UNIVERSIDAD POLITÉCNICA SALESIANA

ASIGNATURA: VISIÓN POR COMPUTADOR

DOCENTE: ING. VLADIMIR ROBLES

Clasificación de Objetos Usando Técnicas de Deep Learning

Realizado por:
Peralta Peralta Diego Felipe.

I. Introducción

La visión por computador moderna demanda una capacidad de procesamiento masiva y algoritmos especializados, especialmente en aplicaciones que requieren alta precisión como la agricultura inteligente y el análisis de video en tiempo real. El presente laboratorio tiene como objetivo principal evaluar el rendimiento de diferentes arquitecturas de hardware (CPU vs. GPU) y paradigmas de desarrollo, abarcando desde la preparación de datos hasta la implementación de pipelines optimizados.

Se desarrollaron las 3 partes solicitadas en el laboratorio:

Parte 1A: Entrenamiento de Segmentación de Instancias con Dataset Agrícola

Esta fase se centra en el entrenamiento de redes neuronales para tareas de visión en agricultura de precisión. Se utiliza el **Fruit Instance Segmentation Dataset** [1], un conjunto de datos robusto compuesto por **11,546 imágenes** divididas en conjuntos de entrenamiento (10,134), validación (947) y prueba (465). El objetivo es entrenar un modelo capaz de segmentar y clasificar seis variedades de frutas: *bitter melon, cucumber, fig, jujube, melon boyang* y *muskmelon*. El dataset incorpora técnicas avanzadas de aumento de datos para mejorar la generalización del modelo. Esta etapa establece las bases del aprendizaje supervisado y la evaluación de métricas en segmentación.

Parte 1B: Benchmarking de Deep Learning con YOLO y Super Resolución

En esta etapa se somete al hardware a pruebas de estrés computacional utilizando modelos de vanguardia en inferencia. Se implementa un sistema híbrido que combina **YOLO** para la detección de objetos y **Real-ESRGAN** para la super resolución de video en tiempo real. El objetivo es evidenciar el cuello de botella que representa la CPU (se usó un Intel Core i9-12900K) frente a la aceleración paralela masiva de una GPU dedicada (se usó una NVIDIA RTX 3070), analizando métricas críticas como FPS, latencia de inferencia y consumo de memoria VRAM.

Parte 1C: Pipeline de Procesamiento GPU Low-Level con C++

Finalmente, se desciende al nivel de optimización de memoria utilizando C++ y la librería `opencv_cuda`. Se construye un pipeline de filtrado manual que se ejecuta íntegramente en la VRAM para minimizar las transferencias a través del bus PCIe. El flujo implementado incluye suavizado gaussiano, ecualización de histograma, filtros morfológicos y detección de bordes con Canny. Se demuestra cómo la gestión manual de memoria en GPU puede superar las barreras de rendimiento convencionales, alcanzando tasas de procesamiento superiores a los 800 FPS.

II. Resolución Parte 1 A

Realizado con: Samantha Suquilanda.

En esta primera fase, el objetivo fue entrenar una red neuronal convolucional capaz de realizar segmentación de instancias para identificar y delimitar frutas en entornos agrícolas. A diferencia de la detección convencional que solo genera cajas delimitadoras, este enfoque genera máscaras de píxeles precisas para cada objeto.

II-A. Arquitectura del Software

Para lograr el entrenamiento y la validación, se desarrolló una suite de scripts en Python utilizando el framework `Ultralytics` (YOLOv11/v8). A continuación se describe la funcionalidad técnica de cada módulo implementado:

- **Gestión del Dataset (`preparar_dataset_full.py`):**
  Este script automatiza la preparación de los datos. Su función principal es validar la estructura

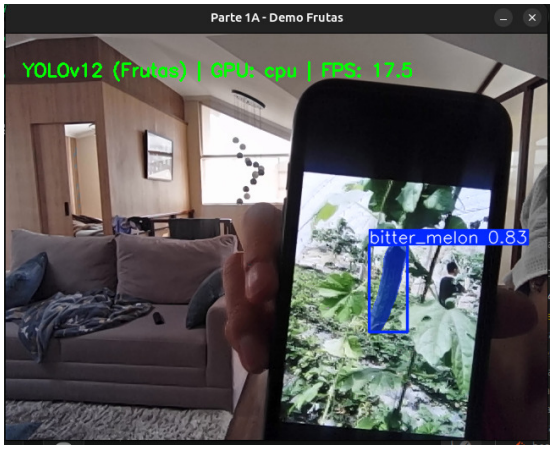
del *Fruit Instance Segmentation Dataset*, asegurando que las imágenes y las etiquetas (formato YOLO poligonales) estén correctamente distribuidas en los directorios de entrenamiento (`/train`) y validación (`/val`). Además, define el archivo de configuración `.yaml` necesario para que la red neuronal entienda las clases a detectar.

- **Entrenamiento del Modelo (`train.py` / `.ipynb`):**
  Se implementó un script de entrenamiento que utiliza la técnica de *Transfer Learning*. Se parte de un modelo base pre-entrenado (`yolo11n-seg.pt` o similar) y se re-entrena con el dataset de frutas.
  - **Hiperparámetros:** Se configuraron épocas de entrenamiento para asegurar la convergencia del loss, con un tamaño de imagen (imgsz) adecuado para capturar detalles pequeños de las frutas.
  - **Entorno:** Se utilizó un Jupyter Notebook (`Entrenamiento_Practica4_Parte1A.ipynb`) para ejecutar el entrenamiento en la nube (Google Colab), aprovechando la aceleración de GPUs Tesla T4 para reducir los tiempos de cómputo.
- **Inferencia en Tiempo Real (`inference_webcam.py`):**
  Una vez obtenido el modelo óptimo (`best.pt`), este script permite la validación en vivo. Utiliza la librería OpenCV para capturar el flujo de video de la webcam, pasa cada frame por la red neuronal segmentadora y superpone tanto la caja delimitadora como la máscara de color sobre las frutas detectadas, mostrando la confianza de la predicción en tiempo real.

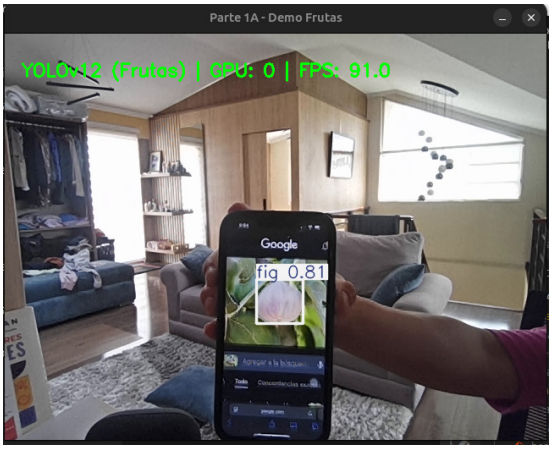
II-B. Resultados de Inferencia y Rendimiento

Para validar el funcionamiento del modelo entrenado, se realizaron pruebas de predicción sobre imágenes de control y se monitoreó el consumo de hardware durante el proceso.

Se presentan ejemplos de detección para las clases *Bitter Melon* y *Fig*, donde el modelo delimita correctamente la geometría del fruto y asigna la etiqueta con su respectiva confianza:



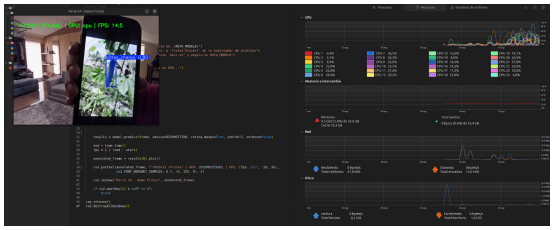
(a) Segmentación de instancia: *Bitter Melon*.



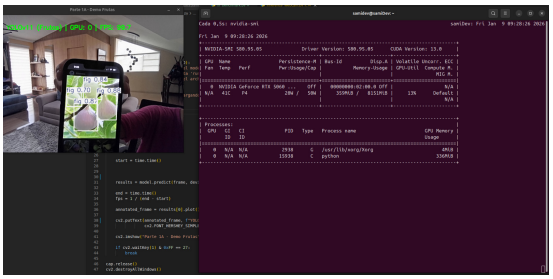
(b) Segmentación de instancia: *Fig* (Higo).

Figura 1: Resultados cualitativos de la segmentación de frutas.

Por otro lado, se evidencia el impacto computacional. Se contrastó el consumo de recursos al realizar estas predicciones, observando las diferencias de carga entre el procesamiento puramente en CPU y la aceleración mediante GPU:



(a) Monitor de recursos durante inferencia en CPU.



(b) Monitor de recursos durante inferencia en GPU.

Figura 2: Análisis de consumo de hardware durante la fase de predicción.

III. Resolución Parte 1 B

En esta sección se detalla el procedimiento experimental para evaluar el rendimiento de inferencia utilizando redes neuronales profundas. El objetivo fue procesar un archivo de video aplicando dos modelos secuenciales: detección de objetos con YOLOv11/12 [2] y super-resolución de imagen con Real-ESRGAN (escalado x2) [3].

III-A. Material Utilizado

Como material de entrada se utilizó un video viral de dominio público, correspondiente a un baile de Donald Trump, citado a continuación:

- **Fuente:** El País Digital [4].
- **Resolución original:** 608×1080 píxeles.
- **Duración:** 14 segundos.

Para el procesamiento se desarrolló el script `main.py`, el cual implementa un pipeline que captura cada frame, detecta objetos como personas y corbatas en el video, aplica anotaciones y posteriormente mejora la resolución de la imagen resultante mediante la red neuronal.

III-B. Descripción del Experimento

Se realizaron dos ejecuciones controladas del mismo script, variando únicamente el dispositivo de procesamiento (`--device cpu` y `--device cuda`). Se grabó la pantalla durante la ejecución para evidenciar las métricas en tiempo real.

III-B1. Prueba en GPU: Se utilizó una tarjeta gráfica **NVIDIA RTX 3070**.

- **Tiempo de ejecución:** El procesamiento del video completo (14 segundos) tomó aproximadamente **3 minutos y 30 segundos** de grabación. Enlace al video: https://youtu.be/hHQVw1QqHYY.
- **Rendimiento:** El sistema mantuvo una tasa de 2FPS, con un uso de GPU del 100 % y un consumo de VRAM dedicado de $\sim$2.2 GB, como se observa en la Figura 3.
- **Resultado:** El video de salida `salida-cuda.mp4` se generó exitosamente con una resolución final escalada de 760×1350 píxeles.

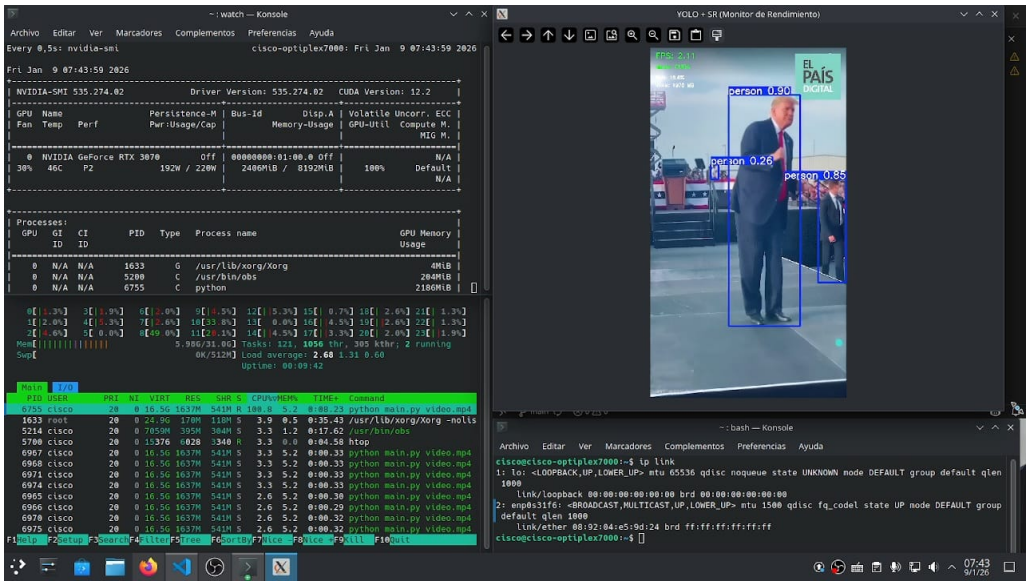


Figura 3: Ejecución en GPU. Se observa el comando `nvidia-smi` indicando carga máxima en la tarjeta gráfica y la detección fluida en la ventana de visualización.

III-B2. Prueba en CPU: Se utilizó el procesador central **Intel Core i9-12900K**.

- **Tiempo de ejecución:** Esta prueba evidenció un cuello de botella severo. La grabación se detuvo a los **21 minutos y 17 segundos**, momento en el cual apenas se habían logrado procesar **3 segundos** del video original. Enlace al video: https://youtu.be/_SPr1W4mVk0.
- **Rendimiento:** La tasa de cuadros cayó drásticamente a ∼0.5 - 0.7 FPS. La herramienta `htop` (Figura 4) muestra la saturación de los núcleos del procesador al 99.4 %, confirmando que la carga computacional de Real-ESRGAN es inmanejable para la CPU en tiempo real.
- **Resultado:** Se generó el archivo `salida-cpu.mp4` con la misma resolución de salida (760 × 1350), pero cubriendo solo una fracción del contenido original debido a la extrema lentitud.

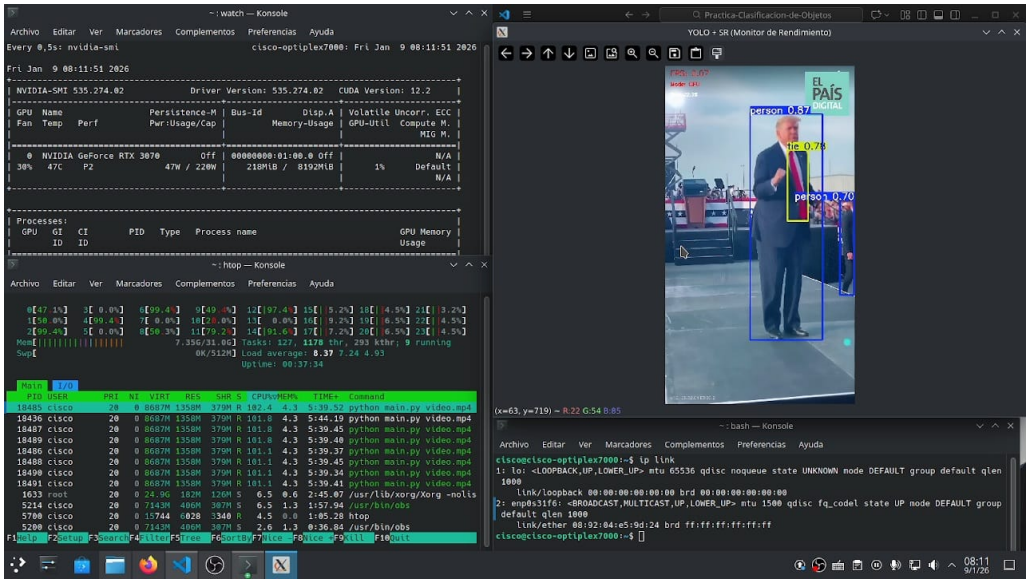


Figura 4: Ejecución en CPU. A pesar de utilizar un procesador i9, el rendimiento es insuficiente para Super Resolución en tiempo real, saturando el núcleo de procesamiento.

III-C. Verificación de Resultados

Finalmente, se verificó la integridad de los videos generados mediante un script de inspección de metadatos (`get_resolutions.py`), que se encuentra en el repositorio. Los resultados confirman que el pipeline de Super Resolución modificó efectivamente la geometría del video, pasando de la entrada original a una salida de mayor dimensión en ambos casos:

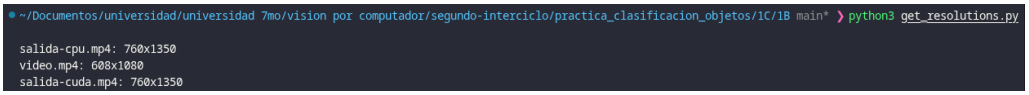


Figura 5: Resoluciones de los videos.

El video original y los procesados se encuentran en esta carpeta de Google Drive: https://drive.google.com/drive/folders/1vZbnvjafUPChQpK_Rw0-aESchYF9g6R2?usp=sharing.

Estos datos demuestran que, aunque ambos dispositivos logran el resultado teórico (la mejora de imagen), solo la GPU es viable para aplicaciones prácticas de video.

IV. Resolución Parte 1 C

En esta sección final se aborda la optimización extrema mediante el uso de C++ y la API nativa `opencv_cuda`. A diferencia de la Parte 1B, que utilizaba modelos de Deep Learning pre-entrenados, aquí se implementó un algoritmo de visión clásica ”desde cero”, diseñado para ejecutarse íntegramente en la memoria de la GPU para maximizar el rendimiento.

IV-A. Descripción de la Implementación

Se desarrolló una aplicación en C++ que establece un flujo de procesamiento (pipeline) secuencial. La característica crítica de esta implementación es la gestión de memoria: los datos se cargan una sola vez a la tarjeta gráfica y no regresan a la CPU hasta que todo el procesamiento ha finalizado, evitando así el cuello de botella del bus PCIe.

El pipeline implementado consta de las siguientes etapas:

1. **Upload:** Transferencia del frame de la cámara a la memoria VRAM (`cv::cuda::GpuMat`).
2. **Pre-procesamiento:** Conversión de color a escala de grises.
3. **Filtrado Gaussiano:** Aplicación de un kernel de 5×5 ($\sigma = 1{,}5$) para reducción de ruido.
4. **Ecualización:** Mejora del contraste global mediante histograma en GPU.
5. **Morfología Matemática:** Operación de erosión con elemento estructurante rectangular de 3×3 para limpiar impurezas.
6. **Detección de Bordes:** Algoritmo de Canny con umbrales de histéresis 50/150.
7. **Download:** Descarga final de la imagen binaria resultante para su visualización.

IV-B. Evidencia Audiovisual

Como respaldo de la ejecución en tiempo real, se adjuntan las grabaciones del funcionamiento del sistema en ambas arquitecturas:

- **Ejecución en GPU (RTX 3070):** https://www.youtube.com/shorts/4KmvtXnmVpk?feature=share
  Se observa fluidez total y una latencia imperceptible.
- **Ejecución en CPU (i9-12900K):** https://www.youtube.com/shorts/lGVJN90FgAg?feature=share
  Aunque funcional, consume recursos del procesador principal.

IV-C. Análisis de Resultados (Benchmarking)

Se realizaron pruebas de estrés procesando un video de prueba de 404 frames. Los resultados obtenidos demuestran la superioridad de la arquitectura paralela incluso en operaciones matemáticas simples.

Métrica	**CPU (OpenCV Clásico)**	**GPU (OpenCV CUDA)**
Hardware	Intel Core i9-12900K	NVIDIA RTX 3070
Tiempo Promedio	2.45 ms/frame	**1.25 ms/frame**
Velocidad (FPS)	407.4 FPS	**800.7 FPS**
FPS Máximos	1034.5 FPS	**1103.7 FPS**

Cuadro I: Comparativa de rendimiento en el pipeline de filtrado C++.

Interpretación: Aunque el procesador i9-12900K es excepcionalmente potente, logrando superar los 400 FPS, la implementación en GPU prácticamente **duplica el rendimiento**, superando los 800 FPS sostenidos.

Más importante aún que la velocidad bruta, la implementación en GPU libera casi totalmente al procesador central (CPU), permitiéndole dedicarse a otras tareas del sistema operativo o lógica de control, mientras que la versión de CPU requiere un uso intensivo de los núcleos del procesador para mantener esa tasa de cuadros.

V. CONCLUSIONES

El desarrollo de esta práctica experimental ha permitido validar empíricamente la superioridad de las arquitecturas de computación heterogénea (CPU + GPU) frente al procesamiento secuencial tradicional en tareas de visión artificial moderna. A partir de los resultados obtenidos, se concluye lo siguiente:

- **Precisión en Segmentación de Instancias (Parte 1A):**
  El entrenamiento de modelos de segmentación (YOLOv8-Seg) demostró ser eficaz para aplicaciones de agricultura de precisión, logrando delimitar morfologías complejas de frutas que una detección estándar (Bounding Box) no podría capturar. Las pruebas de inferencia confirmaron que es posible ejecutar estos modelos con tiempos de respuesta bajos, siempre y cuando se cuente con la infraestructura de hardware adecuada para la inferencia.
- **Necesidad de Aceleración en Generación de Imágenes (Parte 1B):**
  Se demostró que para tareas de *Deep Learning* generativo, como la Super Resolución con Real-ESRGAN, la CPU (Intel Core i9-12900K) es inviable para aplicaciones en tiempo real, alcanzando apenas $\sim$0.7 FPS y saturando los núcleos de procesamiento. En contraste, la aceleración por hardware (NVIDIA RTX 3070) permitió un flujo constante de $\sim$+2 FPS, evidenciando que el paralelismo masivo de los núcleos CUDA es un requisito obligatorio, y no opcional, para el despliegue de modelos generativos complejos.
- **Eficiencia en Pipelines de Bajo Nivel (Parte 1C):**
  En el procesamiento de imágenes clásico (filtrado y morfología), la implementación optimizada en C++ con `opencv_cuda` duplicó el rendimiento del sistema, elevando la tasa de procesamiento de 400 FPS (CPU) a más de 800 FPS (GPU). Más allá de la velocidad bruta, la conclusión crítica es la **liberación de recursos**: al mantener el pipeline de datos residente en la VRAM, se elimina la sobrecarga de transferencias por el bus PCIe y se libera al procesador central para otras tareas de control lógico del sistema.
- **Impacto de la Gestión de Memoria:**
  Tanto en la Parte 1B como en la 1C, se observó que el factor limitante no siempre es la potencia de cómputo, sino el ancho de banda de memoria. La estrategia de T̈upload-Process-Download"(subir datos una vez, procesar todo en GPU y descargar solo el resultado final) demostró ser la arquitectura de software más eficiente para minimizar la latencia en sistemas de visión por computador.

ANEXOS

Se incluye el enlace al repositorio con todo el código utilizado para la resolución de la práctica:

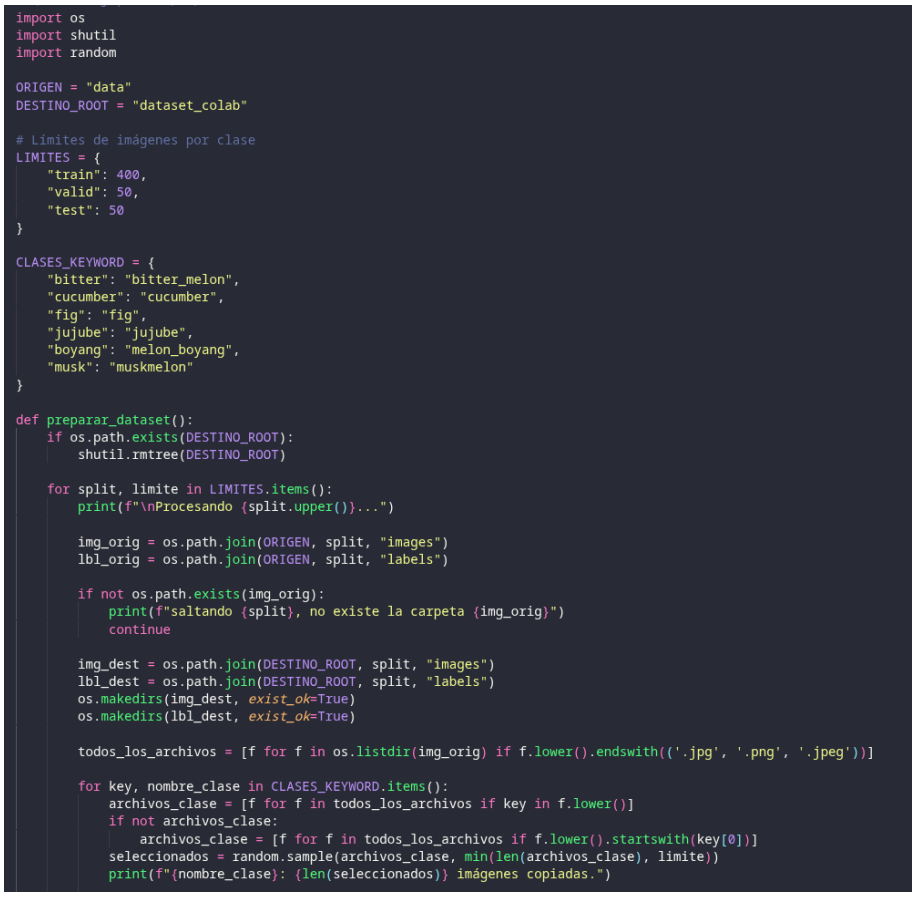
- **Repositorio GitHub:** https://github.com/FepDev25/Practica-Clasificacion-de-Objetos

1A) Código entrenamiento en Google Colab:

```
from google.colab import drive
import os
import shutil
from ultralytics import YOLO

# montar Google Drive
drive.mount('/content/drive')

# instalar YOLO (Ultralytics)
!pip install ultralytics


# copiar el zip de Drive al entorno de Colab
# ajusta la ruta si lo pusiste en una carpeta dentro de Drive
!cp "/content/drive/MyDrive/dataset_colab.zip" "/content/"

# descomprimir
!unzip -q "/content/dataset_colab.zip" -d "/content/"

print("Dataset descomprimido correctamente.")


# cargar modelo n (nano) de segmentación
model = YOLO('yolo11n-seg.pt')

# entrenar
results = model.train(
    data='/content/dataset_colab/data.yaml',
    epochs=50,
    imgsz=640,
    batch=16,
    device=0,           # Usa la GPU Tesla T4 de Colab
    project='/content/drive/MyDrive/Salida_Final',
    name='modelo_frutas_v3',
    save=True
)

# validar con el set de validación
model.val()
```

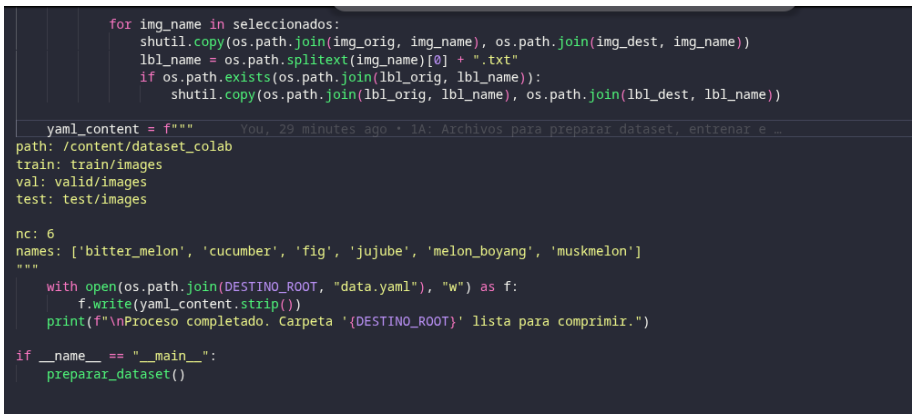
Figura 6: Código ejercicio 1A Parte 1

1A) Código preparar dataset:



```python
import os
import shutil
import random

ORIGEN = "data"
DESTINO_ROOT = "dataset_colab"

# Límites de imágenes por clase
LIMITES = {
    "train": 400,
    "valid": 50,
    "test": 50
}

CLASES_KEYWORD = {
    "bitter": "bitter_melon",
    "cucumber": "cucumber",
    "fig": "fig",
    "jujube": "jujube",
    "boyang": "melon_boyang",
    "musk": "muskmelon"
}

def preparar_dataset():
    if os.path.exists(DESTINO_ROOT):
        shutil.rmtree(DESTINO_ROOT)

    for split, limite in LIMITES.items():
        print(f"\nProcesando {split.upper()}...")

        img_orig = os.path.join(ORIGEN, split, "images")
        lbl_orig = os.path.join(ORIGEN, split, "labels")

        if not os.path.exists(img_orig):
            print(f"saltando {split}, no existe la carpeta {img_orig}")
            continue

        img_dest = os.path.join(DESTINO_ROOT, split, "images")
        lbl_dest = os.path.join(DESTINO_ROOT, split, "labels")
        os.makedirs(img_dest, exist_ok=True)
        os.makedirs(lbl_dest, exist_ok=True)

        todos_los_archivos = [f for f in os.listdir(img_orig) if f.lower().endswith(('.jpg', '.png', '.jpeg'))]

        for key, nombre_clase in CLASES_KEYWORD.items():
            archivos_clase = [f for f in todos_los_archivos if key in f.lower()]
            if not archivos_clase:
                archivos_clase = [f for f in todos_los_archivos if f.lower().startswith(key[0])]
            seleccionados = random.sample(archivos_clase, min(len(archivos_clase), limite))
            print(f"{nombre_clase}: {len(seleccionados)} imágenes copiadas.")
```

Figura 7: Código ejercicio 1A Parte 2



```python
            for img_name in seleccionados:
                shutil.copy(os.path.join(img_orig, img_name), os.path.join(img_dest, img_name))
                lbl_name = os.path.splitext(img_name)[0] + ".txt"
                if os.path.exists(os.path.join(lbl_orig, lbl_name)):
                    shutil.copy(os.path.join(lbl_orig, lbl_name), os.path.join(lbl_dest, lbl_name))

    yaml_content = f"""      You, 29 minutes ago • 1A: Archivos para preparar dataset, entrenar e …
path: /content/dataset_colab
train: train/images
val: valid/images
test: test/images

nc: 6
names: ['bitter_melon', 'cucumber', 'fig', 'jujube', 'melon_boyang', 'muskmelon']
"""
    with open(os.path.join(DESTINO_ROOT, "data.yaml"), "w") as f:
        f.write(yaml_content.strip())
    print(f"\nProceso completado. Carpeta '{DESTINO_ROOT}' lista para comprimir.")

if __name__ == "__main__":
    preparar_dataset()
```

Figura 8: Código ejercicio 1A Parte 3

1A) Código webcam:

```python
import cv2
import time
import os
from ultralytics import YOLO


RUTA_MODELO = 'best.pt'

if not os.path.exists(RUTA_MODELO):
    print(f"ERROR: No encuentro el modelo en: {RUTA_MODELO}")
    print("   -> Revisa la carpeta 'runs' o 'Frutas_Project' en tu explorador de archivos")
    print("   -> Copia la ruta del archivo 'best.pt' y pégala en RUTA_MODELO")
    exit()
else:
    print(f"Modelo encontrado. Cargando en GPU...")


DISPOSITIVO = 0

model = YOLO(RUTA_MODELO)
cap = cv2.VideoCapture(0)

while True:
    ret, frame = cap.read()
    if not ret: break

    start = time.time()



    results = model.predict(frame, device=DISPOSITIVO, retina_masks=True, conf=0.5, verbose=False)

    end = time.time()
    fps = 1 / (end - start)

    annotated_frame = results[0].plot()

    cv2.putText(annotated_frame, f"YOLOv11 (Frutas) | GPU: {DISPOSITIVO} | FPS: {fps:.1f}", (20, 50),
                cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)

    cv2.imshow("Parte 1A - Demo Frutas", annotated_frame)

    if cv2.waitKey(1) & 0xFF == 27:
        break

cap.release()
cv2.destroyAllWindows()      You, 28 minutes ago • 1A: Archivos para preparar dataset, entrenar e …
```

Figura 9: Código ejercicio 1A Parte 4

1A) Google Colab donde se realizó el entrenamiento:

- **Google Colab** https://colab.research.google.com/drive/1qEvVwN9M9S0QtWM1chqN1Dux0g43OSAD?usp=sharing

1B) Código:

```python
import cv2
import torch
import argparse
import sys
import time
import psutil
import uuid
from ultralytics import YOLO
import numpy as np

try:
    from basicsr.archs.rrdbnet_arch import RRDBNet
    from realesrgan import RealESRGANer
    HAS_REALESRGAN = True
except Exception as e:
    HAS_REALESRGAN = False
    print(f"ERROR CRÍTICO AL IMPORTAR IA: {e}")

def get_mac_address():
    mac = uuid.getnode()
    return ':'.join(('%012X' % mac)[i:i+2] for i in range(0, 12, 2))

def parse_args():
    parser = argparse.ArgumentParser(description='YOLO + Super Resolution')
    parser.add_argument('input', nargs='?', default='0', help='Video path or camera index')
    parser.add_argument('--output', '-o', type=str, help='Output video path')
    parser.add_argument('--device', '-d', choices=['cpu', 'cuda', 'auto'], default='auto', help='Device')
    parser.add_argument('--yolo-model', default='yolo11n.pt', help='YOLO model')
    parser.add_argument('--no-sr', action='store_true', help='Disable Super Resolution')
    return parser.parse_args()

args = parse_args()

if args.device == 'auto':
    DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'
else:
    DEVICE = args.device

print(f"Corriendo en: {DEVICE.upper()}")

print("Cargando YOLO...")
try:
    model_yolo = YOLO(args.yolo_model)
except:
    print(f"No se encontró {args.yolo_model}, bajando yolov8n.pt...")
    model_yolo = YOLO('yolov8n.pt')

model_yolo.to(DEVICE)

upsampler = None
```

Figura 10: Código ejercicio 1B Parte 1

```
    print("Cargando Real-ESRGAN...")
    try:
        model_sr = RRDBNet(num_in_ch=3, num_out_ch=3, num_feat=64, num_block=23, num_grow_ch=32, scale=2)
        upsampler = RealESRGANer(
            scale=2,
            model_path='https://github.com/xinntao/Real-ESRGAN/releases/download/v0.2.1/RealESRGAN_x2plus.pth',
            model=model_sr,
            tile=0,
            tile_pad=10,
            pre_pad=0,
            half=(DEVICE == 'cuda'),
            device=torch.device(DEVICE)
        )
    except Exception as e:
        print(f"Error cargando SR: {e}")
        upsampler = None

try:
    input_src = int(args.input) if args.input.isdigit() else args.input
except ValueError:
    input_src = args.input

cap = cv2.VideoCapture(input_src)
if not cap.isOpened():
    sys.exit("Error abriendo video/cámara")

cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

writer = None
fps_input = cap.get(cv2.CAP_PROP_FPS) if cap.get(cv2.CAP_PROP_FPS) > 1 else 30.0

window_name = 'YOLO + SR (Monitor de Rendimiento)'
cv2.namedWindow(window_name, cv2.WINDOW_NORMAL)

print("Presiona 'q' para salir.")

# Variables para métricas
prev_time = 0
mac_address = get_mac_address()
ram_usage_list = []

while True:
    start_loop = time.time()
    ret, frame = cap.read()
    if not ret: break

    # detección YOLO
    results = model_yolo(frame, device=DEVICE, verbose=False)
    annotated_frame = results[0].plot()
```

Figura 11: Código ejercicio 1B Parte 2

```
    final_frame = annotated_frame
    if not args.no_sr:
        if upsampler:
            try:
                final_frame, _ = upsampler.enhance(annotated_frame, outscale=1.25)
                final_frame = np.ascontiguousarray(final_frame)
            except RuntimeError:
                final_frame = cv2.resize(annotated_frame, None, fx=1.25, fy=1.25, interpolation=cv2.INTER_CUBIC)
        else:
            final_frame = cv2.resize(annotated_frame, None, fx=1.25, fy=1.25, interpolation=cv2.INTER_CUBIC)

    # calcular las métricas
    curr_time = time.time()
    fps = 1 / (curr_time - prev_time) if prev_time != 0 else 0
    prev_time = curr_time      FepDev25, yesterday • 1B primera parte

    ram_usage = psutil.virtual_memory().percent
    ram_usage_list.append(ram_usage)

    vram_info = ""
    if DEVICE == 'cuda':
        vram_mb = torch.cuda.memory_reserved(0) / 1024 / 1024
        vram_info = f"VRAM: {vram_mb:.0f} MB"

    # dibujar información en pantalla
    color = (0, 255, 0) if DEVICE == 'cuda' else (0, 0, 255)

    cv2.putText(final_frame, f"FPS: {fps:.2f}", (20, 40), cv2.FONT_HERSHEY_SIMPLEX, 1, color, 2)
    cv2.putText(final_frame, f"Mode: {DEVICE.upper()}", (20, 80), cv2.FONT_HERSHEY_SIMPLEX, 0.7, color, 2)
    cv2.putText(final_frame, f"RAM: {ram_usage}%", (20, 120), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255), 2)
    if vram_info:
        cv2.putText(final_frame, vram_info, (20, 150), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255), 2)

    cv2.putText(final_frame, f"MAC: {mac_address}", (20, final_frame.shape[0] - 20),
                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (200, 200, 200), 1)

    if args.output:
        if writer is None:
            h, w = final_frame.shape[:2]
            fourcc = cv2.VideoWriter_fourcc(*'mp4v')
            writer = cv2.VideoWriter(args.output, fourcc, fps_input, (w, h))
            print(f"Grabando en: {args.output} ({w}x{h})")
        writer.write(final_frame)

    cv2.imshow(window_name, final_frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
```

Figura 12: Código ejercicio 1B Parte 3

```
cap.release()
if writer: writer.release()
cv2.destroyAllWindows()

print("Resumen de Ejecución")
print(f"Dispositivo usado  : {DEVICE.upper()}")
if ram_usage_list:
    avg_ram = sum(ram_usage_list) / len(ram_usage_list)
    max_ram = max(ram_usage_list)
    print(f"RAM Promedio       : {avg_ram:.2f}%")
    print(f"RAM Máxima         : {max_ram:.2f}%")
if DEVICE == 'cuda':
    final_vram = torch.cuda.memory_allocated(0) / 1024 / 1024
    print(f"VRAM Final (Alloc) : {final_vram:.2f} MB")
print(f"MAC Address        : {mac_address}")
print("="*40 + "\n")
```

Figura 13: Código ejercicio 1B Parte 4

1B) Enlace a los videos: Grabaciones de Pantalla para parte GPU, CPU y video original, procesado con GPU y procesado con CPU:

- **Youtube grabación de pantalla GPU:** https://youtu.be/hHQVw1QqHYY
- **Youtube grabación de pantalla CPU:** https://youtu.be/_SPr1W4mVk0
- **Google Drive Videos:** https://drive.google.com/drive/folders/1vZbnvjafUPChQpK_Rw0-aESchYF9g6R2?usp=sharing

1C) Código:

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <chrono>
#include <iomanip>
#include <sstream>
#include <limits>
#include <opencv2/opencv.hpp>      You, yesterday • First commit
#include <opencv2/core/utils/logger.hpp>

#define ENABLE_CUDA

#ifdef ENABLE_CUDA
#include <opencv2/cudaimgproc.hpp>
#include <opencv2/cudafilters.hpp>
#include <opencv2/cudaarithm.hpp>
#endif

using namespace std;
using namespace cv;

double getFPS(chrono::time_point<chrono::high_resolution_clock> start) {
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double> diff = end - start;
    return 1.0 / diff.count();
}

int main(int argc, char* argv[]) {
    cv::utils::logging::setLogLevel(cv::utils::logging::LOG_LEVEL_ERROR);

    if (argc < 2) {
        cerr << "Uso: " << argv[0] << " <ruta_video> [salida.avi] [--cpu|--gpu]" << endl;
        return -1;
    }

    bool save_output = false;
    string output_path;
    bool force_cpu = false;
    bool force_gpu = false;

    for (int i = 2; i < argc; ++i) {
        string arg = argv[i];
        if (arg == "--cpu") {
            force_cpu = true;
        } else if (arg == "--gpu") {
            force_gpu = true;
        } else if (!save_output) {
            save_output = true;
            output_path = arg;
        } else {
            cerr << "Argumento ignorado: " << arg << endl;
        }
    }
```

Figura 14: Código ejercicio 1C Parte 1

```
    VideoCapture cap(argv[1]);
    if (!cap.isOpened()) {
        cerr << "Error: No se puede abrir el video: " << argv[1] << endl;
        return -1;
    }

    double input_fps = cap.get(CAP_PROP_FPS);
    if (input_fps <= 1.0) input_fps = 30.0;
    cap.set(CAP_PROP_FRAME_WIDTH, 1920);
    cap.set(CAP_PROP_FRAME_HEIGHT, 1080);

    Mat frame, result_frame;
    Mat gray, blur, hist, eroded;
    VideoWriter writer;
    size_t frame_count = 0;
    double accum_ms = 0.0;
    double max_fps = 0.0;
    double min_fps = std::numeric_limits<double>::max();

    bool use_cuda = false;

#ifdef ENABLE_CUDA
    use_cuda = !force_cpu;
    if (force_gpu) use_cuda = true;
#else
    use_cuda = false;
    if (force_gpu) {
        cerr << "CUDA no disponible en la compilacion; se usara CPU." << endl;
    }
#endif

#ifdef ENABLE_CUDA
    cuda::GpuMat d_frame, d_gray, d_blur, d_hist, d_eroded, d_edges;
    Ptr<cuda::Filter> gaussFilter;
    Ptr<cuda::Filter> erodeFilter;
    Ptr<cuda::CannyEdgeDetector> cannyFilter;
#endif

    if (use_cuda) {
#ifdef ENABLE_CUDA
        cout << ">>> MODO: GPU (CUDA) ACTIVADO <<<" << endl;
        gaussFilter = cuda::createGaussianFilter(CV_8UC1, CV_8UC1, Size(5, 5), 1.5);
        Mat element = getStructuringElement(MORPH_RECT, Size(3, 3));
        erodeFilter = cuda::createMorphologyFilter(MORPH_ERODE, CV_8UC1, element);
        cannyFilter = cuda::createCannyEdgeDetector(50, 150);
#endif
    } else {
        cout << ">>> modo CPU <<<" << endl;
    }

    while (true) {
        auto start_time = chrono::high_resolution_clock::now();
```

Figura 15: Código ejercicio 1C Parte 2

```
        cap >> frame;
        if (frame.empty()) break;

    if (use_cuda) {
#ifdef ENABLE_CUDA
        // pipeline en GPU
        d_frame.upload(frame);
        cuda::cvtColor(d_frame, d_gray, COLOR_BGR2GRAY);
        gaussFilter->apply(d_gray, d_blur);
        cuda::equalizeHist(d_blur, d_hist);
        erodeFilter->apply(d_hist, d_eroded);
        cannyFilter->detect(d_eroded, d_edges);
        d_edges.download(result_frame);
#endif
    } else {
        // pipeline en CPU
        cvtColor(frame, gray, COLOR_BGR2GRAY);
        GaussianBlur(gray, blur, Size(5, 5), 1.5);
        equalizeHist(blur, hist);
        erode(hist, eroded, getStructuringElement(MORPH_RECT, Size(3, 3)));
        Canny(eroded, result_frame, 50, 150);
    }
        double fps = getFPS(start_time);
        double elapsed_ms = 1000.0 / fps;

        frame_count++;
        accum_ms += elapsed_ms;
        max_fps = std::max(max_fps, fps);
        min_fps = std::min(min_fps, fps);

        std::ostringstream ms_label;
        ms_label << std::fixed << std::setprecision(2) << elapsed_ms;

        string device_tag = use_cuda ? "GPU (CUDA)" : "CPU";
        putText(result_frame, "Dispositivo: " + device_tag, Point(10, 30), FONT_HERSHEY_SIMPLEX, 0.7, Scalar(255),
        putText(result_frame, "FPS: " + to_string((int)fps), Point(10, 60), FONT_HERSHEY_SIMPLEX, 0.7, Scalar(255)
        putText(result_frame, "Tiempo: " + ms_label.str() + " ms", Point(10, 90), FONT_HERSHEY_SIMPLEX, 0.7, Scala

        if (save_output) {
            if (!writer.isOpened()) {
                int fourcc = VideoWriter::fourcc('M', 'J', 'P', 'G');
                bool ok = writer.open(output_path, fourcc, input_fps, result_frame.size(), false);
                if (!ok) {
                    cerr << "No se pudo abrir el archivo de salida: " << output_path << endl;
                }
            }
            if (writer.isOpened()) {
                writer.write(result_frame);
            }
        }

        imshow("Laboratorio Vision - Pipeline", result_frame);
```

Figura 16: Código ejercicio 1C Parte 3

```
        if (waitKey(1) == 27) break; // ESC para salir
    }

    if (frame_count > 0) {
        double avg_ms = accum_ms / static_cast<double>(frame_count);
        double avg_fps = 1000.0 / avg_ms;
        cout << "Resumen de ejecucion" << endl;
        cout << "  Frames procesados: " << frame_count << endl;
        cout << "  Promedio: " << fixed << setprecision(2) << avg_ms << " ms/frame (" << setprecision(1) << avg_fps << " fps)" << endl;
        cout << "  Mejor fps: " << fixed << setprecision(1) << max_fps << " | Peor fps: " << min_fps << endl;
    }

    return 0;
}
```

Figura 17: Código ejercicio 1C Parte 4

1C) Enlace a los videos: Resultado de Pipeline con GPU y CPU:

- **Youtube pipeline GPU:** https://www.youtube.com/shorts/4KmvtXnmVpk?feature=share
- **Youtube pipeline CPU:** https://www.youtube.com/shorts/lGVJN90FgAg?feature=share

REFERENCIAS

[1] figshare, "Fruit instance segmentation dataset for YOLOvFIS network," 2 2025. [Online]. Available: https://figshare.com/articles/dataset/_b_Fruit_instance_segmentation_dataset_for_b_b_YOLOvFIS_network_b_/28436618?file=52444568

[2] Ultralytics, "Ultralytics YOLO11," 12 2025. [Online]. Available: https://docs.ultralytics.com/es/models/yolo11/

[3] Xinntao, "GitHub - xinntao/Real-ESRGAN: Real-ESRGAN aims at developing Practical Algorithms for General Image/Video Restoration." [Online]. Available: https://github.com/xinntao/Real-ESRGAN

[4] E. P. Digital, "El baile de Trump que se volvió viral en TikTok," 10 2020. [Online]. Available: https://www.youtube.com/watch?v=WcWVqMe-voY