

Deteccion de enfermedades en plantas de Frutillas con Yolo-v5

TECNICATURA SUPERIOR EN CIENCIA DE DATOS E INTELIGENCIA ARTIFICIAL

AUTOR: Marcelo E. Mamani



24 de noviembre de 2024

I.E.S. N° 6 “INSTITUTO DE EDUCACION SUPERIOR N° 6

Y4610 Perico, Jujuy

**Índice del Modelo de Entrenamiento YOLOv5**

1. **Introducción**
   * 1.1. ¿Qué es YOLOv5?
   * 1.2. Objetivo del Proyecto
   * 1.3. Requisitos Previos
2. **Preparación del Entorno**
   * 2.1. Montaje de Google Drive
   * 2.2. Descarga e Instalación de YOLOv5
   * 2.3. Configuración de Rutas
3. **Preparación del Dataset**
   * 3.1. Estructura del Dataset
   * 3.2. Generación de Anotaciones
   * 3.3. Creación de Máscaras
   * 3.4. Validación de Datos
4. **Transformaciones y Preprocesamiento**
   * 4.1. Normalización de Imágenes
   * 4.2. Redimensionamiento a la Resolución Esperada
   * 4.3. Aplicación de Transformaciones a Máscaras
5. **Creación del Dataset Personalizado**
   * 5.1. Introducción a PyTorch Dataset
   * 5.2. Implementación de la Clase StrawberryDataset
   * 5.3. Manejo de Archivos JSON para Anotaciones
6. **Diseño y Entrenamiento del Modelo**
   * 6.1. Selección del Modelo YOLOv5
   * 6.2. Configuración de Parámetros de Entrenamiento
   * 6.3. Definición de la Función de Pérdida y el Optimizador
   * 6.4. Proceso de Entrenamiento
   * 6.5. Evaluación en el Conjunto de Validación
7. **Depuración y Resolución de Problemas**
   * 7.1. Manejo de Errores en las Máscaras
   * 7.2. Solución a Inconsistencias en las Anotaciones
   * 7.3. Ajustes en los Parámetros del Modelo
8. **Resultados y Evaluación**
   * 8.1. Métricas de Evaluación Utilizadas
   * 8.2. Análisis de Resultados del Modelo
   * 8.3. Visualización de Predicciones
9. **Implementación del Modelo en Producción**
   * 9.1. Exportación del Modelo Entrenado
   * 9.2. Uso del Modelo en Nuevas Imágenes
   * 9.3. Optimización para Inferencia en Tiempo Real
10. **Apéndices**
    * 10.1. Código Completo del Entrenamiento
    * 10.2. Referencias y Recursos
    * 10.3. Glosario de Términos

**Miembros del Proyecto:**

**Marcelo Mamani, Armando Flores, Carla Salazar**

**RCONOCIMIENTO DE LETRAS MEDIANTE GESTOS DE MANO  
Fecha de presentación: 25 noviembre de 2024  
Curso: 3° AÑO  
Profesor: Alfredo Sardinas**

|  |
| --- |
| “**La inteligencia artificial es la nueva electricidad.**” — **Andrew Ng**, profesor en Stanford y cofundador de Google Brain.  Andrew Ng, uno de los pioneros en el campo de la inteligencia artificial, utiliza esta cita para subrayar el enorme impacto y potencial transformador que tiene la inteligencia artificial en diversas áreas, incluyendo la visión por computadora y el aprendizaje automático. Al igual que la electricidad revolucionó la industria, la IA está configurándose como la herramienta que impulsará nuevas innovaciones, como el entrenamiento de modelos eficientes como YOLOv5. |

1. **Introducción**

El presente manual está diseñado para guiar al lector a través del proceso de entrenamiento de un modelo basado en YOLOv5, una de las arquitecturas más avanzadas para la detección de objetos en tiempo real. Este proyecto se enfoca en el reconocimiento de enfermedades en frutillas a través de imágenes, utilizando técnicas de aprendizaje profundo para identificar patrones y anomalías de manera eficiente.

**1.1. ¿Qué es YOLOv5?**

YOLOv5 (You Only Look Once, versión 5) es un modelo de detección de objetos altamente optimizado para tareas en tiempo real. Diseñado por **Ultralytics**, es capaz de detectar múltiples objetos en una imagen con alta precisión y velocidad. A diferencia de otras versiones, YOLOv5 es:

* **Modular y escalable:** Su arquitectura permite adaptarse a diferentes tamaños de modelos, desde versiones ligeras para dispositivos móviles hasta variantes más grandes para aplicaciones avanzadas.
* **Fácil de usar:** Integra herramientas modernas de Python y PyTorch, lo que simplifica su configuración y entrenamiento.
* **Eficiente:** Gracias a sus mejoras, puede realizar inferencias rápidas con menor consumo de recursos computacionales.

En este proyecto, YOLOv5 se adapta para identificar enfermedades específicas en frutillas mediante un conjunto de imágenes anotadas con máscaras y etiquetas.

**1.2. Objetivo del Proyecto**

El objetivo principal es entrenar un modelo de detección de objetos capaz de:

1. **Identificar enfermedades en frutillas:** Clasificar las imágenes según la presencia de patologías y determinar su severidad (por ejemplo, *leve* o *grave*).
2. **Optimizar el proceso de clasificación:** Garantizar que el modelo pueda realizar detecciones rápidas y precisas, facilitando su integración en entornos agrícolas o de investigación.
3. **Automatizar el análisis visual:** Reducir la dependencia de la inspección manual, ofreciendo una herramienta confiable para los productores.

**1.3. Requisitos Previos**

Antes de iniciar, asegúrate de cumplir con los siguientes requisitos técnicos y de conocimiento:

1. **Hardware:**
   * Acceso a una GPU (recomendado: Google Colab con GPU habilitada).
   * Espacio suficiente en Google Drive para almacenar el dataset y los resultados del entrenamiento.
2. **Software:**
   * Python 3.7 o superior.
   * Librerías esenciales: PyTorch, torchvision, OpenCV, matplotlib, entre otras (instaladas a través de los requisitos de YOLOv5).
   * Google Colab como entorno principal de desarrollo.
3. **Conocimientos básicos:**
   * Fundamentos de aprendizaje profundo y redes neuronales.
   * Familiaridad con PyTorch y el flujo de trabajo de un modelo de detección de objetos.
   * Manejo básico de Google Drive para organizar datasets.
4. **Dataset:**
   * Imágenes organizadas en carpetas según entrenamiento, validación y prueba.
   * Archivos JSON con anotaciones que describen las áreas afectadas en cada imagen.

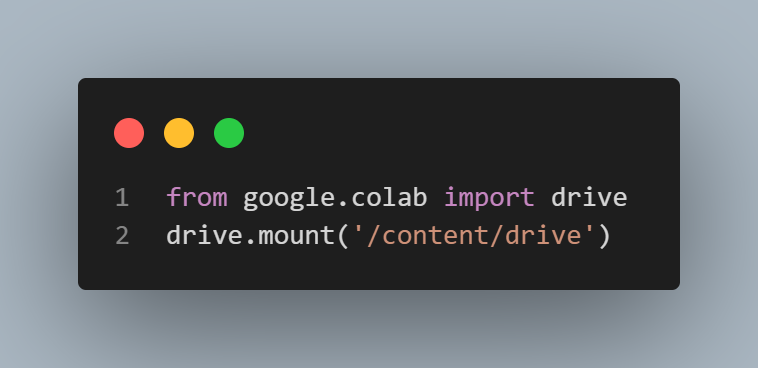
**2. Preparación del Entorno**

En esta sección se detalla el proceso para preparar el entorno de desarrollo necesario para entrenar el modelo YOLOv5. Se incluye el montaje de Google Drive, la descarga e instalación de YOLOv5 y la configuración de las rutas requeridas.

**2.1. Montaje de Google Drive**

Google Drive se utiliza para almacenar el conjunto de datos, las anotaciones y los resultados del entrenamiento. Para conectar tu espacio de Google Drive con Google Colab, sigue los pasos a continuación:

1. Monta Google Drive en el entorno de Colab ejecutando el siguiente código:



Se solicitará acceso a tu cuenta de Google. Concede los permisos necesarios.

1. Una vez montado, podrás acceder a los archivos y carpetas de tu Google Drive desde la ruta /content/drive/MyDrive.

**Nota:** Asegúrate de organizar los archivos de datos dentro de Google Drive antes de proceder. Una estructura recomendada es:

MyDrive/

├── Strawberry Disease Detection Dataset-Usman afzaal/

│ ├── train/

│ │ ├── images/

│ │ └── masks/

│ ├── val/

│ │ ├── images/

│ │ └── masks/

│ └── test/

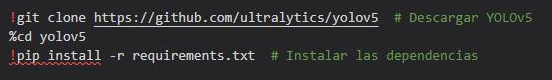
│ ├── images/

│ └── masks/

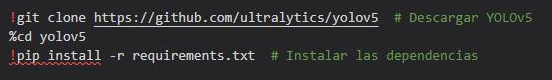
**2.2. Descarga e Instalación de YOLOv5**

YOLOv5 debe instalarse en el entorno de Google Colab para poder entrenar el modelo. Sigue estos pasos:

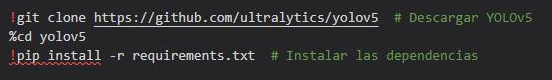
1. Clona el repositorio oficial de YOLOv5 desde GitHub:



1. Cambia al directorio del proyecto YOLOv5:



1. Instala las dependencias requeridas ejecutando el archivo requirements.txt:



1. Verifica la instalación ejecutando:

!python train.py --help

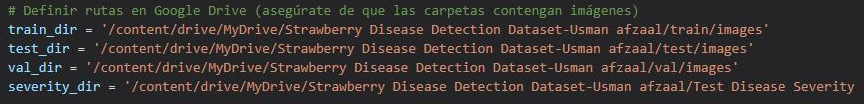
Este comando mostrará una lista de opciones y parámetros disponibles para el entrenamiento.

**Recomendación:** Mantén actualizadas las dependencias para garantizar compatibilidad con las últimas versiones de PyTorch y otras bibliotecas.

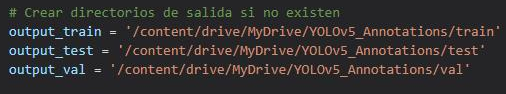
**2.3. Configuración de Rutas**

Para que YOLOv5 pueda acceder a los datos de entrenamiento, validación y prueba, es necesario definir las rutas adecuadas en el código.

1. Define las rutas de las carpetas que contienen las imágenes y anotaciones en Google Drive:



1. Crea directorios para almacenar las anotaciones convertidas (si es necesario):



import os



**3. Preparación del Dataset**

El dataset es un componente fundamental para entrenar un modelo YOLOv5. En esta sección, se explica cómo estructurar el dataset, generar anotaciones, crear máscaras y validar los datos para garantizar su calidad.

**3.1. Estructura del Dataset**

El dataset debe organizarse en carpetas claramente separadas para entrenamiento, validación y prueba, siguiendo una estructura estándar. En este caso, se utiliza el conjunto de datos *Strawberry Disease Detection Dataset*. A continuación se muestra una estructura recomendada:

Bash:

Dataset/

├── train/

│ ├── images/ # Imágenes para entrenamiento

│ └── masks/ # Anotaciones correspondientes (en formato JSON)

├── val/

│ ├── images/ # Imágenes para validación

│ └── masks/ # Anotaciones correspondientes

└── test/

├── images/ # Imágenes para prueba

└── masks/ # Anotaciones correspondientes

**Puntos importantes:**

* Las imágenes deben estar en formato .jpg o .png.
* Las anotaciones deben estar en formato .json, conteniendo las coordenadas de los polígonos que delimitan los objetos.

**3.2. Generación de Anotaciones**

Las anotaciones proporcionan la información necesaria para que el modelo aprenda a identificar los objetos. Estas anotaciones suelen venir en formato JSON con estructuras como esta:

Json:

{

"shapes": [

{

"label": "strawberry",

"points": [[x1, y1], [x2, y2], ...],

"group\_id": null,

"shape\_type": "polygon",

"flags": {}

}

],

"imagePath": "example.jpg",

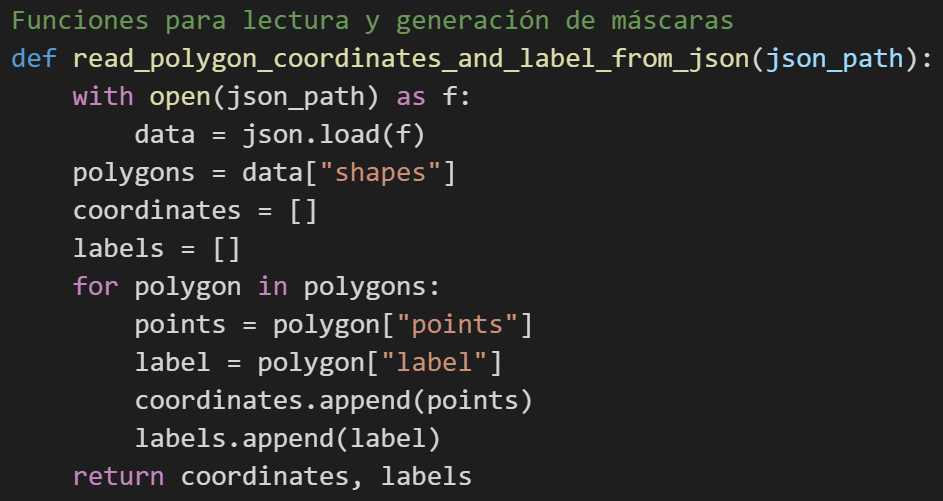
"imageWidth": 640,

"imageHeight": 480

}

**Lectura de Anotaciones:** Para cargar y procesar las anotaciones:

import json



Esta función extrae las coordenadas de los polígonos y las etiquetas asociadas desde los archivos JSON.

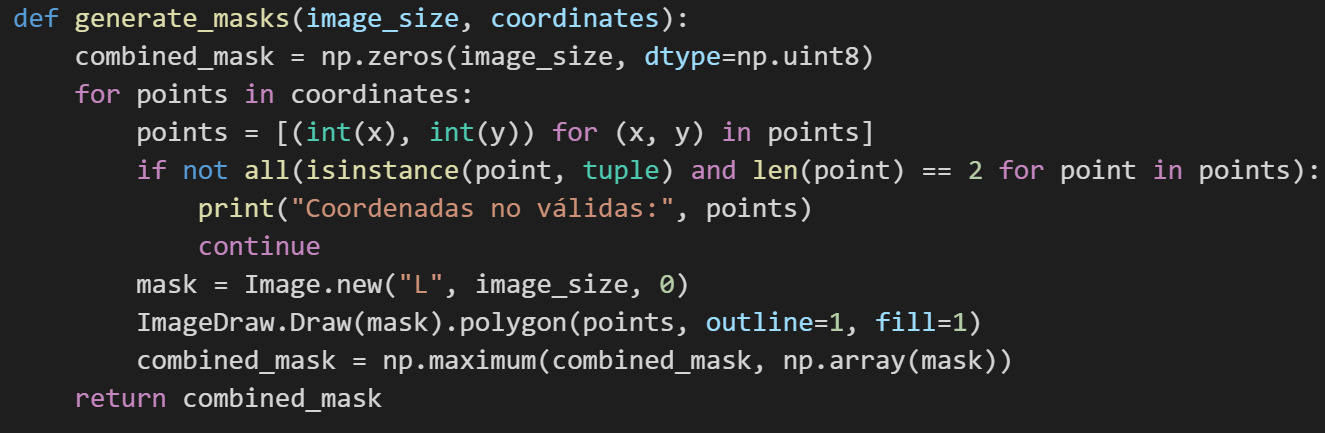
**3.3. Creación de Máscaras**

Las máscaras binarias son esenciales para representar las áreas donde están los objetos en las imágenes. Se generan utilizando las coordenadas de los polígonos presentes en las anotaciones JSON.

**Generación de Máscaras:**

import numpy as np

from PIL import Image, ImageDraw



**3.4. Validación de Datos**

Antes de iniciar el entrenamiento, es crucial asegurarse de que todas las imágenes tengan sus correspondientes máscaras y de que no haya datos faltantes o corruptos.

**Validación de Imágenes y Máscaras:**

import os



Al finalizar esta etapa, se tendrá un dataset organizado, anotaciones procesadas, máscaras generadas y datos validados, listo para iniciar el entrenamiento del modelo YOLOv5.

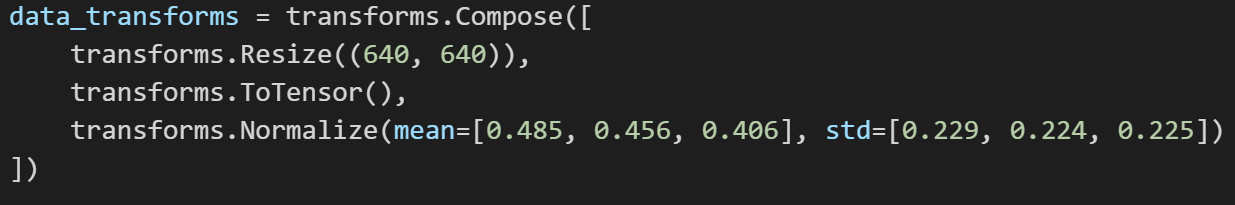
**4. Transformaciones y Preprocesamiento**

El preprocesamiento de los datos es crucial para garantizar que el modelo YOLOv5 reciba las imágenes y las máscaras en un formato compatible. Esta etapa incluye normalización de imágenes, redimensionamiento a la resolución esperada y aplicación de transformaciones a las máscaras.

**4.1. Normalización de Imágenes**

La normalización ajusta los valores de los píxeles de las imágenes a un rango estándar (generalmente entre 0 y 1) y aplica estadísticas de normalización basadas en los valores medios y desviaciones estándar de los canales RGB. Esto mejora la estabilidad del modelo durante el entrenamiento.

**Transformación de Normalización:**



**Ejemplo de Uso:**

from PIL import Image

image\_path = '/path/to/image.jpg'

image = Image.open(image\_path).convert("RGB")

normalized\_image = data\_transforms(image)

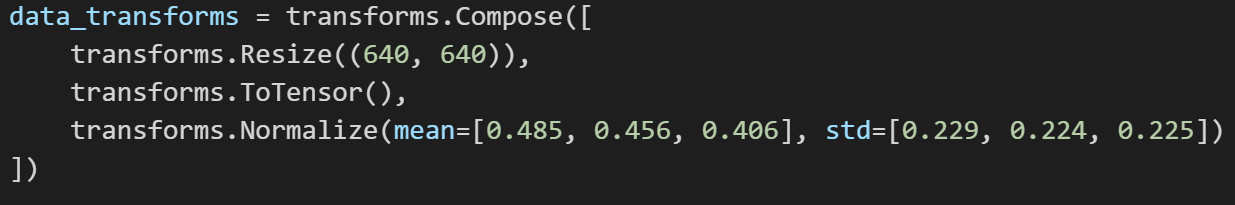
**Explicación:**

* **transforms.ToTensor()**: Convierte la imagen a un tensor PyTorch.
* **transforms.Normalize()**: Ajusta los valores del tensor basándose en las medias y desviaciones estándar de los canales RGB.

**4.2. Redimensionamiento a la Resolución Esperada**

YOLOv5 requiere que todas las imágenes tengan un tamaño fijo, comúnmente de 640x640 píxeles. Las imágenes deben redimensionarse manteniendo la relación de aspecto o mediante un recorte y escalado uniforme.

**Transformación de Redimensionamiento:**



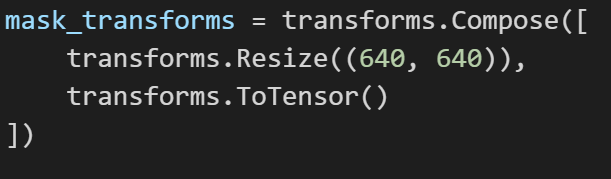
**Consideraciones:**

* Redimensionar todas las imágenes al mismo tamaño es crucial para un entrenamiento eficiente.
* Usar **transforms.Resize()** asegura la compatibilidad con el modelo.

**4.3. Aplicación de Transformaciones a Máscaras**

Las máscaras también deben transformarse para que coincidan con las dimensiones de las imágenes y estén en un formato compatible con PyTorch.

**Transformación de Máscaras:**



**Resumen del Flujo de Preprocesamiento**

1. **Cargar la imagen y la máscara original.**
2. **Aplicar transforms.Resize() a ambas para redimensionarlas a 640x640 píxeles.**
3. **Normalizar la imagen con transforms.Normalize().**
4. **Convertir tanto la imagen como la máscara a tensores mediante transforms.ToTensor().**

Este flujo garantiza que los datos estén listos para ser procesados por el modelo durante el entrenamiento.

**5. Creación del Dataset Personalizado**

**5.1. Introducción a PyTorch Dataset**

PyTorch proporciona la clase Dataset, que se utiliza para cargar datos de manera flexible. La implementación personalizada de un Dataset permite definir cómo se acceden los datos, incluyendo imágenes, anotaciones y máscaras.

**Estructura de la Clase Dataset:**

* **\_\_init\_\_**: Inicializa las rutas y transforma las imágenes y máscaras.
* **\_\_len\_\_**: Devuelve la cantidad total de muestras en el dataset.
* **\_\_getitem\_\_**: Define cómo se obtiene una muestra individual del dataset.

**5.2. Implementación de la Clase StrawberryDataset**

La clase StrawberryDataset se diseñó para trabajar con un conjunto de datos que contiene imágenes de fresas con enfermedades, máscaras de las áreas afectadas y anotaciones en formato JSON.

**Código de Implementación:**

from torch.utils.data import Dataset

from PIL import Image, ImageDraw

import json

import os



**5.3. Manejo de Archivos JSON para Anotaciones**

Las anotaciones JSON contienen los polígonos que delimitan las áreas afectadas en las imágenes.

**Ejemplo de Estructura de un Archivo JSON:**

Json:

{

"shapes": [

{

"label": "disease",

"points": [[10, 20], [30, 40], [50, 20]]

}

]

}

**Procesamiento del JSON:**

* Cada anotación incluye una lista de puntos para dibujar polígonos.
* Estos polígonos se convierten en máscaras binarias.

**6. Diseño y Entrenamiento del Modelo**

**6.1. Selección del Modelo YOLOv5**

El modelo YOLOv5 se selecciona por su rendimiento y facilidad de integración. Para este proyecto, se utiliza la configuración base **yolov5s** por su balance entre velocidad y precisión.

**6.2. Configuración de Parámetros de Entrenamiento**

* **Épocas**: Número total de iteraciones sobre el dataset (por ejemplo, 60).
* **Batch size**: Número de imágenes procesadas simultáneamente (por ejemplo, 10).
* **Dispositivo**: Entrenamiento en GPU para mayor eficiencia.

**Definición de Parámetros:**

python

Copiar código

total\_epochs = 60

batch\_size = 10

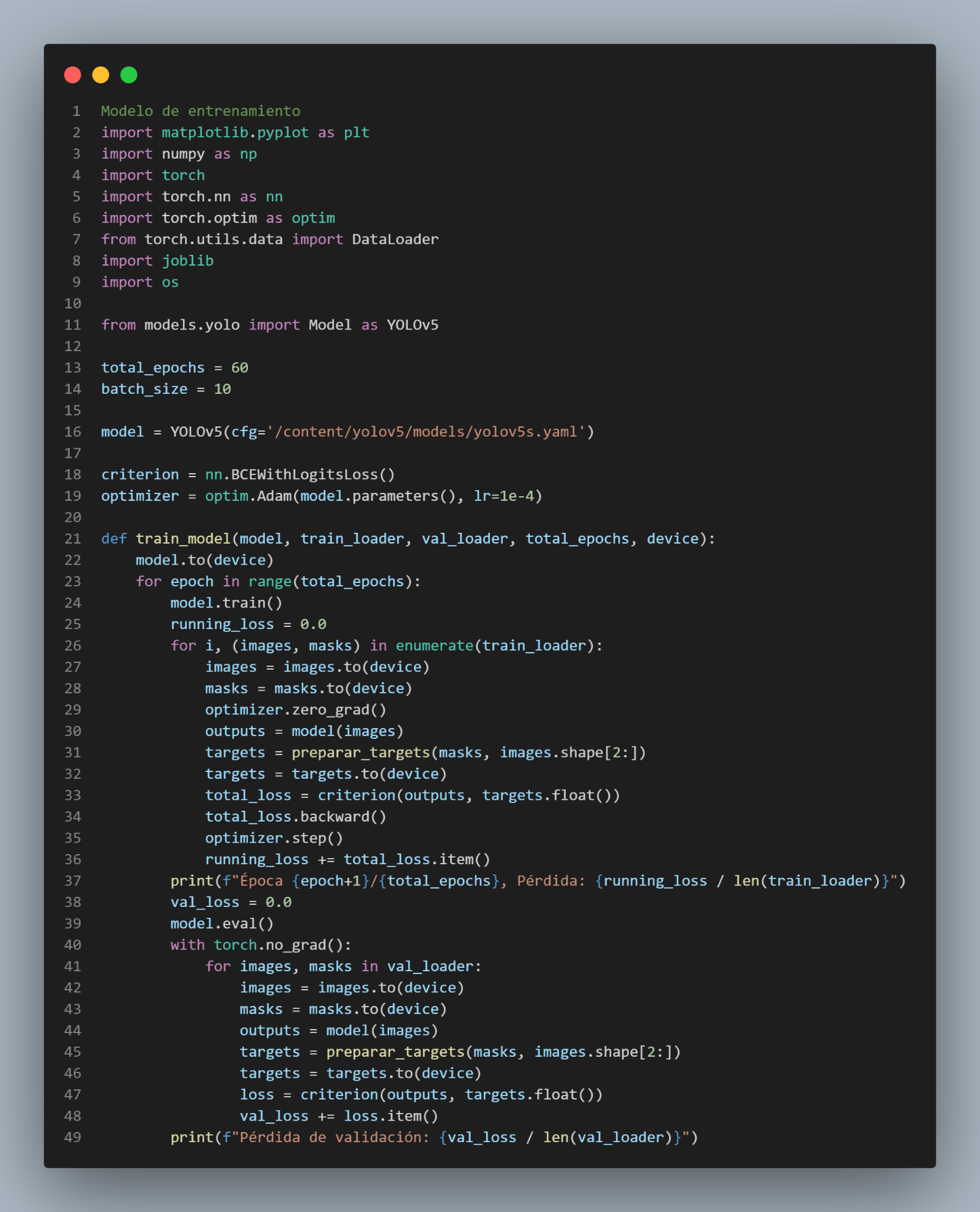
device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

**6.3. Definición de la Función de Pérdida y el Optimizador**

* **Función de pérdida**: BCEWithLogitsLoss para detección binaria.
* **Optimizador**: Adam para actualizar los pesos del modelo.

**6.4. Proceso de Entrenamiento**

El proceso de entrenamiento ajusta los pesos del modelo para minimizar la pérdida. En cada época, se evalúa el rendimiento en los conjuntos de entrenamiento y validación.



**6.2. Configuración de Parámetros de Entrenamiento**

**6.3. Definición de la Función de Pérdida y el Optimizador**

**6.4. Proceso de Entrenamiento**

**6.1. Selección del Modelo YOLOv5**

**6.5. Evaluación en el Conjunto de Validación**

El rendimiento del modelo se evalúa calculando la pérdida en el conjunto de validación. Esto asegura que el modelo generalice correctamente.

**Evaluación:**

def evaluate\_model(model, val\_loader, device):

model.eval()

val\_loss = 0.0

with torch.no\_grad():

for images, masks in val\_loader:

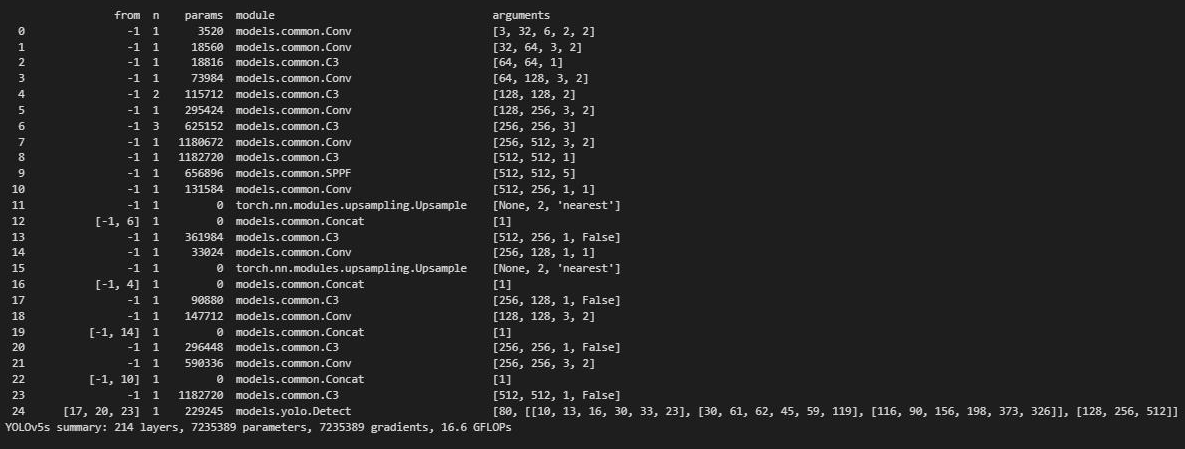
images, masks = images.to(device), masks.to(device)

outputs = model(images)

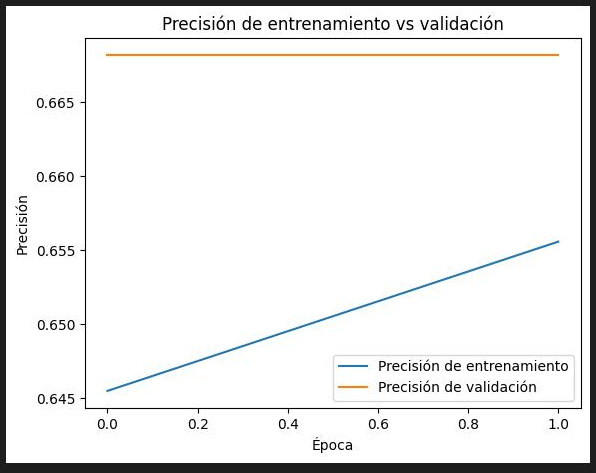
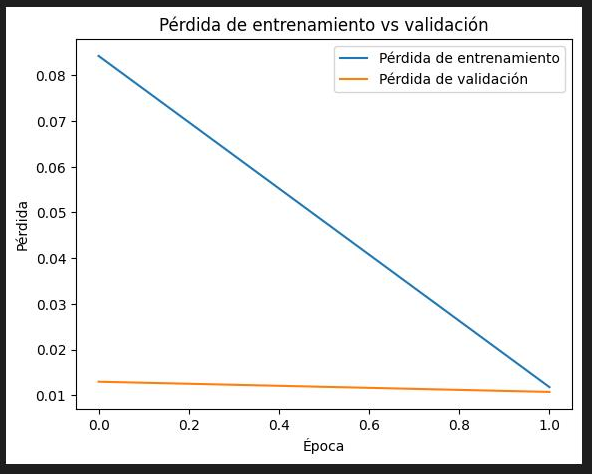
loss = criterion(outputs, masks.float())

val\_loss += loss.item()

print(f"Pérdida de validación: {val\_loss / len(val\_loader)}")



**7. Depuración y Resolución de Problemas**



**7.1. Manejo de Errores en las Máscaras**

Las máscaras pueden presentar problemas como dimensiones incorrectas o datos corruptos. Para solucionar estos problemas:

* **Validación Automática**: Asegúrate de que las máscaras tengan las mismas dimensiones que las imágenes.

if mask.size != image.size:

mask = mask.resize(image.size, resample=Image.NEAREST)

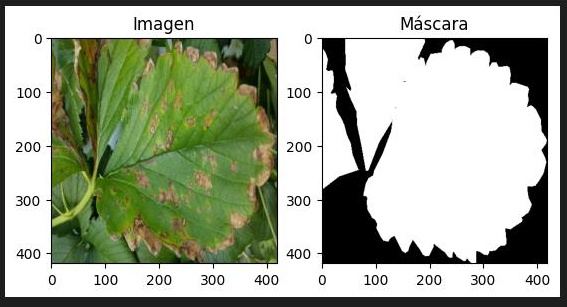
* **Corrección de Polígonos Malformados**: Verifica que los polígonos en los JSON no contengan puntos repetidos o coordenadas fuera de rango.
* **Pruebas de Visualización**: Carga las imágenes y máscaras con una librería como matplotlib para identificar visualmente errores.

import matplotlib.pyplot as plt

plt.imshow(image)

plt.imshow(mask, alpha=0.5) # Transparencia para superponer

plt.show()



**7.2. Solución a Inconsistencias en las Anotaciones**

Las anotaciones pueden tener problemas como etiquetas faltantes o puntos de polígonos desordenados.

* **Validación de JSON**: Crea una función para verificar la estructura del archivo JSON.

import json

def validate\_json(json\_file):

try:

with open(json\_file, 'r') as f:

data = json.load(f)

if 'shapes' not in data:

raise ValueError("Falta la clave 'shapes'")

for shape in data['shapes']:

if 'points' not in shape or len(shape['points']) < 3:

raise ValueError("Polígono no válido en el JSON")

return True

except Exception as e:

print(f"Error en el JSON: {e}")

return False

* **Automatización de Correcciones**: Implementa scripts para normalizar las anotaciones, como ajustar los puntos fuera de los límites.

**7.3. Ajustes en los Parámetros del Modelo**

Si el modelo no está aprendiendo correctamente:

1. **Reducción de la Tasa de Aprendizaje**: Disminuir el valor puede estabilizar el entrenamiento.

optimizer = optim.Adam(model.parameters(), lr=1e-5)

1. **Aumento del Número de Épocas**: Si el modelo no converge, entrena por más iteraciones.
2. **Ajuste de la Función de Pérdida**: Cambia la función de pérdida si hay un desequilibrio en las clases.

criterion = nn.CrossEntropyLoss(weight=torch.tensor([0.1, 0.9]).to(device))

1. **Data Augmentation**: Aplica transformaciones adicionales para aumentar la diversidad del dataset:

from torchvision import transforms

transform = transforms.Compose([

transforms.RandomHorizontalFlip(),

transforms.RandomRotation(10),

transforms.ColorJitter(brightness=0.2, contrast=0.2)

])

**8. Resultados y Evaluación**

**8.1. Métricas de Evaluación Utilizadas**

* **Precisión (Precision)**: Proporción de verdaderos positivos sobre el total de predicciones positivas.
* **Recall (Sensibilidad)**: Proporción de verdaderos positivos sobre el total de verdaderos reales.
* **IoU (Intersection over Union)**: Mide el solapamiento entre la predicción y la anotación.

**Cálculo del IoU:**

def calculate\_iou(pred, target):

intersection = (pred & target).sum()

union = (pred | target).sum()

return intersection / union

**8.2. Análisis de Resultados del Modelo**

1. **Confusión entre Clases**: Identifica si el modelo está confundiendo áreas afectadas y no afectadas.
2. **Efecto de los Parámetros**: Analiza cómo afectan las modificaciones en la tasa de aprendizaje, número de épocas y la estructura del dataset.
3. **Errores Comunes**:
   * Predicciones incompletas en áreas afectadas pequeñas.
   * Falsos positivos en regiones complejas.

**8.3. Visualización de Predicciones**

Presentar las predicciones superpuestas a las imágenes originales ayuda a evaluar visualmente el rendimiento.

**Visualización con Matplotlib:**

import matplotlib.pyplot as plt

def visualize\_predictions(image, ground\_truth, prediction):

plt.figure(figsize=(12, 6))

# Imagen original

plt.subplot(1, 3, 1)

plt.title("Imagen Original")

plt.imshow(image)

# Máscara Real

plt.subplot(1, 3, 2)

plt.title("Máscara Real")

plt.imshow(ground\_truth, cmap='gray')

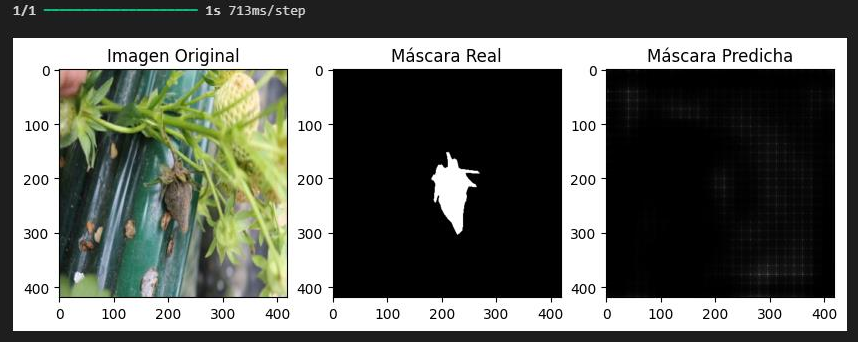
# Predicción del Modelo

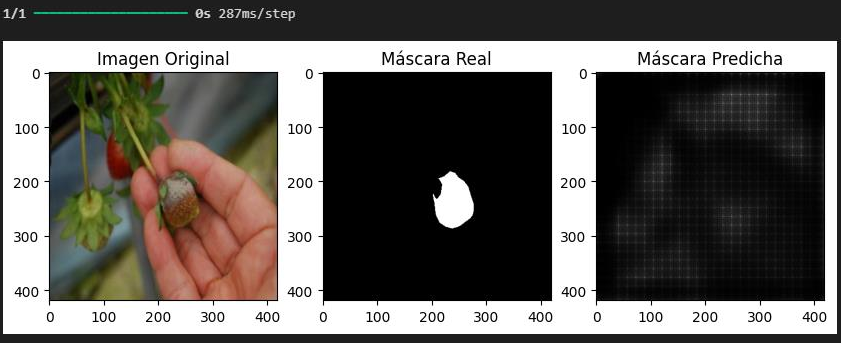
plt.subplot(1, 3, 3)

plt.title("Predicción")

plt.imshow(prediction, cmap='gray')

plt.show()





**Generación de Reportes:**

* Guarda ejemplos destacados donde el modelo funciona bien o presenta problemas.
* Incluye métricas clave en gráficos como curvas ROC o Precision-Recall.

**9. Implementación del Modelo en Producción**

**9.1. Exportación del Modelo Entrenado**

Para usar el modelo fuera del entorno de entrenamiento, es esencial exportarlo a un formato compatible, como TorchScript o ONNX.

* **Exportación a TorchScript**: Permite ejecutar el modelo en diferentes plataformas.

scripted\_model = torch.jit.script(model)

scripted\_model.save("modelo\_entrenado.pt")

* **Exportación a ONNX**: Útil para integraciones con otros frameworks.

dummy\_input = torch.randn(1, 3, 640, 640).to(device) # Según las dimensiones del modelo

torch.onnx.export(

model,

dummy\_input,

"modelo\_entrenado.onnx",

input\_names=["input"],

output\_names=["output"],

dynamic\_axes={"input": {0: "batch\_size"}, "output": {0: "batch\_size"}}

)

**9.2. Uso del Modelo en Nuevas Imágenes**

Una vez exportado, el modelo puede usarse para realizar inferencias en imágenes nuevas.

* **Carga del Modelo Exportado**:

import torch

model = torch.jit.load("modelo\_entrenado.pt")

model.eval()

* **Inferencia en Nuevas Imágenes**: Asegúrate de preprocesar las imágenes según los requisitos del modelo.

from PIL import Image

import torchvision.transforms as transforms

transform = transforms.Compose([

transforms.Resize((640, 640)),

transforms.ToTensor(),

transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

])

image = Image.open("nueva\_imagen.jpg").convert("RGB")

input\_tensor = transform(image).unsqueeze(0)

with torch.no\_grad():

output = model(input\_tensor)

print("Predicción:", output)

**9.3. Optimización para Inferencia en Tiempo Real**

Para aplicaciones en tiempo real, es importante optimizar el modelo y el entorno.

* **Cuantización del Modelo**: Reduce el tamaño del modelo y mejora la velocidad.

model = torch.quantization.quantize\_dynamic(

model, {torch.nn.Linear}, dtype=torch.qint8

)

* **Uso de GPU**: Mueve el modelo y los datos a la GPU si está disponible.

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

model.to(device)

input\_tensor = input\_tensor.to(device)

* **Batch Processing**: Procesa imágenes en lotes para aprovechar mejor los recursos computacionales.

batch = torch.stack([input\_tensor1, input\_tensor2]) # Lote de entradas

outputs = model(batch)

* **Implementación en Servidores con FastAPI o Flask**: Sirve el modelo como una API para recibir y procesar imágenes.

from fastapi import FastAPI, File, UploadFile

from io import BytesIO

app = FastAPI()

@app.post("/predict/")

async def predict(file: UploadFile = File(...)):

image = Image.open(BytesIO(await file.read())).convert("RGB")

input\_tensor = transform(image).unsqueeze(0).to(device)

with torch.no\_grad():

output = model(input\_tensor)

return {"prediction": output.tolist()}

**10. Apéndices**

**10.1. Código Completo del Entrenamiento**

Incluye una recopilación del código utilizado para cargar datos, definir el modelo, entrenarlo y evaluar su desempeño.

|  |
| --- |
| from google.colab import drive  drive.mount('/content/drive') |
| !git clone https://github.com/ultralytics/yolov5  # Descargar YOLOv5  %cd yolov5  !pip install -r requirements.txt  # Instalar las dependencias |
| import os  import json  import glob  import numpy as np  import matplotlib.pyplot as plt  from PIL import Image, ImageDraw  import cv2  import torch  from torchvision import transforms  from torch.utils.data import Dataset, DataLoader  # Definir rutas en Google Drive (asegúrate de que las carpetas contengan imágenes)  train\_dir = '/content/drive/MyDrive/Strawberry Disease Detection Dataset-Usman afzaal/train/images'  test\_dir = '/content/drive/MyDrive/Strawberry Disease Detection Dataset-Usman afzaal/test/images'  val\_dir = '/content/drive/MyDrive/Strawberry Disease Detection Dataset-Usman afzaal/val/images'  severity\_dir = '/content/drive/MyDrive/Strawberry Disease Detection Dataset-Usman afzaal/Test Disease Severity Level'  # Crear directorios de salida si no existen  output\_train = '/content/drive/MyDrive/YOLOv5\_Annotations/train'  output\_test = '/content/drive/MyDrive/YOLOv5\_Annotations/test'  output\_val = '/content/drive/MyDrive/YOLOv5\_Annotations/val'  os.makedirs(output\_train, exist\_ok=True)  os.makedirs(output\_test, exist\_ok=True)  os.makedirs(output\_val, exist\_ok=True) |
| Funciones para lectura y generación de máscaras  def read\_polygon\_coordinates\_and\_label\_from\_json(json\_path):      with open(json\_path) as f:          data = json.load(f)      polygons = data["shapes"]      coordinates = []      labels = []      for polygon in polygons:          points = polygon["points"]          label = polygon["label"]          coordinates.append(points)          labels.append(label)      return coordinates, labels  def generate\_masks(image\_size, coordinates):      combined\_mask = np.zeros(image\_size, dtype=np.uint8)      for points in coordinates:          points = [(int(x), int(y)) for (x, y) in points]          if not all(isinstance(point, tuple) and len(point) == 2 for point in points):              print("Coordenadas no válidas:", points)              continue          mask = Image.new("L", image\_size, 0)          ImageDraw.Draw(mask).polygon(points, outline=1, fill=1)          combined\_mask = np.maximum(combined\_mask, np.array(mask))      return combined\_mask  def check\_image\_mask\_paths(image\_paths, mask\_paths):      print(f"Cantidad de imágenes: {len(image\_paths)}")      print(f"Cantidad de máscaras: {len(mask\_paths)}")      for img\_path in image\_paths:          print(f"Imagen: {img\_path}")          mask\_path = img\_path.replace('images', 'masks').replace('.jpg', '.png')          print(f"Máscara esperada: {mask\_path}")          if not os.path.exists(mask\_path):              print(f"Máscara no encontrada: {mask\_path}")      assert len(image\_paths) == len(mask\_paths), "La cantidad de imágenes y máscaras no coincide."  def get\_image\_paths\_masks\_and\_labels(directory):      image\_paths = []      masks = []      labels = []      for image\_path in glob.glob(os.path.join(directory, "\*.jpg")):          json\_path = image\_path.replace(".jpg", ".json")          if os.path.exists(json\_path):              coordinates, label = read\_polygon\_coordinates\_and\_label\_from\_json(json\_path)              image = Image.open(image\_path)              image\_size = image.size              image\_paths.append(image\_path)              masks.append(generate\_masks(image\_size, coordinates))              labels.append(label)      return image\_paths, masks, labels  def get\_image\_paths\_masks\_labels\_and\_severity(directory):      image\_paths = []      masks = []      labels = []      severities = []      for severity\_level in ["Level 1", "Level 2"]:          level\_dir = os.path.join(directory, severity\_level)          for image\_path in glob.glob(os.path.join(level\_dir, "\*.jpg")):              json\_path = image\_path.replace(".jpg", ".json")              if os.path.exists(json\_path):                  coordinates, label = read\_polygon\_coordinates\_and\_label\_from\_json(json\_path)                  image = Image.open(image\_path)                  image\_size = image.size                  image\_paths.append(image\_path)                  masks.append(generate\_masks(image\_size, coordinates))                  labels.append(label)                  severities.append(severity\_level)      return image\_paths, masks, labels, severities |
| Dataset de frutillas con transformaciones  import os  import json  from PIL import Image, ImageDraw  import numpy as np  import torch  from torch.utils.data import Dataset, DataLoader  from torchvision import transforms  data\_transforms = transforms.Compose([      transforms.Resize((640, 640)),      transforms.ToTensor(),      transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  ])  mask\_transforms = transforms.Compose([      transforms.Resize((640, 640)),      transforms.ToTensor()  ])  def draw\_mask\_on\_image(image, mask\_json):      draw = ImageDraw.Draw(image)      for region in mask\_json['shapes']:          points = region['points']          polygon\_coords = [(int(x), int(y)) for x, y in points]          draw.polygon(polygon\_coords, outline=None, fill=255)      return image  class StrawberryDataset(Dataset):      def \_\_init\_\_(self, image\_dir, mask\_dir, image\_transforms=None, mask\_transforms=None):          self.image\_dir = image\_dir          self.mask\_dir = mask\_dir          self.image\_transforms = image\_transforms          self.mask\_transforms = mask\_transforms          # Crear un diccionario con los nombres de los archivos de imagen sin extensión          image\_files = {f.split(".")[0]: f for f in os.listdir(image\_dir) if f.endswith('.jpg')}          mask\_files = {f.split(".")[0]: f for f in os.listdir(mask\_dir) if f.endswith('.json')}          # Solo quedarnos con los archivos que tienen tanto imagen como máscara          common\_files = image\_files.keys() & mask\_files.keys()          # Depuración: Imprimir información sobre los archivos          print(f"Total de archivos de imagen: {len(image\_files)}")          print(f"Total de archivos de máscara: {len(mask\_files)}")          print(f"Archivos comunes: {len(common\_files)}")          # Crear listas de paths para imágenes y máscaras correspondientes          self.image\_paths = [os.path.join(image\_dir, image\_files[f]) for f in common\_files]          self.mask\_paths = [os.path.join(mask\_dir, mask\_files[f]) for f in common\_files]      def \_\_len\_\_(self):          return len(self.image\_paths)      def \_\_getitem\_\_(self, idx):          # Cargar la imagen y la máscara          image\_path = self.image\_paths[idx]          mask\_path = self.mask\_paths[idx]          # Cargar la imagen y aplicar transformaciones          image = Image.open(image\_path).convert("RGB")          if self.image\_transforms:              image = self.image\_transforms(image)          # Cargar la máscara desde el archivo JSON          with open(mask\_path, 'r') as f:              mask\_data = json.load(f)          # Crear una imagen en blanco para la máscara (del mismo tamaño que la imagen)          mask = Image.new('L', (image.width, image.height))          draw = ImageDraw.Draw(mask)          # Dibujar la máscara basada en los puntos del archivo JSON          for shape in mask\_data['shapes']:              points = shape['points']              draw.polygon(points, outline=1, fill=1)          if self.mask\_transforms:              mask = self.mask\_transforms(mask)          return image, mask  # Nueva función: preparar\_targets para YOLOv5  def preparar\_targets(masks, image\_size):      targets = []      for mask in masks:          mask = mask.numpy()  # Convertir el tensor de PyTorch a numpy          object\_indices = np.where(mask == 1)          if len(object\_indices[0]) > 0:              x\_min, x\_max = np.min(object\_indices[1]), np.max(object\_indices[1])              y\_min, y\_max = np.min(object\_indices[0]), np.max(object\_indices[0])              # Normalizar las coordenadas              bbox = [x\_min / image\_size[0], y\_min / image\_size[1], x\_max / image\_size[0], y\_max / image\_size[1]]              targets.append(bbox)          else:              targets.append([0, 0, 0, 0])  # Si no hay objeto, añades un bbox vacío      return torch.tensor(targets, dtype=torch.float32) |
| import os  # Ruta de la carpeta de las máscaras  mask\_dir = '/content/drive/MyDrive/Strawberry Disease Detection Dataset-Usman afzaal/train/masks'  # Listar los archivos en la carpeta de máscaras  mask\_files = [f for f in os.listdir(mask\_dir) if f.endswith('.json')]  # Depuración: Mostrar los primeros 10 archivos en la carpeta de máscaras  print(f"Total de archivos de máscara: {len(mask\_files)}")  print("Algunos archivos de máscara:", mask\_files[:10])  # Verifica si hay archivos con extensión ".JSON" en mayúsculas  mask\_files\_upper = [f for f in os.listdir(mask\_dir) if f.endswith('.JSON')]  print(f"Total de archivos de máscara con .JSON en mayúsculas: {len(mask\_files\_upper)}") |
| Modelo de entrenamiento  import matplotlib.pyplot as plt  import numpy as np  import torch  import torch.nn as nn  import torch.optim as optim  from torch.utils.data import DataLoader  import joblib  import os  from models.yolo import Model as YOLOv5  total\_epochs = 60  batch\_size = 10  model = YOLOv5(cfg='/content/yolov5/models/yolov5s.yaml')  criterion = nn.BCEWithLogitsLoss()  optimizer = optim.Adam(model.parameters(), lr=1e-4)  def train\_model(model, train\_loader, val\_loader, total\_epochs, device):      model.to(device)      for epoch in range(total\_epochs):          model.train()          running\_loss = 0.0          for i, (images, masks) in enumerate(train\_loader):              images = images.to(device)              masks = masks.to(device)              optimizer.zero\_grad()              outputs = model(images)              targets = preparar\_targets(masks, images.shape[2:])              targets = targets.to(device)              total\_loss = criterion(outputs, targets.float())              total\_loss.backward()              optimizer.step()              running\_loss += total\_loss.item()          print(f"Época {epoch+1}/{total\_epochs}, Pérdida: {running\_loss / len(train\_loader)}")          val\_loss = 0.0          model.eval()          with torch.no\_grad():              for images, masks in val\_loader:                  images = images.to(device)                  masks = masks.to(device)                  outputs = model(images)                  targets = preparar\_targets(masks, images.shape[2:])                  targets = targets.to(device)                  loss = criterion(outputs, targets.float())                  val\_loss += loss.item()          print(f"Pérdida de validación: {val\_loss / len(val\_loader)}") |
| import os  import joblib  # Definir la ruta donde guardar el modelo  save\_dir = "/content/drive/MyDrive/YOLOv5\_Models"  # Verificar si el directorio existe, si no, crearlo  if not os.path.exists(save\_dir):      os.makedirs(save\_dir)  # Guardar el modelo  model\_path = os.path.join(save\_dir, "strawberry\_model\_v2.pth")  joblib.dump(model.state\_dict(), model\_path)  print(f"Modelo guardado en: {model\_path}") |

**10.2. Referencias y Recursos**

* Documentación oficial de PyTorch: https://pytorch.org/docs/
* Repositorio YOLOv5: <https://github.com/ultralytics/yolov5>
* Tutoriales sobre Computer Vision: <https://www.kaggle.com/>
* S. Chen, Y. Liao, F. Lin and B. Huang, "An Improved Lightweight YOLOv5 Algorithm for Detecting Strawberry Diseases," in IEEE Access, vol. 11, pp. 54080-54092, 2023, doi: 10.1109/ACCESS.2023.3282309.

keywords: {Diseases;Feature extraction;Computational modeling;Plant diseases;Computer vision;Real-time systems;Mathematical models;Image classification;Computer vision;image classification;lightweight network;YOLOv5},

* Tutorial de aprendizaje automático con Python, [Bernd Klein](https://www.google.com.ar/search?hl=es&tbo=p&tbm=bks&q=inauthor:%22Bernd+Klein%22&source=gbs_metadata_r&cad=1), Alanna Maldonado, 2023, <https://python-course.eu/books/bernd_klein_python_and_machine_learning_a4.pdf>, 454 páginas
* X. Nie, L. Wang, H. Ding and M. Xu, "Strawberry Verticillium Wilt Detection Network Based on Multi-Task Learning and Attention," in IEEE Access, vol. 7, pp. 170003-170011, 2019, doi: 10.1109/ACCESS.2019.2954845.
* keywords: {Diseases;Feature extraction;Object detection;Task analysis;Proposals;Computer vision;Multitasking;Computer vision;image classification;multitasking;object detection},
* J. Chen, W. Chen, A. Zeb, S. Yang and D. Zhang, "Lightweight Inception Networks for the Recognition and Detection of Rice Plant Diseases," in IEEE Sensors Journal, vol. 22, no. 14, pp. 14628-14638, 15 July15, 2022, doi: 10.1109/JSEN.2022.3182304.
* keywords: {Diseases;Feature extraction;Crops;Support vector machines;Image recognition;Deep learning;Solid modeling;Crop disease detection;deep learning;enhanced Inception module;lightweight network;transfer learning},
* T. -Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan and S. Belongie, "Feature Pyramid Networks for Object Detection," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 2017, pp. 936-944, doi: 10.1109/CVPR.2017.106. keywords: {Feature extraction;Detectors;Semantics;Computer architecture;Proposals;Object detection;Robustness},
* S. Liu, L. Qi, H. Qin, J. Shi and J. Jia, "Path Aggregation Network for Instance Segmentation," 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 2018, pp. 8759-8768, doi: 10.1109/CVPR.2018.00913. keywords: {Proposals;Feature extraction;Task analysis;Image segmentation;Object detection;Training;Semantics},
* K. Han, Y. Wang, Q. Tian, J. Guo, C. Xu and C. Xu, "GhostNet: More Features From Cheap Operations," 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Seattle, WA, USA, 2020, pp. 1577-1586, doi: 10.1109/CVPR42600.2020.00165. keywords: {Convolution;Computational modeling;Computer architecture;Redundancy;Biological neural networks;Convolutional neural networks},
* M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and L. -C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 2018, pp. 4510-4520, doi: 10.1109/CVPR.2018.00474. keywords: {Manifolds;Neural networks;Computer architecture;Standards;Computational modeling;Task analysis},
* A. Howard et al., "Searching for MobileNetV3," 2019 IEEE/CVF International Conference on Computer Vision (ICCV), Seoul, Korea (South), 2019, pp. 1314-1324, doi: 10.1109/ICCV.2019.00140. keywords: {Computer architecture;Proposals;Computational modeling;Image segmentation;Neural networks;Next generation networking;Mobile handsets},
* X. Ding, X. Zhang, J. Han and G. Ding, "Scaling Up Your Kernels to 31×31: Revisiting Large Kernel Design in CNNs," 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), New Orleans, LA, USA, 2022, pp. 11953-11965, doi: 10.1109/CVPR52688.2022.01166. keywords: {Convolutional codes;Shape;Scalability;Transformers;Data models;Pattern recognition;Convolutional neural networks;Deep learning architectures and techniques},

**10.3. Glosario de Términos**

**Anotaciones**

Son las etiquetas o metadatos que se asignan a las imágenes dentro de un dataset para describir las características de los objetos presentes en ellas. Estas anotaciones pueden ser de diferentes tipos, como las cajas delimitadoras (bounding boxes) para detección de objetos, o máscaras para segmentación.

**API (Interfaz de Programación de Aplicaciones)**

Conjunto de reglas y herramientas que permiten a diferentes aplicaciones interactuar entre sí. En el contexto de la implementación en producción, puede hacer referencia a cómo el modelo entrenado se expone para ser utilizado desde otras aplicaciones o servicios.

**Batch Processing (Procesamiento en Lotes)**

Es una técnica en la que se procesan varias imágenes de manera simultánea, en lugar de hacerlo una por una, para optimizar el uso de la memoria y mejorar la eficiencia computacional, especialmente cuando se usan GPUs.

**Bounding Box (Caja Delimitadora)**

Rectángulo utilizado para identificar y localizar un objeto en una imagen. Se define por las coordenadas de sus esquinas superiores izquierda e inferior derecha. Es comúnmente usada en la detección de objetos.

**Cuantización**

Proceso de reducir la precisión de los cálculos matemáticos en el modelo para hacerlo más eficiente en términos de memoria y tiempo de ejecución. Se busca equilibrar entre el rendimiento y la precisión del modelo, logrando que sea más rápido sin perder demasiada exactitud.

**Evaluación de Modelo**

Es el proceso de medir el rendimiento del modelo después de haber sido entrenado. Incluye el uso de métricas como la precisión (accuracy), la intersección sobre la unión (IoU), y otras que indican qué tan bien el modelo realiza su tarea en el conjunto de datos de validación.

**Función de Pérdida (Loss Function)**

Es la medida que indica qué tan lejos está la predicción del modelo de la verdad. La función de pérdida se utiliza durante el entrenamiento para ajustar los parámetros del modelo con el objetivo de minimizarla.

**GPU (Unidad de Procesamiento Gráfico)**

Dispositivo especializado en realizar cálculos de alto rendimiento, especialmente útil para entrenar modelos de aprendizaje automático debido a su capacidad para manejar operaciones de matrices grandes de manera más eficiente que las CPU convencionales.

**Inferencia**

Proceso de utilizar un modelo entrenado para hacer predicciones sobre datos nuevos. A diferencia del entrenamiento, la inferencia solo usa el modelo previamente aprendido para obtener resultados, como la detección de objetos en una nueva imagen.

**IoU (Intersection over Union)**

Métrica utilizada para evaluar la precisión de las predicciones del modelo en tareas de detección de objetos. Mide la superposición entre la caja delimitadora predicha por el modelo y la caja delimitadora real (anotada).

**Máscara**

En el contexto de segmentación, es una representación binaria en la que se marca (generalmente con valores de 1) la zona de interés o la región correspondiente a un objeto, y el resto de la imagen está marcado con valores de 0. Es usada para identificar la forma exacta de un objeto en la imagen.

**Normalización**

Proceso de ajustar las imágenes para que sus valores de píxeles estén en un rango estándar (normalmente [0, 1] o [-1, 1]) para facilitar el aprendizaje del modelo y mejorar la convergencia durante el entrenamiento.

**ONNX (Open Neural Network Exchange)**

Formato abierto utilizado para representar modelos de aprendizaje automático, permitiendo que un modelo entrenado en un framework como PyTorch se use en otros frameworks como TensorFlow, o en dispositivos como teléfonos móviles y servidores de producción.

**Optimizador**

Algoritmo utilizado para actualizar los parámetros del modelo durante el entrenamiento. Los optimizadores como Adam, SGD, y RMSprop ayudan a minimizar la función de pérdida ajustando los pesos de la red neuronal.

**Overfitting (Sobreajuste)**

Es el fenómeno en el que un modelo aprende demasiado bien los detalles del conjunto de entrenamiento, incluyendo el ruido, lo que le impide generalizar correctamente a nuevos datos. Se soluciona mediante técnicas como regularización o el uso de datos adicionales.

**PyTorch Dataset**

Una estructura en PyTorch para manejar conjuntos de datos personalizados. Permite definir cómo cargar, preprocesar y etiquetar datos antes de ser alimentados al modelo para el entrenamiento o la inferencia.

**Redimensionamiento de Imágenes**

Proceso de cambiar el tamaño de las imágenes para que coincidan con las dimensiones esperadas por el modelo. En el caso de YOLOv5, las imágenes se redimensionan típicamente a 640x640 píxeles.

**Red Neuronal Convolucional (CNN)**

Arquitectura de red neuronal utilizada en el análisis de imágenes. Está diseñada para identificar patrones espaciales en las imágenes, como bordes, formas y texturas, siendo fundamental en tareas de clasificación y detección de objetos.

**Transferencia de Aprendizaje**

Técnica que implica tomar un modelo previamente entrenado en un conjunto de datos grande y usarlo como punto de partida para un nuevo modelo en un conjunto de datos más pequeño, lo que acelera el entrenamiento y mejora la precisión.

**TorchScript**

Es una representación intermedia de los modelos entrenados en PyTorch que permite su exportación a otros lenguajes o plataformas y facilita la inferencia en producción.

**Transformaciones de Imágenes**

Conjunto de técnicas para modificar las imágenes antes de ser alimentadas al modelo, como la rotación, el volteo, el recorte, y otras que ayudan a aumentar la diversidad del conjunto de entrenamiento y mejorar la robustez del modelo.

**YOLO (You Only Look Once)**

Es una técnica de detección de objetos en tiempo real. YOLOv5 es una implementación de esta arquitectura que divide la imagen en una cuadrícula y predice las ubicaciones y las clases de los objetos en cada celda. Es eficiente y rápido, adecuado para aplicaciones de producción.

**Miembros del Proyecto:**

**Marcelo Mamani, Armando Flores, Carla Salazar**

**RCONOCIMIENTO DE LETRAS MEDIANTE GESTOS DE MANO  
Fecha de presentación: 25 noviembre de 2024  
Curso: 3° AÑO  
Profesor: Alfredo Sardinas**

**Índice para la Aplicación de Detección de Enfermedades en Frutillas**

1. **Introducción**
   * Descripción general de la aplicación.
   * Objetivo: detectar enfermedades en cultivos de frutillas utilizando modelos de IA y proporcionar recomendaciones de tratamiento.
2. **Requisitos del Sistema**
   * Hardware necesario.
   * Software y librerías requeridas (Kivy, OpenCV, Torch, etc.).
3. **Estructura del Proyecto**
   * Explicación de la estructura de carpetas y archivos.
   * Archivos principales:
     + app.py: contiene la lógica de la aplicación.
     + strawberry\_model\_v2.pt: modelo entrenado de YOLOv5.
4. **Descripción de las Pantallas de la Aplicación**
   * **HomeScreen (Pantalla Principal)**
     + Funcionalidad de la pantalla.
     + Botones de interacción: "Capturar Foto", "Cargar Imagen".
   * **CameraScreen (Pantalla de Captura de Imagen)**
     + Captura de foto utilizando la cámara del dispositivo.
     + Guardar o analizar la imagen capturada.
   * **ProcessingScreen (Pantalla de Procesamiento)**
     + Proceso de análisis y detección de enfermedades utilizando el modelo YOLOv5.
   * **ResultScreen (Pantalla de Resultados)**
     + Mostrar la imagen analizada junto con el resultado de la detección (enfermedad detectada o estado sano).
5. **Lógica de Procesamiento de Imágenes**
   * **Captura de Foto**
     + Uso de la cámara para tomar una imagen y guardarla temporalmente.
     + Selección de acciones (guardar, analizar o eliminar la imagen).
   * **Cargar Imagen desde el Sistema de Archivos**
     + Uso del FileChooser para cargar una imagen desde la galería o el sistema de archivos del dispositivo.
     + Filtros para seleccionar solo imágenes en formatos .png, .jpg, .jpeg.
6. **Detección de Enfermedades**
   * Explicación del uso del modelo entrenado strawberry\_model\_v2.pt (YOLOv5).
   * Proceso de análisis de la imagen:
     + Carga de la imagen con OpenCV.
     + Uso de torch para realizar inferencias con el modelo.
     + Manejo de los resultados (detección de enfermedades o frutilla sana).
7. **Recomendaciones de Tratamiento**
   * Dependiendo del resultado de la detección, mostrar recomendaciones específicas para cada enfermedad detectada.
8. **Manejo de Errores**
   * Control de errores en la carga de imágenes, procesamiento y uso del modelo.
   * Ejemplos de manejo de excepciones y despliegue de mensajes emergentes.
9. **Pruebas y Validación**
   * Métodos para probar la aplicación con diferentes imágenes de frutillas.
   * Validación de la precisión del modelo utilizando imágenes de referencia.
10. **Conclusión**
    * Resumen del flujo de la aplicación y su funcionamiento.
    * Posibles mejoras y expansiones del proyecto.

**1. Introducción**

**1.1. Descripción general de la aplicación**

La aplicación propuesta es un sistema avanzado de detección de enfermedades en cultivos de frutillas, diseñado para asistir a agricultores en la identificación temprana de problemas en sus cultivos. Utiliza inteligencia artificial y visión por computadora para analizar imágenes de frutillas, identificar enfermedades comunes y ofrecer recomendaciones de tratamiento basadas en los resultados. La aplicación también tiene la capacidad de aprovechar datos climáticos para sugerir las mejores fechas para aplicar fertilizantes o tratamientos, mejorando así la eficiencia y el rendimiento de los cultivos.

**1.2. Objetivo**

El objetivo principal de esta aplicación es proporcionar a los agricultores una herramienta fácil de usar que les permita detectar enfermedades en cultivos de frutillas de manera rápida y precisa. Utilizando modelos de inteligencia artificial, la aplicación puede identificar visualmente enfermedades comunes y sugerir tratamientos adecuados, ayudando a los agricultores a tomar decisiones informadas que optimicen la salud de los cultivos y prevengan pérdidas significativas.

2. **Requisitos del Sistema**

**2.1. Hardware necesario**

Para ejecutar la aplicación de detección de enfermedades en cultivos de frutillas de manera eficiente, se requiere el siguiente hardware mínimo:

* **Procesador**: Intel Core i5 o superior.
* **Memoria RAM**: 8 GB (16 GB recomendados para un rendimiento óptimo).
* **Almacenamiento**: Al menos 10 GB de espacio libre en disco para el almacenamiento de imágenes, modelos de IA y dependencias.
* **GPU**: NVIDIA con soporte para CUDA (recomendado para acelerar el procesamiento de imágenes con redes neuronales).
* **Cámara**: Cámara integrada o externa para capturar imágenes de los cultivos de frutillas.
* **Conexión a Internet**: Opcional, para actualizar modelos o recibir datos climáticos en tiempo real.

**2.2. Software y librerías requeridas**

La aplicación requiere el siguiente software y las librerías para su desarrollo y ejecución:

* **Sistema Operativo**: Windows, macOS o Linux.
* **Python 3.10**: Lenguaje de programación principal.
* **Librerías necesarias**:
  + **Kivy**: Para el desarrollo de la interfaz gráfica de usuario (GUI).
  + **OpenCV**: Para la captura y procesamiento de imágenes.
  + **PyTorch**: Para el uso del modelo YOLOv5 entrenado.
  + **YOLOv5**: Modelo de detección de objetos, preentrenado para la identificación de enfermedades en cultivos de frutillas.
  + **PIL (Pillow)**: Para el manejo de imágenes.
  + **Matplotlib**: Para la visualización de datos y análisis de resultados.
  + **Numpy**: Para la manipulación eficiente de matrices y arrays numéricos.
  + **Requests**: Para las peticiones HTTP (opcional, para obtener datos climáticos).

**Estructura del Proyecto**

**3.1. Explicación de la estructura de carpetas y archivos**

La estructura del proyecto sigue una organización modular para facilitar el desarrollo, la navegación y el mantenimiento. A continuación se describe la estructura de carpetas y archivos:

Bash:

/strawberry-disease-detection-app

│

├── /data # Contiene las imágenes y los archivos JSON de las máscaras

│ ├── /train # Subcarpeta con imágenes de entrenamiento

│ └── /val # Subcarpeta con imágenes de validación

│

├── /models # Modelos entrenados

│ └── strawberry\_model\_v2.pt # Modelo YOLOv5 entrenado

│

├── /utils # Herramientas adicionales (preprocesamiento, evaluación)

│ └── preprocessing.py # Código para el preprocesamiento de imágenes y máscaras

│

├── /gui # Interfaz gráfica de usuario

│ └── kivy\_interface.kv # Archivo con el diseño visual en Kivy

│

├── app.py # Archivo principal con la lógica de la aplicación

├── requirements.txt # Lista de librerías necesarias para el entorno virtual

└── README.md # Documentación del proyecto

**3.2. Archivos principales**

**3.2.1. app.py: contiene la lógica de la aplicación**

Este archivo es el corazón de la aplicación. Aquí se implementa la lógica principal de la interfaz gráfica, la interacción con la cámara, el procesamiento de imágenes y la integración con el modelo YOLOv5. También es responsable de las recomendaciones de tratamiento basadas en la detección de enfermedades y de la integración opcional con datos climáticos para sugerencias.

Puntos clave:

* Manejo de la captura de imágenes desde la cámara.
* Preprocesamiento de las imágenes capturadas.
* Uso del modelo YOLOv5 para detectar enfermedades.
* Presentación de los resultados al usuario en la interfaz gráfica.
* Llamadas opcionales a un servicio de predicción meteorológica para optimizar tratamientos.

**3.2.2. strawberry\_model\_v2.pt: modelo entrenado de YOLOv5**

Este archivo contiene el modelo YOLOv5 entrenado, específicamente diseñado para detectar enfermedades en frutillas. El modelo fue entrenado con un conjunto de datos especializado, que incluye diversas enfermedades como la mancha foliar, el mildiu y otras afecciones comunes en los cultivos de frutillas.

4. **Descripción de las Pantallas de la Aplicación**

**4.1. HomeScreen (Pantalla Principal)**

**4.1.1. Funcionalidad de la pantalla**

La **HomeScreen** es la pantalla principal de la aplicación. Su propósito es servir como punto de partida para las interacciones del usuario. Desde esta pantalla, el usuario puede acceder a las funciones clave de la aplicación: capturar una foto en tiempo real o cargar una imagen desde el dispositivo para su análisis. También muestra un título o mensaje de bienvenida que describe brevemente la funcionalidad de la aplicación.

**4.1.2. Botones de interacción:**

* **Capturar Foto**: Este botón abre la cámara del dispositivo y dirige al usuario a la pantalla **CameraScreen**, donde puede capturar una foto de sus cultivos de frutillas.
* **Cargar Imagen**: Permite al usuario cargar una imagen almacenada localmente en su dispositivo. Una vez cargada, la aplicación procesa la imagen y la envía a la pantalla de procesamiento para la detección de enfermedades.

**4.2. CameraScreen (Pantalla de Captura de Imagen)**

**4.2.1. Captura de foto utilizando la cámara del dispositivo**

La **CameraScreen** permite al usuario tomar una foto de sus frutillas directamente utilizando la cámara del dispositivo. La interfaz de usuario incluye una vista en tiempo real de lo que captura la cámara, permitiendo al usuario ajustar la posición y el encuadre de la imagen antes de tomar la foto.

**4.2.2. Guardar o analizar la imagen capturada**

Una vez que el usuario captura la foto, se le presenta la opción de:

* **Guardar la imagen** en el almacenamiento del dispositivo.
* **Analizar la imagen**, lo que enviará la imagen directamente al modelo YOLOv5 para procesarla y detectar posibles enfermedades en el cultivo. Esto redirige al usuario a la pantalla de procesamiento **ProcessingScreen** para iniciar el análisis.

**4.3. ProcessingScreen (Pantalla de Procesamiento)**

**4.3.1. Proceso de análisis y detección de enfermedades utilizando el modelo YOLOv5**

En la **ProcessingScreen**, se muestra al usuario una animación o un indicador de carga mientras la imagen capturada o cargada está siendo procesada por el modelo **YOLOv5**. Este modelo analiza la imagen en busca de síntomas visibles de enfermedades en las frutillas. Dependiendo de la complejidad de la imagen y la cantidad de objetos detectados, el procesamiento puede tardar unos segundos.

El modelo realiza las siguientes tareas:

* Localización de áreas afectadas en la imagen.
* Clasificación de la enfermedad detectada o indicación de que el cultivo está sano.
* Generación de un mapa de calor o cuadro delimitador sobre la imagen para marcar las áreas con enfermedades.

Una vez finalizado el procesamiento, la aplicación redirige automáticamente al usuario a la pantalla de resultados **ResultScreen**.

**4.4. ResultScreen (Pantalla de Resultados)**

**4.4.1. Mostrar la imagen analizada junto con el resultado de la detección (enfermedad detectada o estado sano)**

La **ResultScreen** muestra el resultado del análisis de la imagen. En esta pantalla, el usuario puede ver:

* La **imagen analizada** con marcadores visuales, como cuadros delimitadores o mapas de calor, indicando las áreas donde se han detectado enfermedades.
* El **resultado de la detección**, que puede ser el nombre de la enfermedad detectada (por ejemplo, "mancha foliar", "mildiu") o un mensaje indicando que el cultivo está sano.
* **Recomendaciones de tratamiento**, si se ha detectado una enfermedad. Estas recomendaciones incluyen sugerencias específicas sobre pesticidas, fertilizantes o procedimientos que el agricultor puede aplicar para tratar la enfermedad detectada.
* **Opciones adicionales**, como:
  + Guardar la imagen con los resultados.
  + Compartir el informe a través de correo electrónico u otros medios.
  + Volver a la pantalla principal **HomeScreen** para realizar un nuevo análisis.

**5. Lógica de Procesamiento de Imágenes**

**5.1. Captura de Foto**

**5.1.1. Uso de la cámara para tomar una imagen y guardarla temporalmente**

Cuando el usuario elige la opción de **Capturar Foto** en la pantalla principal, la aplicación abre la cámara del dispositivo. Utilizando la funcionalidad de captura de imágenes en Kivy, la foto tomada se guarda de manera temporal en un directorio predefinido del dispositivo, generalmente en una carpeta temporal o de caché. Este archivo temporal será utilizado para análisis posterior o se eliminará si el usuario decide no continuar con el proceso.

**5.1.2. Selección de acciones (guardar, analizar o eliminar la imagen)**

Una vez que la foto ha sido capturada, el usuario tiene tres opciones:

* **Guardar la imagen**: Se permite al usuario almacenar la foto en una ubicación permanente en el sistema de archivos del dispositivo.
* **Analizar la imagen**: La imagen capturada se envía al modelo YOLOv5 para su análisis. La aplicación redirige a la pantalla de procesamiento donde se muestra el progreso del análisis.
* **Eliminar la imagen**: El usuario puede optar por descartar la imagen y regresar a la pantalla principal para tomar otra foto o cargar una imagen existente desde el sistema de archivos.

**5.2. Cargar Imagen desde el Sistema de Archivos**

**5.2.1. Uso del FileChooser para cargar una imagen desde la galería o el sistema de archivos del dispositivo**

Si el usuario elige la opción de **Cargar Imagen**, la aplicación utiliza el componente **FileChooser** de Kivy para abrir una ventana de exploración de archivos. Esto permite al usuario buscar y seleccionar una imagen desde la galería o cualquier carpeta de su sistema de archivos. Una vez seleccionada, la imagen se prepara para ser analizada por el modelo YOLOv5.

**5.2.2. Filtros para seleccionar solo imágenes en formatos .png, .jpg, .jpeg**

El **FileChooser** está configurado con filtros específicos para garantizar que el usuario solo pueda seleccionar imágenes en los formatos más comunes: **.png**, **.jpg** y **.jpeg**. Esto asegura que el sistema procese solo archivos compatibles con el modelo de detección y evita errores de procesamiento debidos a formatos no soportados.

**6. Detección de Enfermedades**

**6.1. Explicación del uso del modelo entrenado strawberry\_model\_v2.pt (YOLOv5)**

El modelo entrenado **strawberry\_model\_v2.pt** es una versión personalizada de **YOLOv5** entrenada específicamente para detectar enfermedades comunes en los cultivos de frutillas. Este modelo ha sido entrenado utilizando imágenes de frutillas afectadas por diversas enfermedades, así como imágenes de frutillas sanas. El modelo es capaz de detectar y clasificar las enfermedades en función de características visibles, como manchas, cambios de color o formas anormales en las hojas o frutos.

**6.2. Proceso de análisis de la imagen**

**6.2.1. Carga de la imagen con OpenCV**

Una vez que el usuario selecciona o captura una imagen, la aplicación utiliza **OpenCV** para cargar la imagen en memoria. **OpenCV** es una biblioteca de procesamiento de imágenes que facilita la manipulación y lectura de archivos de imagen. En este paso, la imagen es convertida a un formato compatible que puede ser procesado por el modelo YOLOv5.

**6.2.2. Uso de torch para realizar inferencias con el modelo**

El modelo **strawberry\_model\_v2.pt** es cargado utilizando **torch**, una biblioteca de aprendizaje profundo que permite realizar inferencias con redes neuronales. La imagen cargada es preprocesada y luego se realiza la inferencia. Durante la inferencia, el modelo analiza la imagen en busca de patrones y características asociadas a las enfermedades de frutillas y devuelve una lista de predicciones, que pueden incluir:

* **Cuadro delimitador** de las áreas afectadas.
* **Clase de enfermedad** detectada o indicación de que la frutilla está sana.
* **Confianza** en la predicción (probabilidad).

**6.2.3. Manejo de los resultados (detección de enfermedades o frutilla sana)**

Una vez obtenidos los resultados del modelo, la aplicación los procesa de la siguiente manera:

* Si se detecta una enfermedad, se muestra un cuadro delimitador sobre la imagen, indicando la ubicación del área afectada. Además, se muestra el nombre de la enfermedad detectada y el nivel de confianza de la predicción.
* Si no se detecta ninguna enfermedad, la aplicación muestra un mensaje indicando que la frutilla está **sana**.
* A continuación, el usuario recibe recomendaciones sobre posibles tratamientos para las enfermedades detectadas, o sugerencias sobre buenas prácticas de mantenimiento si la frutilla está en buen estado.

**7. Recomendaciones de Tratamiento**

Dependiendo del resultado de la detección, la aplicación proporciona recomendaciones específicas para cada enfermedad detectada. Estas recomendaciones se basan en prácticas agrícolas comunes y sugerencias de tratamiento para las enfermedades más comunes en frutillas. Las recomendaciones incluyen:

* **Mancha foliar**: Aplicar fungicidas específicos recomendados para controlar la enfermedad. También se sugiere mejorar la circulación del aire alrededor de las plantas y eliminar las hojas afectadas.
* **Moho gris**: Utilizar un fungicida adecuado y mantener el cultivo bien ventilado. Se aconseja evitar el exceso de riego para prevenir la propagación.
* **Oídio**: Recomendación de un fungicida especializado y la eliminación de las partes afectadas de la planta.
* **Frutilla sana**: Recomendaciones de buenas prácticas de manejo agrícola, como la rotación de cultivos, el riego moderado y el control preventivo de plagas y enfermedades.

Cada recomendación se muestra junto al resultado de la detección en la pantalla de resultados, brindando al agricultor una guía clara sobre los pasos a seguir.

**8. Manejo de Errores**

**8.1. Control de errores en la carga de imágenes, procesamiento y uso del modelo**

El manejo de errores es clave para garantizar que la aplicación funcione sin problemas. Los posibles errores incluyen:

* **Carga de imágenes no válidas**: Si el usuario intenta cargar un archivo que no es una imagen o que tiene un formato no soportado (diferente de .png, .jpg o .jpeg), la aplicación mostrará un mensaje de advertencia indicando que el archivo no es válido.
* **Errores en el procesamiento del modelo**: Si ocurre un error durante la inferencia del modelo, como un problema de memoria o una configuración incorrecta del archivo de modelo, la aplicación mostrará un mensaje emergente para informar al usuario que hubo un problema con el análisis.
* **Problemas con la cámara**: Si el dispositivo tiene problemas para acceder a la cámara o capturar una imagen, se mostrará un mensaje emergente para que el usuario intente nuevamente o verifique los permisos del dispositivo.

**8.2. Ejemplos de manejo de excepciones y despliegue de mensajes emergentes**

Ejemplo de manejo de excepciones en Python utilizando **try-except** para prevenir fallos inesperados:

python

try:

image = cv2.imread(image\_path)

if image is None:

raise ValueError("No se pudo cargar la imagen.")

except ValueError as e:

popup = Popup(title='Error de Imagen', content=Label(text=str(e)), size\_hint=(0.8, 0.2))

popup.open()

En este ejemplo, si la imagen no se carga correctamente, la excepción **ValueError** activa un mensaje emergente para notificar al usuario. De manera similar, se pueden agregar bloques **try-except** para el uso del modelo YOLOv5 y la captura de fotos.

**9. Pruebas y Validación**

**Pruebas de la aplicación con diferentes imágenes de frutillas**

Para garantizar que la aplicación funcione de manera adecuada en diferentes escenarios, se debe probar con una variedad de imágenes que representen distintas condiciones, tales como:

* **Imágenes con diferentes niveles de calidad**: Probar con imágenes en alta y baja resolución para verificar si el modelo puede detectar las enfermedades de manera confiable.
* **Imágenes con condiciones de iluminación variables**: Verificar cómo la luz afecta la detección de enfermedades.
* **Frutillas en diferentes estados de salud**: Probar tanto con frutillas sanas como enfermas para validar la precisión del modelo en la detección y la emisión de recomendaciones.

**Validación de la precisión del modelo utilizando imágenes de referencia**

Para validar la precisión del modelo, se deben realizar las siguientes acciones:

* **Conjunto de pruebas**: Usar un conjunto de imágenes de referencia con etiquetas conocidas (frutillas sanas y frutillas con enfermedades) para comparar los resultados del modelo con las predicciones correctas.
* **Métricas de precisión**: Evaluar el modelo utilizando métricas como la **precisión**, **recall** y el **F1-score** para medir su rendimiento en la clasificación de enfermedades.
* **Ajuste del modelo**: Si es necesario, realizar ajustes en el modelo entrenado o utilizar técnicas de ajuste fino (fine-tuning) para mejorar la precisión en la detección de enfermedades específicas.

Las pruebas y la validación garantizan que el sistema sea confiable y útil para los agricultores en la detección temprana de enfermedades y la toma de decisiones informadas sobre el tratamiento de sus cultivos.

**10. Conclusión**

La aplicación de detección de enfermedades en cultivos de frutillas ha sido diseñada para proporcionar a los agricultores una herramienta accesible y eficiente para identificar enfermedades comunes y recibir recomendaciones de tratamiento en tiempo real. A lo largo de este manual, se ha detallado el flujo y funcionamiento de la aplicación, que incluye las siguientes etapas clave:

* **Captura y carga de imágenes**: La aplicación permite capturar fotos directamente desde la cámara del dispositivo o cargar imágenes existentes desde el sistema de archivos.
* **Detección de enfermedades**: Utilizando el modelo entrenado de YOLOv5, la aplicación analiza las imágenes para detectar enfermedades en frutillas o identificar si la fruta está en buen estado.
* **Recomendaciones de tratamiento**: Dependiendo de los resultados del análisis, se brindan recomendaciones específicas para tratar las enfermedades detectadas.
* **Manejo de errores**: La aplicación cuenta con mecanismos de manejo de errores que permiten al usuario recibir alertas claras cuando ocurre algún problema durante la captura, carga o procesamiento de imágenes.
* **Validación y pruebas**: Se ha implementado un proceso de validación para asegurar la precisión del modelo en diferentes condiciones, mejorando la fiabilidad de los resultados.

**Posibles mejoras y expansiones del proyecto**

Aunque la aplicación cumple con los objetivos iniciales de detectar enfermedades y proporcionar recomendaciones, existen varias mejoras y expansiones que podrían considerarse para incrementar su funcionalidad y alcance:

1. **Ampliación del conjunto de enfermedades**: El modelo actual está limitado a detectar solo algunas enfermedades comunes en frutillas. En futuras versiones, se podría entrenar el modelo para detectar una gama más amplia de enfermedades y otras condiciones relacionadas con el cultivo.
2. **Integración de datos climáticos**: Implementar un sistema de pronóstico climático que permita al agricultor planificar mejor el uso de fertilizantes y tratamientos según las condiciones climáticas locales.
3. **Aplicación multiplataforma**: Expandir la aplicación a otras plataformas, como iOS y versiones de escritorio, utilizando tecnologías compatibles con Kivy.
4. **Optimización del rendimiento**: Mejorar el tiempo de procesamiento de las imágenes para hacer que la experiencia del usuario sea más rápida y fluida, especialmente en dispositivos con menor capacidad de hardware.
5. **Almacenamiento de resultados históricos**: Incluir una funcionalidad para almacenar y visualizar un historial de las enfermedades detectadas y los tratamientos recomendados para ayudar al agricultor a hacer un seguimiento a lo largo del tiempo.
6. **Detección de plagas**: Ampliar la funcionalidad del modelo para incluir la detección de plagas comunes que también afectan el cultivo de frutillas, brindando un enfoque más integral en el cuidado de los cultivos.