glass), metal (metal), papel (paper) y plástico (plastic). Durante este proceso se demostró cómo el transfer learning y técnicas de aumento de datos mejoran la precisión en tareas de visión por computadora, incluso con datasets limitados de imágenes, conseguidos y reestructurados de plataformas como Kaggle (imágenes de internet).

**Definiciones clave:**

- **Transfer Learning** se implementa mediante la adaptación de MobileNetV2, un modelo pre-entrenado en ImageNet, para clasificar imágenes entre cinco categorías generales de residuos, aprovechando características aprendidas previamente.
- **MobileNetV2**, una arquitectura de red neuronal ligera y eficiente, se elige por su equilibrio entre rendimiento y consumo computacional, ideal para entornos prácticos como dispositivos móviles o sistemas embebidos.
- **Aumento de Datos** introduce variaciones artificiales en las imágenes como rotaciones, desplazamientos, ajustes de brillo, para simular escenarios reales y mejorar la generalización del modelo.
- **Regularización** combina técnicas como Dropout, que es la desactivación aleatoria de neuronas y L2, penalización de pesos grandes, para mitigar el sobreajuste, especialmente relevante en datasets de tamaño moderado.
- **Métricas de Evaluación** incluyen precisión, exactitud global, recall es la capacidad de detectar todas las instancias de una clase y F1-Score, son el balance entre precisión y recall, que son críticas para validar el rendimiento en clases prioritarias (como podría ser el vidrio o el plástico presentes en el proyecto).

    - **Precisión**: Proporción de predicciones correctas.
    - **Recall**: Capacidad de detectar todas las instancias relevantes.
    - **F1-Score**: Media armónica entre precisión y recall.
- **Callbacks** como EarlyStopping, consiste en la detención anticipada ante estancamiento, ModelCheckpoint, para el guardado del mejor modelo y ReduceLROnPlateau que es el

ajuste dinámico de la tasa de aprendizaje, optimiza el entrenamiento y conservan recursos.

- **Batch Normalization** normaliza las activaciones intermedias para acelerar la convergencia y reducir la sensibilidad a inicializaciones aleatorias.
- **ImageDataGenerator** gestiona el flujo de datos, aplicando aumentos en tiempo real y dividiendo el dataset en entrenamiento/validación (70%/30%), con batches de 64 imágenes para aprovechar la memoria de la GPU.
- **Clases Desbalanceadas**, aunque no son un problema en este dataset (1,000 imágenes/clase), se abordan con pesos específicos (class_weights) para priorizar clases críticas en aplicaciones reales.
- **Google Colab + GPU** proporciona el entorno de ejecución, acelerando el entrenamiento mediante acceso a recursos gráficos y facilitando la integración con Google Drive para almacenamiento persistente.

**Explicación de los Bloques de Código Esenciales:**

## Parte 1: Importación de Librerías y Configuración Inicial


- **TensorFlow/Keras**: Para construir y entrenar el modelo de red neuronal.
- **NumPy/Matplotlib**: Manipulación de datos y visualización.
- **Google Colab**: Acceso a la GPU y montaje de Google Drive.
- **ImageDataGenerator**: Preprocesamiento y aumento de datos.
- **MobileNetV2**: Modelo base para transfer learning.
- **Sklearn**: Generación del reporte de clasificación.

**# Verificación de estructura del dataset**

```python
# 1. Verificación de estructura del dataset
def verify_dataset_structure(dataset_path, expected_classes):
    print("🔍 Verificando estructura del dataset...")
    if not os.path.exists(dataset_path):
        raise FileNotFoundError(f"Directorio no encontrado: {dataset_path}")

    detected_classes = sorted([d for d in os.listdir(dataset_path) if os.path.isdir(os.path.join(dataset_path, d))])

    if detected_classes != expected_classes:
        print(f"⚠️ Error: Clases detectadas {detected_classes} no coinciden con las esperadas {expected_classes}")
        return False

    # Verificar 5 imágenes aleatorias por clase
    for class_name in detected_classes:
        class_path = os.path.join(dataset_path, class_name)
        images = [f for f in os.listdir(class_path) if f.lower().endswith(('.png', '.jpg', '.jpeg'))]
        if len(images) < 5:
            print(f"⚠️ Clase {class_name} tiene menos de 5 imágenes")
            return False

        # Mostrar ejemplo de imagen
        img_path = os.path.join(class_path, images[0])
        img = tf.keras.preprocessing.image.load_img(img_path)
        if img.mode != 'RGB':
            print(f"⚠️ Imagen {img_path} no está en formato RGB")
            return False

    print("✅ Estructura del dataset validada correctamente")
    return True

# Lista de clases esperadas
expected_classes = ['cardboard', 'glass', 'metal', 'paper', 'plastic']

if not verify_dataset_structure(dataset_dir, expected_classes):
    raise ValueError("Problemas detectados en el dataset. Verificar estructura y archivos.")
```

```
🔍 Verificando estructura del dataset...
✅ Estructura del dataset validada correctamente
```

Esta es una de las funciones más importantes antes de realizar el pre-procesado de los datos, ya que con ella nos aseguramos que el dataset esté organizado correctamente, con las clases esperadas y archivos válidos, antes del entrenamiento del modelo. Primero verifica la existencia del directorio del dataset en el almecamiento Google Drive: Si la ruta no existe, detiene todo con un error.

**Compara las clases detectadas con las esperadas**, busca carpetas con los nombres cardboard, glass, metal, paper, y plastic. Si hay clases faltantes, adicionales o mal escritas, muestra un error. Se revisa que cada clase tenga al menos 5 imágenes: Si una clase tiene menos de 5 imágenes (en formatos .png, .jpg, o .jpeg), se reporta un problema. Esto evita entrenar con datasets demasiado pequeños.

Valida el **formato de las imágenes**, carga la primera imagen de cada clase para confirmar que esté en modo RGB, este es el formato estándar para modelos de visión por computadora. Por lo que si alguna imagen está en escala de grises o tiene otro formato, se detecta el error.

Entre los **resultados posibles**, si todo está correcto, imprime: ✅ Estructura del dataset validada correctamente. Si hay errores por clases incorrectas, imágenes insuficientes o formato inválido, detiene el programa con un mensaje claro indicando el problema.

**# Visualización de muestras:**

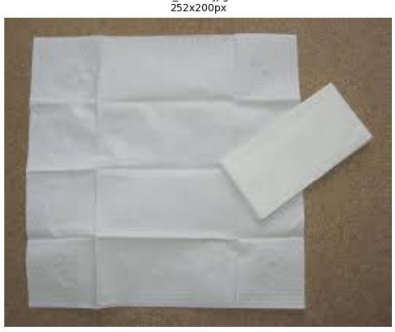
```
[ ]  # Visualización de muestras
     print("\n🖼️ MUESTRAS VISUALES")
     for category in counts.keys():
         display_category_samples(dataset_dir, category, num_samples=3)
```

**Categoría: PAPER**

R_10754.jpg
225x225px

R_10756.jpg
200x150px

R_10761.jpg
252x200px



**Categoría: METAL**

R_11111.jpg
1460x1500px


R_1624.jpg
224x224px


R_1626.jpg
251x201px


**Categoría: PLASTIC**

R_1005.jpg
241x209px

R_101.jpg
225x225px


R_1026.jpg
225x225px


Este paso es fundamental, ya que permite poder visualizar gráficamente las características representativas de 3 imágenes de cada clase del dataset. Ayudándonos a detectar errores que no se capturan en validaciones automáticas por ejemplo una foto de metal

etiquetada como glass. Y también evalúa la diversidad de imágenes dentro de cada clase, clave para poder entrenar un modelo consistente.

## **Parte 2: Procesado de la información:**

### 2. Preprocesamiento de los datos

- Parametros de entrenamiento 3500 (70%) para entrenar 1500 (30%) para validar
- tamaño de imagen para entrenar 224x224
- Normalizacion de datos
- Zoom y desplazamientos de imagenes
- Batch size optimizado para GPU
- Shuffle solo para datos de entrenamiento

```
# Configuración aumentos de datos
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=25,
    width_shift_range=0.15,
    height_shift_range=0.15,
    shear_range=0.15,
    zoom_range=0.15,
    horizontal_flip=True,
    vertical_flip=False,  # Menos relevante para objetos de reciclaje
    brightness_range=[0.9, 1.1],
    validation_split=0.3
)

# Generadores de datos
img_size = (224, 224)
batch_size = 64

train_generator = train_datagen.flow_from_directory(
    dataset_dir,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='categorical',
    subset='training',
    shuffle=True,
    color_mode='rgb'
)

val_generator = train_datagen.flow_from_directory(
    dataset_dir,
    target_size=img_size,
    batch_size=batch_size,
    class_mode='categorical',
    subset='validation',
    shuffle=False,
    color_mode='rgb'
)
```

A. **Aumento de Datos (Data Augmentation)**:

- **ImageDataGenerator** aplica transformaciones aleatorias a las imágenes de entrenamiento para evitar el sobreajuste (*overfitting*). Las transformaciones incluyen:
  - **Normalización**: rescale=1./255 (convierte valores de píxeles a [0, 1]).
  - **Rotación**: Hasta 25 grados (rotation_range=25).
  - **Desplazamientos**: Horizontal y vertical (15% del tamaño de la imagen).
  - **Deformaciones**: Inclinación (shear_range=0.15) y zoom (zoom_range=0.15).
  - **Volteo horizontal**: horizontal_flip=True (útil para objetos simétricos).
  - **Ajuste de brillo**: brightness_range=[0.9, 1.1] (variación del 10%).
  - **División entrenamiento/validación**: 70% para entrenar, 30% para validar (validation_split=0.3).
- **No se usa volteo vertical** (vertical_flip=False), ya que objetos como botellas o cajas no cambian su significado al voltearlos verticalmente.

B. **Generadores de Datos**:

- **train_generator y val_generator** cargan las imágenes desde el directorio del dataset:
  - **Redimensionan imágenes** a 224x224 píxeles (target_size), compatible con modelos como MobileNet.
  - **Batch size**: 64 imágenes por lote (optimizado para uso eficiente de la GPU).

- **Modo categórico**: class_mode='categorical' (para clasificación multi-clase).
- **Mezcla de datos**: Solo en entrenamiento (shuffle=True), no en validación.

**Parte 3: Construir el modelo**

Este código construye un modelo de clasificación de imágenes basado en transferencia learning usando MobileNetV2 como base, adaptándolo al problema específico de clasificación de imágenes de reciclaje de este proyecto. El propósito como tal es aprovechar las características aprendidas en ImageNet para acelerar el entrenamiento.

A. **Configuración:**
   - **include_top=False:** Elimina las capas finales de clasificación de MobileNetV2 para reemplazarlas con capas personalizadas.
   - **alpha=1.0:** Usa la versión completa del modelo (mejor precisión, aunque más pesada que otras versiones con alpha < 1).

**# Congelar capa base:**

- **Por qué:** Las primeras capas de MobileNetV2 detectan características genéricas (bordes, texturas), que son útiles para cualquier tarea de visión. Al congelarlas, se evita reentrenarlas y se reduce el costo computacional.
- **Capa 130:** Empíricamente, las capas posteriores a la 130 se ajustan para aprender características específicas del nuevo dataset (reciclaje).

**# Capas personalizadas:**

A. **GlobalAveragePooling2D**: Reduce la dimensionalidad de los mapas de características de MobileNetV2 (de 7x7x1280 a 1280 valores), simplificando la entrada para las capas densas.

B. **Dropout(0.6)**: Elimina aleatoriamente el 60% de las neuronas durante el entrenamiento para prevenir el overfitting, crucial si el dataset es pequeño.

C. **Capa Dense(256)**:

   - activation='relu': Introduce no linealidad.
   - Regularización L2: Penaliza pesos grandes en kernel y bias (l2(0.01)), controlando el sobreajuste.

D. **BatchNormalization**: Estabiliza el entrenamiento normalizando las salidas de la capa anterior.

E. **Capa de salida**: softmax para clasificación en 5 clases.

**# Optimizador Configurado y Compilación**:

A. **Adam**:

   - learning_rate=1e-4: Tasa baja para ajustes finos (no reentrenar desde cero).
   - Parámetros estándar (beta_1=0.9, beta_2=0.999).

B. **Métricas**:

   - accuracy: Precisión general.
   - Precision y Recall: Importantes para detectar falsos positivos/negativos, especialmente si hay clases desbalanceadas por ejemplo más imágenes de "plástic" que de "glass".

```python
def build_optimized_model(input_shape=(224, 224, 3), num_classes=5):
    # Cargar MobileNetV2 pre-entrenado
    base_model = MobileNetV2(
        input_shape=input_shape,
        include_top=False,
        weights='imagenet',
        alpha=1.0  # Versión completa
    )

    # Congelar capas base
    for layer in base_model.layers[:130]:
        layer.trainable = False

    # Capas personalizadas
    model = Sequential([
        base_model,
        GlobalAveragePooling2D(),
        Dropout(0.6),
        Dense(256,
              activation='relu',
              kernel_regularizer=l2(0.01),
              bias_regularizer=l2(0.01)),
        BatchNormalization(),
        Dense(num_classes, activation='softmax')
    ])
    # Optimizador configurado
    optimizer = Adam(
        learning_rate=1e-4,
        beta_1=0.9,
        beta_2=0.999,
        epsilon=1e-07,
        amsgrad=False
    )

    model.compile(
        optimizer=optimizer,
        loss='categorical_crossentropy',
        metrics=[
            'accuracy',
            tf.keras.metrics.Precision(name='precision'),
            tf.keras.metrics.Recall(name='recall')
        ]
    )

    return model

model = build_optimized_model()
model.summary()
```

```
Model: "sequential_1"
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| mobilenetv2_1.00_224 (Functional) | (None, 7, 7, 1280) | 2,257,984 |
| global_average_pooling2d_1 (GlobalAveragePooling2D) | (None, 1280) | 0 |
| dropout_2 (Dropout) | (None, 1280) | 0 |
| dense_2 (Dense) | (None, 256) | 327,936 |
| batch_normalization_1 (BatchNormalization) | (None, 256) | 1,024 |
| dense_3 (Dense) | (None, 5) | 1,285 |

```
Total params: 2,588,229 (9.87 MB)
Trainable params: 1,689,733 (6.45 MB)
Non-trainable params: 898,496 (3.43 MB)
```

| Capa (Tipo) | Output Shape | Parámetros | Función |
|---|---|---|---|
| MobileNetV2 | (None, 7, 7, 1280) | 2,257,984 | Extrae características visuales complejas (bordes, texturas, patrones). |
| GlobalAveragePooling2D | (None, 1280) | 0 | Reduce dimensionalidad (7x7x1280 → 1280 valores). |
| Dropout | (None, 1280) | 0 | Apaga el 60% de neuronas aleatoriamente para evitar sobreajuste. |
| Dense (256 neuronas) | (None, 256) | 327,936 | Capa oculta con regularización L2 (controla pesos grandes). |
| BatchNormalization | (None, 256) | 1,024 | Normaliza las salidas de la capa anterior (estabiliza el entrenamiento). |
| Dense (5 neuronas) | (None, 5) | 1,285 | Capa de salida para clasificar en 5 clases (activación: softmax). |

**Parte 4: Entrenar el modelo**

**# Callbacks (Acciones Automáticas)**

A. **EarlyStopping**:

- Detiene el entrenamiento si la precisión en validación no mejora durante 10 épocas seguidas. Evita perder tiempo en épocas que no aportan mejoras.

B. **ModelCheckpoint**:

- Guarda automáticamente el modelo con el mejor recall en validación, prioriza detectar todos los objetos relevantes, como vidrio o plástico). En el archivo best_model_reciclaje.keras.

C. **ReduceLROnPlateau**:

- Reduce a la mitad la tasa de aprendizaje *si la* pérdida en validación no mejora en 4 épocas. Ayudar al modelo a ajustarse mejor en etapas finales.

**# Pesos de clases**:

Se trata de equilibrar el entrenamiento si hay clases con menos imágenes. La clase glass (1) tiene peso **1.2**, lo que indica que es más importante o tiene menos ejemplos.

**# Entrenamiento:**

A. **Duración máxima**: 40 épocas, pero puede terminar antes por el *EarlyStopping*.
B. **Datos usados**:
    - **Entrenamiento**: 3500 imágenes (70% del dataset).
    - **Validación**: 1500 imágenes (30% del dataset).
C. **Enfoque**:
    - Usa MobileNetV2 (optimizado para ser rápido y ligero).
    - Prioriza detectar bien las clases críticas (como glass) con pesos y métricas de recall.

```python
# Callbacks mejorados
callbacks = [
    EarlyStopping(
        monitor='val_precision',
        patience=10,
        mode='max',
        restore_best_weights=True,
        verbose=1
    ),
    ModelCheckpoint(
        filepath=os.path.join(model_dir, 'best_model_reciclaje.keras'),
        monitor='val_recall',
        save_best_only=True,
        mode='max'
    ),
    ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.5,
        patience=4,
        min_lr=1e-6,
        verbose=1
    )
]

# Pesos de clases para balanceo
class_weights = {0: 1.1, 1: 1.2, 2: 1.0, 3: 1.0, 4: 1.1}  # Ajustar según necesidad

# Entrenamiento
history = model.fit(
    train_generator,
    epochs=40,
    validation_data=val_generator,
    callbacks=callbacks,
    class_weight=class_weights,
    verbose=1
)
```

## # Resultados:

```
/usr/local/lib/python3.11/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessi
  self._warn_if_super_not_called()
Epoch 1/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 120s 2s/step - accuracy: 0.3013 - loss: 6.8029 - precision: 0.4676 - recall: 0.2068 - val_accuracy: 0.6520 - val_loss: 5.0617 - val_precision: 0.7264 - val_recall: 0.5753 - learning_rate: 1.0000e-04
Epoch 2/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 85s 2s/step - accuracy: 0.7472 - loss: 4.8525 - precision: 0.7996 - recall: 0.6869 - val_accuracy: 0.7053 - val_loss: 4.8516 - val_precision: 0.7421 - val_recall: 0.6680 - learning_rate: 1.0000e-04
Epoch 3/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 84s 2s/step - accuracy: 0.7896 - loss: 4.5778 - precision: 0.8316 - recall: 0.7536 - val_accuracy: 0.7033 - val_loss: 4.7851 - val_precision: 0.7271 - val_recall: 0.6767 - learning_rate: 1.0000e-04
Epoch 4/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 142s 2s/step - accuracy: 0.8469 - loss: 4.2942 - precision: 0.8774 - recall: 0.8240 - val_accuracy: 0.7213 - val_loss: 4.6908 - val_precision: 0.7405 - val_recall: 0.7000 - learning_rate: 1.0000e-04
Epoch 5/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 84s 2s/step - accuracy: 0.8811 - loss: 4.0761 - precision: 0.9051 - recall: 0.8579 - val_accuracy: 0.7507 - val_loss: 4.5112 - val_precision: 0.7614 - val_recall: 0.7340 - learning_rate: 1.0000e-04
Epoch 6/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 142s 2s/step - accuracy: 0.9080 - loss: 3.8412 - precision: 0.9266 - recall: 0.8879 - val_accuracy: 0.7480 - val_loss: 4.3702 - val_precision: 0.7589 - val_recall: 0.7367 - learning_rate: 1.0000e-04
Epoch 7/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 84s 2s/step - accuracy: 0.9104 - loss: 3.6715 - precision: 0.9246 - recall: 0.8974 - val_accuracy: 0.7440 - val_loss: 4.3421 - val_precision: 0.7517 - val_recall: 0.7367 - learning_rate: 1.0000e-04
Epoch 8/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 84s 2s/step - accuracy: 0.9231 - loss: 3.4859 - precision: 0.9343 - recall: 0.9119 - val_accuracy: 0.7320 - val_loss: 4.1986 - val_precision: 0.7420 - val_recall: 0.7227 - learning_rate: 1.0000e-04
Epoch 9/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 143s 2s/step - accuracy: 0.9353 - loss: 3.3116 - precision: 0.9432 - recall: 0.9224 - val_accuracy: 0.7480 - val_loss: 4.0865 - val_precision: 0.7599 - val_recall: 0.7407 - learning_rate: 1.0000e-04
Epoch 10/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 85s 2s/step - accuracy: 0.9391 - loss: 3.1353 - precision: 0.9483 - recall: 0.9287 - val_accuracy: 0.7580 - val_loss: 3.9390 - val_precision: 0.7616 - val_recall: 0.7540 - learning_rate: 1.0000e-04
Epoch 11/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 86s 2s/step - accuracy: 0.9433 - loss: 2.9684 - precision: 0.9493 - recall: 0.9336 - val_accuracy: 0.7740 - val_loss: 3.6963 - val_precision: 0.7834 - val_recall: 0.7667 - learning_rate: 1.0000e-04
Epoch 12/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 85s 2s/step - accuracy: 0.9537 - loss: 2.7867 - precision: 0.9599 - recall: 0.9480 - val_accuracy: 0.7760 - val_loss: 3.5712 - val_precision: 0.7830 - val_recall: 0.7673 - learning_rate: 1.0000e-04
Epoch 13/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 85s 2s/step - accuracy: 0.9578 - loss: 2.6488 - precision: 0.9617 - recall: 0.9529 - val_accuracy: 0.7787 - val_loss: 3.4438 - val_precision: 0.7861 - val_recall: 0.7767 - learning_rate: 1.0000e-04
Epoch 14/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 143s 2s/step - accuracy: 0.9684 - loss: 2.4720 - precision: 0.9732 - recall: 0.9637 - val_accuracy: 0.7760 - val_loss: 3.3728 - val_precision: 0.7832 - val_recall: 0.7660 - learning_rate: 1.0000e-04
Epoch 15/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 83s 2s/step - accuracy: 0.9674 - loss: 2.3451 - precision: 0.9698 - recall: 0.9641 - val_accuracy: 0.7767 - val_loss: 3.2238 - val_precision: 0.7829 - val_recall: 0.7693 - learning_rate: 1.0000e-04
Epoch 16/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 83s 2s/step - accuracy: 0.9705 - loss: 2.2166 - precision: 0.9742 - recall: 0.9643 - val_accuracy: 0.7767 - val_loss: 3.1095 - val_precision: 0.7815 - val_recall: 0.7700 - learning_rate: 1.0000e-04
Epoch 17/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 85s 2s/step - accuracy: 0.9740 - loss: 2.0809 - precision: 0.9764 - recall: 0.9698 - val_accuracy: 0.7413 - val_loss: 3.2796 - val_precision: 0.7463 - val_recall: 0.7353 - learning_rate: 1.0000e-04
Epoch 18/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 84s 2s/step - accuracy: 0.9731 - loss: 1.9547 - precision: 0.9766 - recall: 0.9705 - val_accuracy: 0.7620 - val_loss: 3.1599 - val_precision: 0.7691 - val_recall: 0.7573 - learning_rate: 1.0000e-04
Epoch 19/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 83s 2s/step - accuracy: 0.9786 - loss: 1.8278 - precision: 0.9830 - recall: 0.9766 - val_accuracy: 0.7633 - val_loss: 2.9229 - val_precision: 0.7675 - val_recall: 0.7593 - learning_rate: 1.0000e-04
Epoch 20/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 85s 2s/step - accuracy: 0.9770 - loss: 1.7258 - precision: 0.9793 - recall: 0.9756 - val_accuracy: 0.7587 - val_loss: 2.9659 - val_precision: 0.7663 - val_recall: 0.7540 - learning_rate: 1.0000e-04
Epoch 21/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 84s 2s/step - accuracy: 0.9807 - loss: 1.6205 - precision: 0.9813 - recall: 0.9785 - val_accuracy: 0.7660 - val_loss: 2.7787 - val_precision: 0.7696 - val_recall: 0.7640 - learning_rate: 1.0000e-04
Epoch 22/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 83s 2s/step - accuracy: 0.9810 - loss: 1.5091 - precision: 0.9822 - recall: 0.9793 - val_accuracy: 0.7787 - val_loss: 2.6721 - val_precision: 0.7844 - val_recall: 0.7760 - learning_rate: 1.0000e-04
Epoch 23/40
55/55 ━━━━━━━━━━━━━━━━━━━━ 100s 2s/step - accuracy: 0.9878 - loss: 1.4040 - precision: 0.9887 - recall: 0.9871 - val_accuracy: 0.7753 - val_loss: 2.5336 - val_precision: 0.7815 - val_recall: 0.7727 - learning_rate: 1.0000e-04
Epoch 23: early stopping
Restoring model weights from the end of the best epoch: 13.
```

## A. Desempeño general

- **Épocas ejecutadas**: 23 (de 40 programadas).
- **Motivo de parada temprana**: No hubo mejora en val_precision durante 10 épocas seguidas.
- **Mejor modelo**: Se restauraron los pesos de la **época 13**, donde se alcanzó el mejor equilibrio entre precisión y recall en validación.

## B. Métricas clave del mejor modelo (Época 13):

| **Métrica** | **Entrenamiento** | **Validación** |
|---|---|---|
| Precisión (accuracy) | 95.8% | 77.9% |
| Pérdida (loss) | 2.65 | 3.44 |
| Recall | 95.3% | 77.7% |

## C. Observaciones

a. **Sobreajuste**:

- La precisión en entrenamiento subió hasta **98.7%** (época 23), pero la precisión en validación se estancó en **~77%**.
- La pérdida en entrenamiento bajó drásticamente (de 6.0 a 1.4), mientras que la pérdida en validación solo bajó de 5.06 a 2.53.

b. **Recall vs. Precisión**:

- En validación, el recall máximo fue **77.6%** (época 22), ligeramente por debajo de la precisión, lo que sugiere que el modelo tiene más falsos negativos que falsos positivos.

## Parte 5: Visualización Gráfica de los Resultados

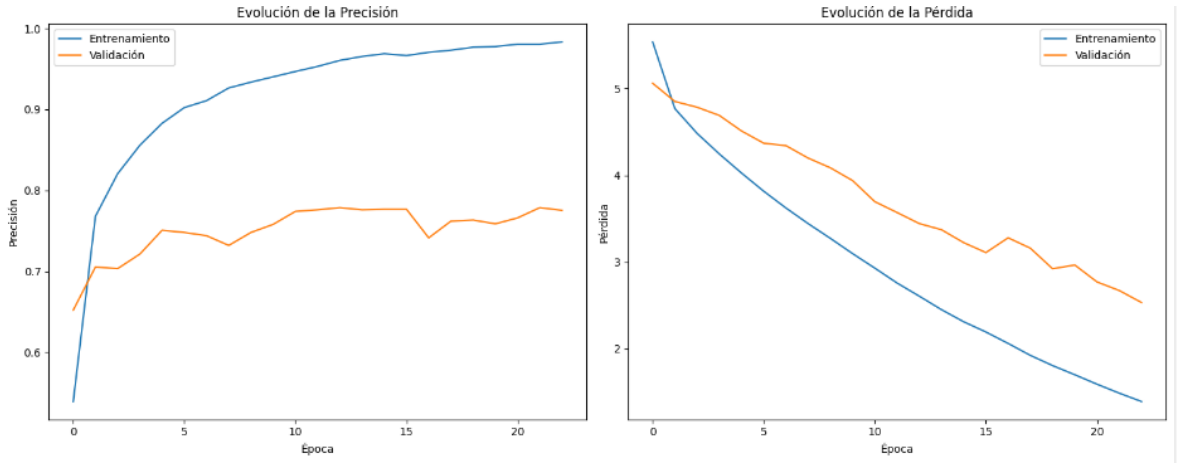
```
[ ]  def plot_training_history(history):
         plt.figure(figsize=(15, 6))

         # Gráfico de precisión
         plt.subplot(1, 2, 1)
         plt.plot(history.history['accuracy'], label='Entrenamiento')
         plt.plot(history.history['val_accuracy'], label='Validación')
         plt.title('Evolución de la Precisión')
         plt.ylabel('Precisión')
         plt.xlabel('Época')
         plt.legend()

         # Gráfico de pérdida
         plt.subplot(1, 2, 2)
         plt.plot(history.history['loss'], label='Entrenamiento')
         plt.plot(history.history['val_loss'], label='Validación')
         plt.title('Evolución de la Pérdida')
         plt.ylabel('Pérdida')
         plt.xlabel('Época')
         plt.legend()

         plt.tight_layout()
         plt.show()

     plot_training_history(history)
```




Recapitulando y enlazando los resultados del modelo de clasificación, tenemos que se entrenó durante 23 épocas de las 40 programadas, deteniéndose anticipadamente debido a la falta de mejora en la precisión de validación durante 10 épocas consecutivas. El mejor rendimiento se obtuvo en la época 13, con una precisión de validación del 77.9%, una pérdida

de 3.44 y un recall del 77.7%, indicando un equilibrio razonable entre detectar objetos correctamente y minimizar errores.

Se observó sobreajuste, mientras la precisión en entrenamiento alcanzó hasta el 98.7% (época 23), la precisión en validación se estancó en torno al 77-78%. La pérdida en entrenamiento disminuyó drásticamente (de 6.0 a 1.4), pero en validación solo se redujo a la mitad (de 5.06 a 2.53), lo que sugiere que el modelo memorizó patrones específicos de los datos de entrenamiento en lugar de aprender características generalizables.

El modelo final priorizó el recall, o capacidad de detectar positivos reales, especialmente en clases críticas como el glass, gracias a los pesos de clase ajustados.

```
# Generar reporte de clasificación
val_preds = model.predict(val_generator)
val_preds = np.argmax(val_preds, axis=1)

print("\n Reporte de Clasificación:")
print(classification_report(val_generator.classes, val_preds, target_names=val_generator.class_indices.keys()))
```

```
24/24 ━━━━━━━━━━━━━━━━━━━━ 37s 1s/step

 Reporte de Clasificación:
              precision    recall  f1-score   support

   cardboard       0.75      0.86      0.80       300
       glass       0.85      0.60      0.71       300
       metal       0.69      0.84      0.76       300
       paper       0.91      0.71      0.80       300
     plastic       0.73      0.84      0.78       300

    accuracy                           0.77      1500
   macro avg       0.79      0.77      0.77      1500
weighted avg       0.79      0.77      0.77      1500
```

**# Resumen del Reporte de Clasificación:**

Por lo que modelo alcanzó un **77%** de precisión general (accuracy) en el conjunto de validación, con un rendimiento variable entre las cinco clases analizadas. La clase glass presentó la mayor debilidad: aunque logró una precisión del 85% (solo el 15% de

sus predicciones fueron incorrectas), su recall fue del 60%, lo que indica que el modelo falló en detectar el 40% de las muestras reales de vidrio.

Por otro lado, cardboard y plastic mostraron un equilibrio robusto, con recalls del 86% y 84%, respectivamente, lo que implica que el modelo detecta eficientemente estos materiales. Paper destacó por su alta precisión (91%), pero su recall fue modesto (71%), revelando que, aunque casi no comete errores al predecir papel, ignora una parte significativa de muestras reales. Metal, aunque con una precisión moderada (69%), logró un recall del 84%, similar a las clases mejor desempeñadas.

Las métricas promedio (79% precisión macro, 77% recall macro) confirman que el modelo no está sesgado hacia clases específicas, pero su capacidad para generalizar se ve limitada por desafíos en la detección de vidrio y papel. En su estado actual, el modelo es funcional para clasificar la mayoría de los residuos, pero requiere optimizaciones para escenarios donde omitir objetos reciclables sea inaceptable.

## Conclusión:

Se concluye que esta iniciativa tiene el potencial de revolucionar el reciclaje al ir paulatinamente sumando a la robustez en la clasificación del modelo y así ir reduciendo los errores humanos que implica esta labor y su vez acelerar este proceso de forma humana, haciendo accesible la capacitación básica para lograrlo, integrando en el funcionamiento de la otra parte del programa (interfaz gráfica de usuario), información oportuna y confiable sobre el reciclaje y en cualquier caso llevar a cabo una separación más precisa de materiales, lo que facilitará su adopción y contribuirá en la transformación en donde quiera que se le implemente.

En una realidad mundial donde la crisis ambiental exige soluciones innovadoras, proyectos como EcoClass reflejan el poder de la tecnología para transformar estos desafíos globales en oportunidades tangibles y al alcance de todos. La inteligencia artificial no solo optimiza procesos técnicos, sino que también inspira un cambio colectivo hacia la sostenibilidad.

Es importante enfatizar que este camino requiere equilibrar el avance tecnológico con la conciencia social, por lo que esta es la razón de ser de este informe. Donde se prevé que cada próxima mejora en el modelo, irá acompañada de educación ambiental, políticas claras y colaboración comunitaria. En general la clasificación de residuos es un eslabón clave en la cadena del reciclaje, y al fortalecerlo con estas herramientas, no solo cuidamos el planeta, sino que construimos este futuro hoy, donde la innovación y la responsabilidad ambiental convergen, regalándonos entre todos el hogar que queremos.

Desde Light of Hope - EcoClass: **You decide what kind of difference you make.**