

Universidad Mariano Gálvez de Guatemala

Sede Central

Arquitectura de computadoras II

Sección "A"

Grupo #5

Modelo de detección de productos con YOLO26 Nano para inferencia local en el borde

Javier Alejandro Barillas González
Kenneth Gerardo García Lemus
Hugo Daniel Velásquez Patzan

0900-23-3058
0900-23-3445
0900-23-6212

Contenido

Introducción y objetivo	3
Enlace del video de defensa	3
Problema que buscamos resolver	3
Arquitectura y Hardware	4
Importancia del Switch	5
Raspberry PI 400	6
YOLO26 Nano	7
Por qué entrenar un modelo propio	8
Entorno de trabajo	8
Instalación y prueba inicial en Debian	9
Entrenamiento de un detector de manos	10
Dataset de productos	11
Entrenamiento en Windows con GPU	13
Justificación	13
Configuración y versiones del modelo	14
Versiones 3 y 4: entrenamiento incremental	15
Resultados	15
Resultados de la versión 2	15
Comparación entre la versión 1 y la versión 2	17
Versión 3: supermercado	18
Versión 4: snacks	18
Evolución de las 4 versiones	20
Pruebas de inferencia	20
Despliegue en el clúster	22
Arquitectura	22
Red	22
Servicio de inferencia en la máquina virtual	22
Caja de autoservicio en la Raspberry	23
Cámara	23
Funcionamiento de la caja	23
Resultados del sistema completo	24
Problemas encontrados y soluciones	24
Conclusiones	25
Referencias	26

Introducción y objetivo

En esta sección se documenta el desarrollo del modelo de visión por computadora del proyecto, este caso pide desplegar un modelo YOLO nano para detectar y clasificar productos de forma local, mediante una cámara conectada directamente al clúster y sin comunicación con internet.

El objetivo fue obtener un modelo YOLO26 Nano capaz de reconocer productos de consumo y objetos cotidianos usando solo el procesador sin tener que depender de una GPU, ya que los nodos del clúster son equipos de bajo consumo que no la tienen. Para comprobarlo, el modelo se probó en una máquina virtual Debian, que no tiene acceso a la GPU de la laptop y por eso se comporta de forma similar a un nodo del clúster. El trabajo incluyó preparar el entorno, validar el proceso de entrenamiento con una prueba de concepto, armar un dataset de productos y entrenar 4 versiones sucesivas del modelo, que fueron ampliando lo que reconoce, de 137 clases en la versión 1 a 194 en la versión 4, que es el modelo final.

Enlace del video de defensa

ProyectoFinal_Grupo5.mp4

Link directo:

https://umgt-my.sharepoint.com/:v:/g/personal/jbarillasg8_miumg_edu_gt/IQAMk1H9Qy08QoySG2PJ4RW5ASIB4bVOYWQ3JOMgR1goYVg?nav=eyJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJPbmVEcmI2ZUZvckJ1c2luZXNzIiwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsTW9kZSI6InZpZXciLCJyZWZlcnJhbFZpZXciOiJNeUZpbGVzTGlua0NvcHkifX0&e=NFuwJn

Problema que buscamos resolver

En los supermercados, el registro de los productos durante el proceso de compra puede requerir que cada artículo sea identificado manualmente o mediante el uso de códigos de barras. Esto puede generar demoras en el proceso de pago y depender de la intervención constante del usuario o del personal del establecimiento.

Para abordar esta problemática, se propone desarrollar un sistema de caja de autoservicio basado en visión artificial, capaz de identificar productos mediante una cámara y registrar automáticamente sus precios en un ticket de compra. Para ello, se utiliza un modelo de detección de objetos YOLO Nano, entrenado para reconocer distintas categorías de productos.

La solución se implementa mediante una arquitectura distribuida: una Raspberry Pi 400 ejecuta la aplicación web de autoservicio y administra el ticket de compra, mientras que una máquina virtual con Debian ejecuta el servicio de inferencia, encargado de procesar las imágenes y devolver los productos detectados. Ambos equipos se comunican mediante una red local Ethernet a través de un switch.

El sistema permite escanear productos, consultar sus precios, modificar el ticket y simular el pago en efectivo o con tarjeta, sin necesidad de procesar las imágenes mediante servicios externos de inteligencia

artificial. Además, los servicios se inician mediante scripts de shell, lo que permite poner en funcionamiento los componentes de la aplicación de manera independiente.

De esta forma, se busca reducir la intervención manual en el registro de productos y demostrar que es posible implementar una solución de autoservicio mediante inferencia de inteligencia artificial en el borde (Edge Computing), distribuyendo las tareas entre dispositivos conectados localmente y manteniendo la funcionalidad principal sin depender de una conexión a Internet.

Arquitectura y Hardware

El proyecto implementa una infraestructura de Edge computing orientada a la inferencia local de inteligencia artificial mediante un clúster compuesto por una Raspberry Pi 400 y una máquina virtual Debian que se aloja en una computadora dentro de la red local. La infraestructura permite integrar recursos de hardware para poder ejecutar los componentes necesarios para el reconocimiento de productos mediante visión por computadora.

La ejecución de los servicios se gestiona mediante scripts sh, encargados de preparar el entorno e iniciar las aplicaciones correspondientes en cada dispositivo. En la máquina virtual Debian, el script `iniciar-inferencia.sh` activa el entorno virtual de Python y ejecuta el servicio de inferencia basado en FastAPI y Uvicorn, que utiliza el modelo YOLO26 Nano para detectar productos mediante visión por computadora.

La inferencia se ejecuta utilizando el procesador de la máquina virtual, sin acceso a la GPU de la computadora anfitriona. Esta configuración permite evaluar el funcionamiento del modelo en un entorno con recursos limitados, similar al de los dispositivos de bajo consumo empleados en soluciones de Edge computing.

La conectividad entre los equipos se establece mediante una interfaz Ethernet y un switch, el router de casa proporciona conexión de red cuando se requiere acceder a Internet para instalar paquetes, actualizar el sistema o descargar dependencias, la comunicación entre los nodos se realiza mediante la red local de forma aislada.

Para poder gestionar la aplicación web se levanta a partir del Raspberry Pi 400, el cual gestiona estos servicios mediante el puerto 8000 permitiendo así a los demás usuarios dentro de la red local acceder al servicio sin ninguna complicación, se usa un archivo JSON para poder mantener una estructura de datos ordenados en pares de nombre y valor, lo que permite leer la caja una sola vez cuando arranca.

Se utiliza SQLite como base de datos que vive en un solo archivo, sin necesidad de instalar servidores adicionales, este posee una tabla la cual maneja y registra las ventas realizadas dentro del sistema, insertándose los datos una vez se realice una venta, y mostrando el listado de ventas al acceder al apartado "Ventas de hoy"

Componente	Función dentro del proyecto
Raspberry Pi 400	Nodo principal del clúster y equipo para ejecutar los servicios de administración.
Computadora	Equipo que ejecuta VirtualBox y proporciona los recursos para la máquina virtual.
Máquina virtual Debian	Nodo de trabajo destinado a ejecutar el modelo YOLO26 Nano y realizar la inferencia de imágenes.
Cámara / iPhone con la app DroidCam	Dispositivo de captura de imágenes para detectar los productos del supermercado.
Switch	Interconecta los dispositivos mediante conexiones Ethernet.
Router doméstico	Proporciona conectividad con la red doméstica y acceso a Internet cuando está disponible.
YOLO Nano	Modelo IA de visión por computadora entrenado para reconocer las clases de productos definidas en el dataset.
JSON	Archivo de texto con datos ordenados que tiene el registro de productos y su precio correspondiente definidos.
SQLite	Base de datos que permite visualizar e insertar registros de ventas realizadas en el sistema.

Importancia del Switch

El switch es un componente fundamental para poder establecer la comunicación física entre los dispositivos que forman parte del proyecto, su función consiste en interconectar los equipos mediante Ethernet y permitir el intercambio de datos dentro de la red local.

Este modelo cuenta con cinco puertos Gigabit Ethernet y es de tipo no administrable, por lo que no requiere una configuración específica de VLAN o enrutamiento para poder establecer una comunicación básica entre los equipos conectados.

El switch permite conectar la Raspberry Pi 400 y la computadora a la infraestructura de red, la máquina virtual utiliza el adaptador Ethernet de la computadora en modo puente para así poder comunicarse como un equipo independiente dentro de la red.

El router doméstico se conecta directamente al switch para poder proporcionar acceso a internet cuando es necesario instalar paquetes, actualizar el sistema operativo o descargar dependencias. Sin embargo, la función de interconexión entre los dispositivos locales corresponde al switch, mientras que el router se encarga de proporcionar conectividad entre redes, por esto es que se configuraron redes est.

La incorporación del switch nos da las siguientes ventajas por las cuales optamos para implementarlo en el proyecto:

- **Conectividad Ethernet:** permite establecer enlaces físicos entre la Raspberry Pi y la computadora que ejecuta la máquina virtual.

- **Comunicación local:** facilita el intercambio de información entre los nodos sin depender de una conexión Wi-Fi.
- **Velocidad de enlace:** sus puertos Gigabit Ethernet admiten velocidades de hasta 1 Gbps, siempre que los dispositivos, las interfaces y el cableado sean compatibles.
- **Simplicidad de implementación:** al ser un switch no administrable, no requiere configurar parámetros avanzados para la comunicación básica.

Raspberry Pi 400

La Raspberry Pi 400 es una computadora compacta basada en una arquitectura ARM de 64 bits, diseñada para ofrecer capacidad de procesamiento en un formato de bajo consumo energético.

En este proyecto se utiliza como nodo principal del clúster de Edge Computing, su función es alojar los servicios necesarios para la administración del aplicativo y así poder proporcionar la infraestructura necesaria para consumir el modelo.

El dispositivo utilizado cuenta con 4 GB de memoria RAM y ejecuta Debian GNU/Linux 13 (Trixie) en arquitectura aarch64.

Sus principales características para el proyecto son:

- Procesador de arquitectura ARM de 64 bits.
- 4 GB de memoria RAM.
- Conectividad Ethernet para comunicarse con la máquina virtual.
- Consumo energético reducido en comparación con una computadora de escritorio convencional.
- Compatibilidad con sistemas Linux y herramientas de contenedores.
- Teclado incluido a la placa.

La utilización de la Raspberry Pi 400 permite demostrar cómo un equipo compacto puede participar en una infraestructura de Edge sin necesitar un servidor dedicado de alto consumo.

Antes de instalar los componentes del clúster, fue necesario poder comprobar el estado del sistema operativo, actualizar los repositorios y aplicar las actualizaciones disponibles.

Para esto primero se consultaron los datos del sistema con los siguientes comandos:

```
cat /etc/os-release
uname -m
free -h
df -h
```

Estos comandos permiten identificar la distribución instalada, la arquitectura del procesador, la memoria disponible y el espacio de almacenamiento.

En el dispositivo del proyecto se identificó Debian GNU/Linux 13 (Trixie) y arquitectura aarch64, luego de esto fue necesario actualizar los índices de los paquetes usando:

```
sudo apt update
sudo apt full-upgrade -y
sudo apt autoremove -y
```

La actualización del sistema permite disponer de versiones recientes de los paquetes disponibles en los repositorios configurados y reducir posibles incompatibilidades durante la instalación de las herramientas necesarias.

Se verifico la conectividad de red, los recursos del sistema y la disponibilidad de las herramientas necesarias, para comprobar la conectividad se ejecutó:

```
ip -br addr
ip route
ping -c 4 google.com
ping -c 4 deb.debian.org
```

El ultimo comando comprueba la conectividad con un dominio externo en este caso deb.debian.org el que contiene los paquetes en línea de debian y permite la instalación y actualización de estos, siempre que el acceso a Internet y la resolución DNS estén disponibles, para poder continuar con la actualización se usó:

```
sudo apt install -y curl ca-certificates iproute2 iputils-ping
```

Paquete	Función
curl	Permite realizar solicitudes HTTP y HTTPS y descargar recursos.
ca-certificates	Proporciona certificados de autoridades de confianza para conexiones TLS.
iproute2	Incluye herramientas como <code>ip</code> para consultar interfaces y rutas de red.
iputils-ping	Proporciona el comando <code>ping</code> para comprobar conectividad mediante ICMP.

Estos paquetes son útiles para la preparación y el diagnóstico del entorno, por esta razón es necesario poder tenerlos actualizados e instalados en el sistema para su funcionamiento correcto.

YOLO26 Nano

YOLO es una familia de modelos que detectan objetos en imágenes y video en tiempo real y analizan la imagen completa en una sola pasada. Se usó YOLO26, de Ultralytics, lanzado en enero de 2026.

Cada versión de YOLO tiene 5 tamaños que son nano, small, medium, large y extra large. Nano es el más pequeño porque reduce la profundidad de la red a la mitad y el ancho de sus canales a un cuarto. El modelo

base yolo26n tiene 2.4 millones de parámetros y pesa 5.3 MB por lo que es el que más sirve para dispositivos de borde como la Raspberry Pi, a cambio de una precisión algo menor que la de los modelos grandes.

Por qué entrenar un modelo propio

El modelo preentrenado de YOLO26 se entrenó con el dataset COCO, que tiene solo 80 clases de objetos generales, como personas, botellas, celulares y frutas. No distingue productos concretos, como una Coca-Cola de una Pepsi, ni reconoce objetos fuera de esas 80 clases. En las pruebas confundió un dedo con una zanahoria y unos lentes con un celular. Para detectar productos, el modelo debe entrenarse con imágenes de esos productos, partiendo del modelo preentrenado.

Entorno de trabajo

Se usó una laptop ASUS TUF Gaming F16 con Windows 11 y dentro de ella una máquina virtual Debian en VirtualBox. Debian se usó para la instalación y las pruebas de inferencia, porque representa un equipo sin GPU como los nodos del clúster. Los entrenamientos del modelo de productos se hicieron en Windows, con la GPU de la laptop, por las razones explicadas en la sección de entrenamiento.

Recurso	Windows, equipo anfitrión	Debian, máquina virtual
Sistema operativo	Windows 11 Home, build 26300	Debian 13.7
Procesador	Intel Core i7-14650HX, 16 núcleos y 24 hilos	4 hilos asignados, 2 al inicio
Memoria RAM	31.6 GB	10 GB asignados, 3.8 GB al inicio
GPU	NVIDIA GeForce RTX 5060 Laptop, 8 GB, CUDA 13.4	Ninguna: adaptador gráfico virtual
Python	3.12.10	3.13.5
PyTorch	2.14.1+cu130 con CUDA	2.14.1+cpu
Ultralytics	8.4.175	8.4.175
Virtualización	Oracle VirtualBox 7.2.4	

```

PS C:\WINDOWS\system32> Get-CimInstance Win32_ComputerSystem | Select-Object Manufacturer, Model, @{n="RAM_GB";e={[math]::Round($_.TotalPhysicalMemory/1GB,1)}}
Manufacturer      Model              RAM_GB
-----
ASUSTek COMPUTER INC. ASUS TUF Gaming F16 FX608JMR-FX608JMR 31.6

PS C:\WINDOWS\system32> Get-CimInstance Win32_Processor | Select-Object Name, NumberOfCores, NumberOfLogicalProcessors
Name              NumberOfCores      NumberOfLogicalProcessors
-----
Intel(R) Core(TM) i7-14650HX          16          24

PS C:\WINDOWS\system32> Get-CimInstance Win32_OperatingSystem | Select-Object Caption, Version, BuildNumber
Caption      Version      BuildNumber
-----
Microsoft Windows 11 Home 10.0.26300 26300

PS C:\WINDOWS\system32> nvidia-smi
Mon Oct 12 16:58:35 2026

+-----+
| NVIDIA-SMI 617.42                Driver Version: 617.42                CUDA UMD Version: 13.4                |
+-----+-----+
| GPU  Name      Perf      Driver-Model      Bus-Id      Memory-Usage      Disp.A      Volatile Uncorr. ECC      |
| Fan  Temp      Perf      Par:Usage/Cap      Memory-Usage      GPU-Util      Compute M.      MIG M.      |
+-----+-----+
| 0  NVIDIA GeForce RTX 5060 ... WDDM      00000000:01:00.0 Off      0%      Default      N/A      |
| N/A  58C      P0      12W / 95W      0MiB / 8151MiB      0%      Default      N/A      |
+-----+-----+

Processes:
+-----+
| GPU  GI  CI      PID  Type  Process name      GPU Memory      |
| ID   ID   ID      ID   Type  Process name      Usage      |
+-----+-----+
No running processes found

PS C:\WINDOWS\system32> & "C:\Program Files\Oracle\VirtualBox\VBBoxManage.exe" --version
7.2.4r178995
PS C:\WINDOWS\system32> & "C:\Program Files\Oracle\VirtualBox\VBBoxManage.exe" showvminfo "Debian" | findstr /i "Memory CPUs"
Memory size:
10000MB

```

Instalación y prueba inicial en Debian

En Debian, Ultralytics se instaló dentro de un entorno virtual de Python, porque desde Debian 12 el sistema no permite instalar paquetes de pip de forma global. Se usó la versión de PyTorch para CPU, más liviana que la versión con CUDA:

```

sudo apt install -y python3 python3-venv python3-pip libgl1 libgl1-dev
python3 -m venv ~/yolo-env
source ~/yolo-env/bin/activate
pip install torch torchvision --index-url https://download.pytorch.org/whl/cpu
pip install -U ultralytics

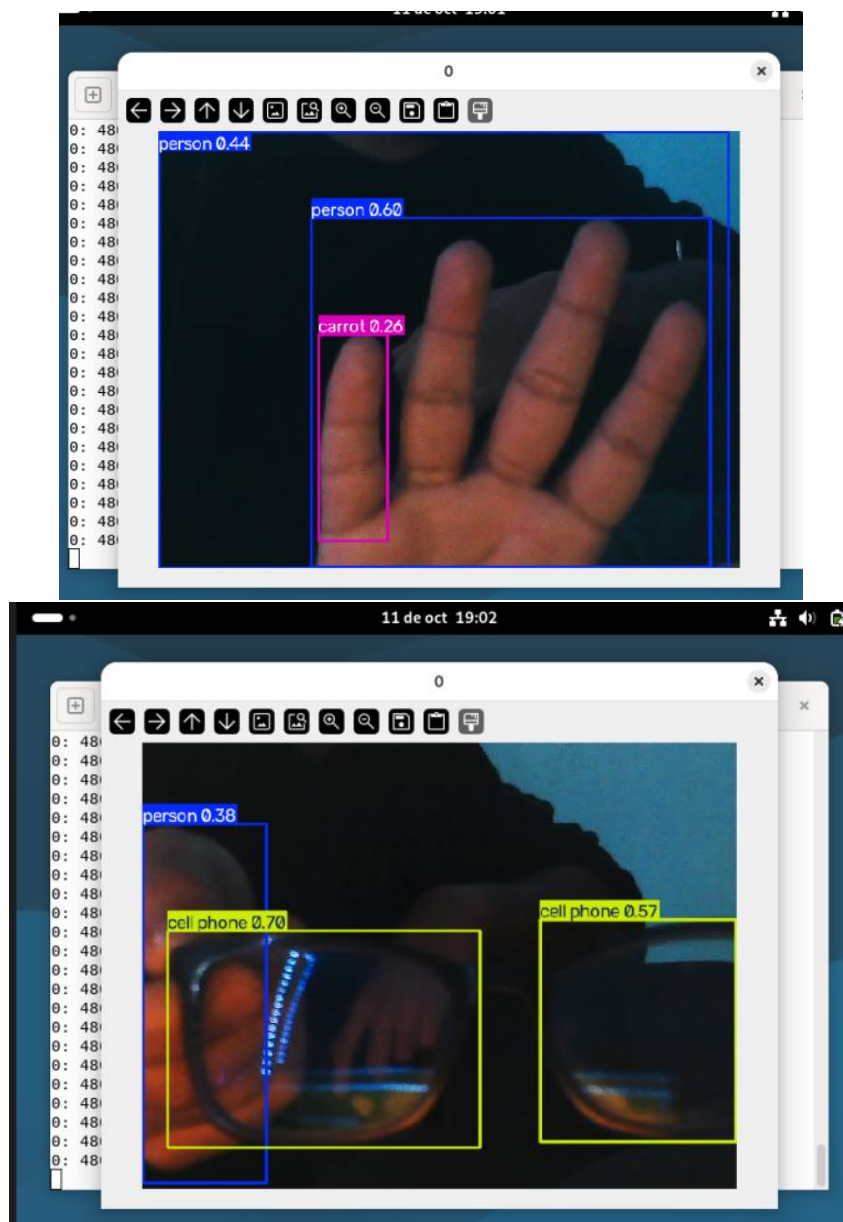
```

Para usar la cámara de la laptop en Debian hubo que activarla en VirtualBox. Con la cámara disponible, la prueba en tiempo real mostró las limitaciones del modelo genérico:

```

yolo predict model=yolo26n.pt source=0 show=True save=False

```



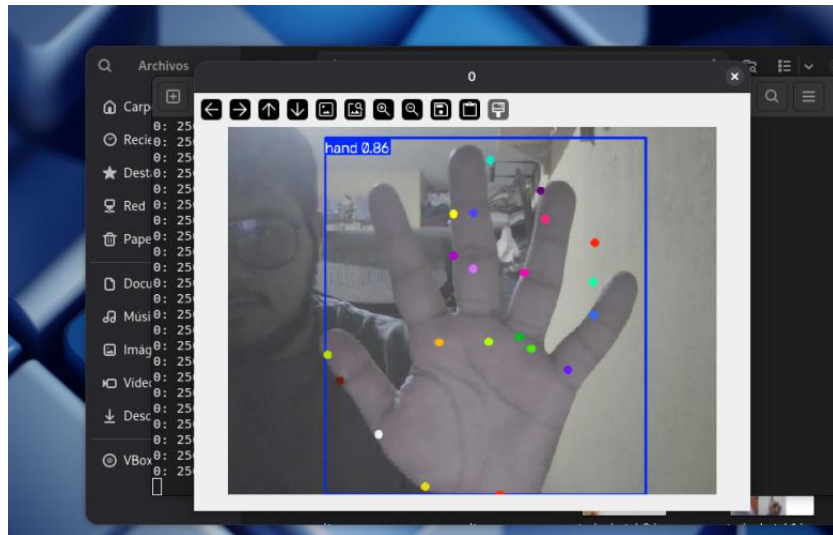
Entrenamiento de un detector de manos

Antes de armar el dataset de productos se hizo un entrenamiento pequeño para validar todo el proceso en Debian, desde la descarga de un dataset hasta el entrenamiento, la lectura de métricas y la prueba con la cámara. Se usó el dataset Hand Keypoints de Ultralytics para enseñarle al modelo a reconocer manos.

Parámetro	Valor
Modelo base	yolo26n-pose.pt
Imágenes de entrenamiento	1854, el 10% del dataset
Épocas	10
Tamaño de imagen	320px
Recursos	Debian, 4 núcleos y 10 GB de RAM, sin GPU
Tiempo total	30 minutos

```
yolo pose train model=yolo26n-pose.pt data=hand-keypoints.yaml epochs=10 imgsz=320 batch=8 fraction=0.1 val=False
```

El modelo aprendió a encontrar las manos con mucha precisión, aunque la ubicación de los 21 puntos de los dedos quedó baja, con 0.095, algo esperable en un entrenamiento tan corto. Frente a la cámara, donde el modelo original veía una zanahoria, el modelo entrenado detectó la mano con 0.86 de confianza.



Entrenar en la máquina virtual era muy lento. Con 2 núcleos y 640 px, una sola época tardaba 37 minutos, más de 12 horas para 20 épocas, y por eso hubo que reducir el entrenamiento al 10% del dataset y a 320 px. Esto definió que el modelo de productos, con decenas de miles de imágenes, se entrenaría en Windows con la GPU.

Dataset de productos

Cuando se entrena un modelo YOLO con clases nuevas, se reemplaza su última capa, la que decide qué es cada objeto, y el modelo olvida las clases que no estén en el nuevo entrenamiento. Un ejemplo es el detector de manos de la prueba de concepto, que solo reconoce manos. Para que el modelo final reconozca a la vez productos, objetos generales y personas, se armó un único dataset que reúne todas esas fuentes.

Fuente	Contenido	Imágenes train y val
COCO 2017	80 clases generales, como personas, botellas, celulares, mouse y frutas, con 500 imágenes por clase	19909 / 607
Hand Keypoints de Ultralytics	Manos, convertidas a un solo recuadro hand	2000 / 300
Roboflow: productos	Bebidas como Coca-Cola, Pepsi, Sprite y Fanta, limpieza como detergente, jabón y pasta dental, hogar, comida y verduras	6683 / 892
MVTec D2S	Productos de supermercado: frutas, verduras, pasta, cereal, café y bebidas	4853 / 1747
Roboflow: objetos cotidianos	Audífonos, cajas, lentes, billetera, llaves, lapicero, reloj de pulsera, cargador	3206 / 582
Roboflow: supermercado, versión 3	Frutas, pan, huevos, detergente, bebidas, leche, harina, jugos, fresas y frijoles en lata, de 2 datasets	8188 / 3745
Roboflow: snacks, versión 4	Lays, Doritos, Cheetos, Takis, Ruffles, Pringles, Snickers, chicharrones y otras bolsas de snacks, de 4 datasets	1779 / 492
Total	194 clases	46618 / 8365

El dataset de la versión 2 ocupaba 4.3 GB, y las versiones 3 y 4 le sumaron 14204 imágenes más. Se arma de forma automática con un script propio en Python que:

1. Descarga un subconjunto de COCO, eligiendo un número fijo de imágenes por clase para que todas queden representadas.
2. Descarga los datasets de Roboflow por su API y convierte MVTec D2S, que viene en formato COCO con archivos JSON.
3. Unifica los nombres de las clases y une con COCO las que ya existen ahí.
4. Marca automáticamente las personas que aparecen en las fotos de productos, que nadie había etiquetado, usando el modelo preentrenado.
5. Reduce las imágenes muy grandes a 1024 px y separa un 15% para validación en los datasets que no la traen.
6. Permite agregar datasets nuevos al dataset ya armado, sin rehacerlo, como se hizo en las versiones 3 y 4.

Las 60 categorías de D2S quedaron en 51 clases, al unir con COCO o con clases existentes las frutas, verduras y bebidas repetidas. Hasta la versión 3, ninguna clase tenía menos de 30 etiquetas y en la versión 4, la clase con menos ejemplos es Takis con 17 etiquetas, seguida de Ruffles con 56 y Cheetos con 64.

```
Windows PowerShell
PS C:\WINDOWS\system32> (Get-ChildItem C:\Users\USER\dataset_unificado\images\train).Count
36651
PS C:\WINDOWS\system32> (Get-ChildItem C:\Users\USER\dataset_unificado\images\val).Count
4128
PS C:\WINDOWS\system32> (Get-Content C:\Users\USER\dataset_unificado\clases.txt).Count
178
PS C:\WINDOWS\system32> "{0:N1} GB" -f (((Get-ChildItem C:\Users\USER\dataset_unificado -Recurse -File | Measure-Object Length -Sum).Sum/1GB)
4.3 GB
PS C:\WINDOWS\system32> |
```

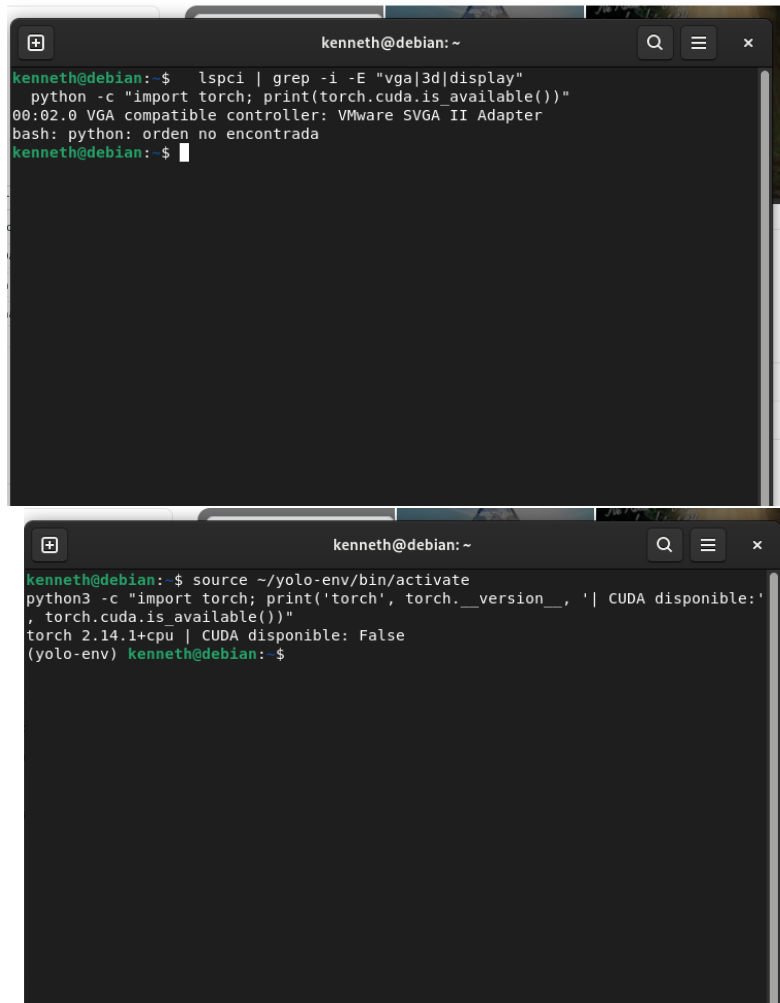
Entrenamiento en Windows con GPU

El modelo de productos se entrenó en Windows sobre el hardware completo de la laptop y no en la máquina virtual. Debian se mantuvo como entorno de inferencia, que es donde se ejecuta el modelo localmente y sin GPU.

Justificación

- **La máquina virtual no tiene acceso a la GPU.** VirtualBox le presenta a Debian un adaptador gráfico virtual, no la tarjeta NVIDIA, y PyTorch en Debian responde **CUDA disponible: False**. Sin GPU el entrenamiento solo puede usar el procesador.
- **La máquina virtual usa solo una parte del equipo:** 4 de los 24 hilos y 10 de los 31.6 GB de RAM.
- **La diferencia de velocidad es de unas 40 veces.** En Debian cada imagen tardaba unos 0.47 s en entrenarse y en Windows con GPU, unos 0.012 s. Entrenar la versión 2 en la máquina virtual habría tomado unos 5 días, frente a 6 horas en Windows.
- **No contradice las restricciones del proyecto.** El enunciado prohíbe usar computadoras convencionales o la nube en la etapa de inferencia, no en el entrenamiento. El archivo del modelo no depende del sistema operativo ni de la GPU, por lo que el mismo archivo entrenado en Windows funciona sin cambios en Debian y en los nodos del clúster.

Entrenamiento	Entorno	Imágenes	Tiempo por época
Manos, primer intento a 640 px	Debian, 2 núcleos, CPU	1854	37 min
Manos, final a 320 px	Debian, 4 núcleos, CPU	1854	3 min
Productos v1 a 640 px	Windows, GPU	17161	3.1 min
Productos v2 a 640 px	Windows, GPU	36651	7 min
Productos v3 incremental a 640 px	Windows, GPU	26513	5.0 a 5.1 min
Productos v4 incremental a 640 px	Windows, GPU	37650	7.1 a 7.5 min



The image shows two terminal windows from a Debian system. The top window shows the user running `lspci | grep -i -E "vga|3d|display"` to check for a graphics card, which returns `00:02.0 VGA compatible controller: VMware SVGA II Adapter`. Then, the user runs `python -c "import torch; print(torch.cuda.is_available())"`, which returns `False`. The bottom window shows the user activating a virtual environment with `source ~/yolo-env/bin/activate`. Then, they run `python3 -c "import torch; print('torch', torch.__version__, '| CUDA disponible:', torch.cuda.is_available())"`, which returns `torch 2.14.1+cpu | CUDA disponible: False`.

```
kenneth@debian: ~  
kenneth@debian:~$ lspci | grep -i -E "vga|3d|display"  
python -c "import torch; print(torch.cuda.is_available())"  
00:02.0 VGA compatible controller: VMware SVGA II Adapter  
bash: python: orden no encontrada  
kenneth@debian:~$  
  
kenneth@debian:~$ source ~/yolo-env/bin/activate  
python3 -c "import torch; print('torch', torch.__version__, '| CUDA disponible:',  
torch.cuda.is_available())"  
torch 2.14.1+cpu | CUDA disponible: False  
(yolo-env) kenneth@debian:~$
```

Configuración y versiones del modelo

Se entrenaron 4 versiones. Las versiones 1 y 2 se entrenaron a partir del modelo base `yolo26n.pt`, para poder compararlas en igualdad de condiciones: la versión 1 usó 200 imágenes de COCO por clase y no incluía D2S ni objetos cotidianos, y la versión 2 es la del dataset descrito arriba. Las versiones 3 y 4 son incrementales porque parten de la versión anterior y le agregan productos nuevos.

Parámetro	Versión 1	Versión 2	Versión 3	Versión 4
Punto de partida	yolo26n.pt	yolo26n.pt	unificado_v2.pt	unificado_v3.pt
Dataset	17161 train / 1353 val, 137 clases	36651 train / 4128 val, 178 clases	44839 train / 7873 val, 183 clases	46618 train / 8365 val, 194 clases
Imágenes usadas para entrenar	17161	36651	26513 con 8188 nuevas y 18325 de repaso al 50%	37650 con 1779 nuevas y 35871 de repaso al 80%
Épocas y tamaño de imagen	50 épocas, 640 px	50 épocas, 640 px	20 épocas, 640 px	20 épocas, 640 px
Lote con AutoBatch	15 imágenes	14 imágenes	11 imágenes	11 imágenes
Tiempo total	2.6 horas	6 horas 14 minutos	1 hora 44 minutos	2 horas 27 minutos

Versiones 3 y 4: entrenamiento incremental

Las versiones 3 y 4 no se entrenaron desde el inicio, sino que se agregaron productos nuevos al dataset y se continuó el entrenamiento de la versión anterior. Si el modelo se entrenara solo con las imágenes nuevas, olvidaría lo que ya sabía. Por eso, en cada ronda se usan todas las imágenes de entrenamiento nuevas más una parte de las anteriores, elegidas al azar, como repaso. Ese repaso fue del 50% en la versión 3 y del 80% en la versión 4. Las clases nuevas se agregan al final de la lista, sin cambiar el número de las existentes, y la validación usa todas las imágenes de validación, para medir a la vez lo aprendido y lo conservado.

Versión	Datasets agregados	Imágenes nuevas train y val	Clases nuevas	Tiempo de agregar y entrenar
Versión 3	Supermercado con grocery_picking_2 y Retail object detection	8188 / 3745	5: bebida, fresa, harina, jugo y leche	4 min y 1 h 44 min
Versión 4	Snacks: chips, cpvis, x2 y lays	1779 / 492	11: bolsa-snack, cheetos, chicharrón, chips-amarillas, chips-sal, doritos, lays, pringles, ruffles, snickers y takis	2 min y 2 h 27 min

Resultados

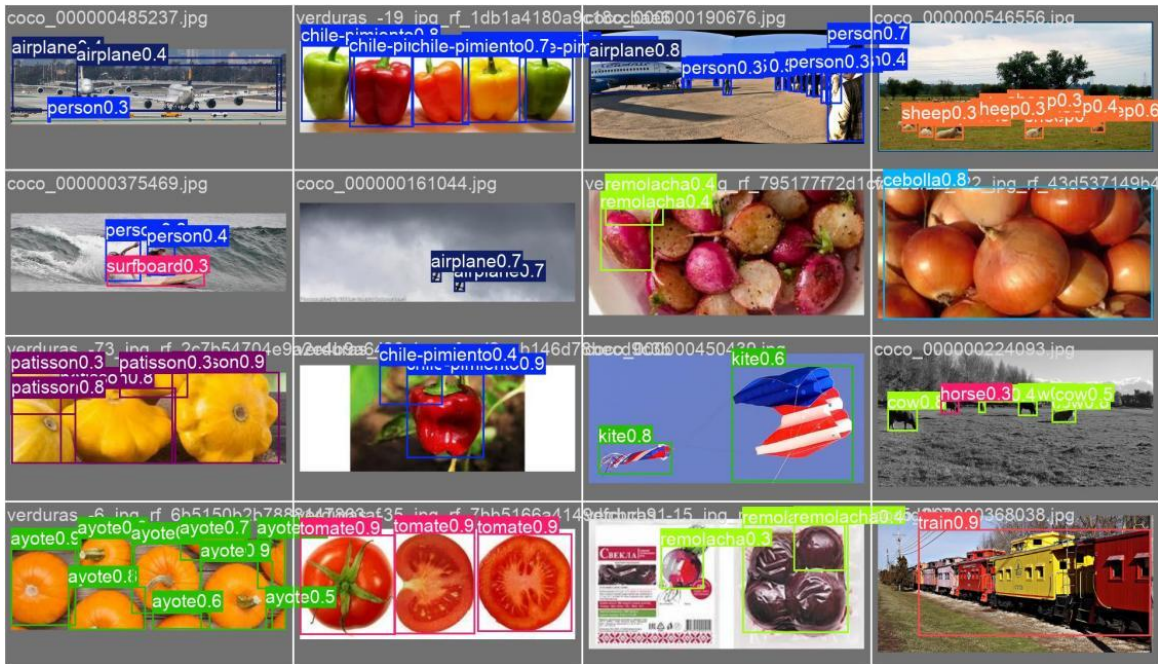
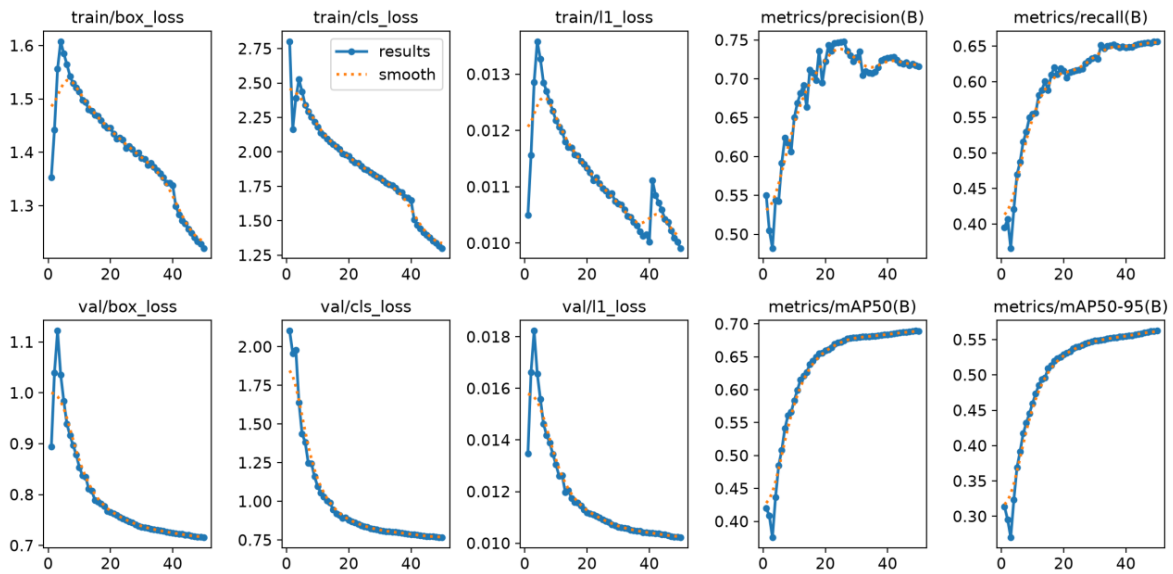
Las métricas se miden con imágenes de validación que el modelo no vio durante el entrenamiento: 4128 en la versión 2, 7873 en la versión 3 y 8365 en la versión 4, porque cada versión suma sus propias imágenes de validación. La precisión indica qué parte de lo detectado era correcto. El recall indica qué parte de los objetos presentes se encontró, y el mAP50 y el mAP50-95 resumen la calidad general de la detección.

Resultados de la versión 2

Fuente	Imágenes de val	Precisión	Recall	mAP50	mAP50-95
Todas, 177 clases	4128	0.713	0.657	0.689	0.562
COCO	607	0.557	0.444	0.471	0.331

Fuente	Imágenes de val	Precisión	Recall	mAP50	mAP50-95
Manos y personas	300	0.903	0.889	0.949	0.816
Productos de Roboflow	892	0.617	0.618	0.637	0.469
MVTec D2S	1747	0.990	0.963	0.974	0.936
Objetos cotidianos	582	0.914	0.852	0.905	0.684

- **Mejores clases según el mAP50:** Audífonos 0.993, Coca-Cola Light 0.980, lechuga 0.979, billetera 0.976, kiwi 0.973, aguacate 0.965, tomate 0.964, Coca-Cola 0.921 y lentes 0.916.
- **Clases débiles:** huevo 0.05, ensalada 0.18 y pollo 0.29. Vienen de un dataset pequeño de platos servidos, no de productos empacados.
- **D2S es optimista:** Su 0.974 se debe a que sus imágenes de validación son del mismo laboratorio, con el mismo fondo y la misma cámara que las de entrenamiento.



Comparación entre la versión 1 y la versión 2

El mAP50 total pasó de 0.491 a 0.689, pero esos valores no son directamente comparables, porque cambiaron las clases y el conjunto de validación. Evaluando ambos modelos sobre las mismas imágenes:

Subconjunto	mAP50 versión 1	mAP50 versión 2
COCO, 607 imágenes	0.434	0.471
Manos, 300 imágenes	0.949	0.949
Productos comunes a ambas, 27 clases en promedio	0.714	0.739

Entrenar con 500 imágenes de COCO por clase en lugar de 200 mejoró los objetos generales sin perder rendimiento en las manos. Entre los productos, Coca-Cola subió de 0.65 a 0.92 y kiwi de 0.71 a 0.97, gracias a las imágenes adicionales de D2S. De las 107 clases presentes en ambas versiones, 62 mejoraron, 26 empeoraron y 19 quedaron igual.

Versión 3: supermercado

La versión 3 se midió con las 3745 imágenes de validación nuevas de supermercado y lácteos, donde alcanzó un mAP50 de 0.707. La tabla compara, sobre esas mismas imágenes, el mAP50 de la versión 2 con el de la versión 3.

Producto	mAP50 versión 2	mAP50 versión 3
Manzana	0.344	0.973
Pan empacado	0.048	0.956
Banano	0.346	0.908
Zanahoria	0.325	0.861
Naranja	0.347	0.817
Chile pimienta	0.175	0.801
Detergente	0.045	0.788
Limón	0.001	0.562
Huevo	0.000	0.370

Además, la versión 3 aprendió 5 clases que la versión 2 no conocía: harina con 0.93, bebida con 0.92, fresa con 0.77, leche con 0.72 y jugo con 0.67.

A cambio, la versión 3 olvidó parte de lo anterior. Sobre las 4128 imágenes de validación de la versión 2, el mAP50 total bajó de 0.689 a 0.668, COCO bajó de 0.471 a 0.457 y los productos de Roboflow de 0.637 a 0.588, mientras que las manos, D2S y los objetos cotidianos cambiaron en 0.014 o menos. 26 clases bajaron más de 0.05, en parte porque 3 nombres pasaron a mezclar objetos distintos: el pan servido en un plato y el pan empacado, los frijoles frescos y los frijoles en lata, y los envases de detergente de dos datasets. Por ejemplo, el pan servido bajó de 0.52 a 0.14 mientras el pan empacado subió a 0.96.

Versión 4: snacks

La versión 4 se midió con las 492 imágenes de validación de los 4 datasets de snacks, donde alcanzó un mAP50 de 0.820. Las bolsas de snacks, las chips amarillas, las chips de sal y Lays se reconocen bien. Takis, Ruffles, Cheetos y Pringles tienen muy pocos ejemplos, por lo que sus métricas se calculan con entre 1 y 10 imágenes de validación y no son confiables.

Clase	Etiquetas en el dataset	Imágenes de validación	mAP50
bolsa-snack	1590	250	0.979
chips-amarillas	113	18	0.943
chips-sal	102	24	0.915
takis *	17	3	0.913
lays	780	117	0.875
ruffles *	56	7	0.796
chicharrón	129	16	0.754
doritos	1259	43	0.740
pringles *	83	1	0.698
snickers	285	23	0.681
cheetos *	64	10	0.640

Para comprobar que no se olvidó lo anterior, se compararon las versiones 3 y 4 sobre las 7873 imágenes de validación previas:

Fuente	mAP50 versión 3	mAP50 versión 4
Total	0.670	0.671
COCO	0.457	0.439
Manos	0.947	0.947
Productos de Roboflow	0.588	0.588
MVTec D2S	0.974	0.972
Objetos cotidianos	0.891	0.903
Supermercado	0.707	0.777

El total quedó igual y el supermercado mejoró de 0.707 a 0.777. La excepción es COCO, que baja un poco en cada versión incremental, porque cada ronda vuelve a ajustar el modelo anterior. 31 clases bajaron más de 0.05, pero 26 subieron más de 0.05 y el cambio promedio por clase fue de +0.001, y la mayoría de las bajas son de clases con pocas imágenes de validación, cuyo valor varía por azar. La baja real más relevante entre los productos es la de bebida, que bajó de 0.445 a 0.380, probablemente por confusión con las bolsas y botellas de los snacks.

Evolución de las 4 versiones

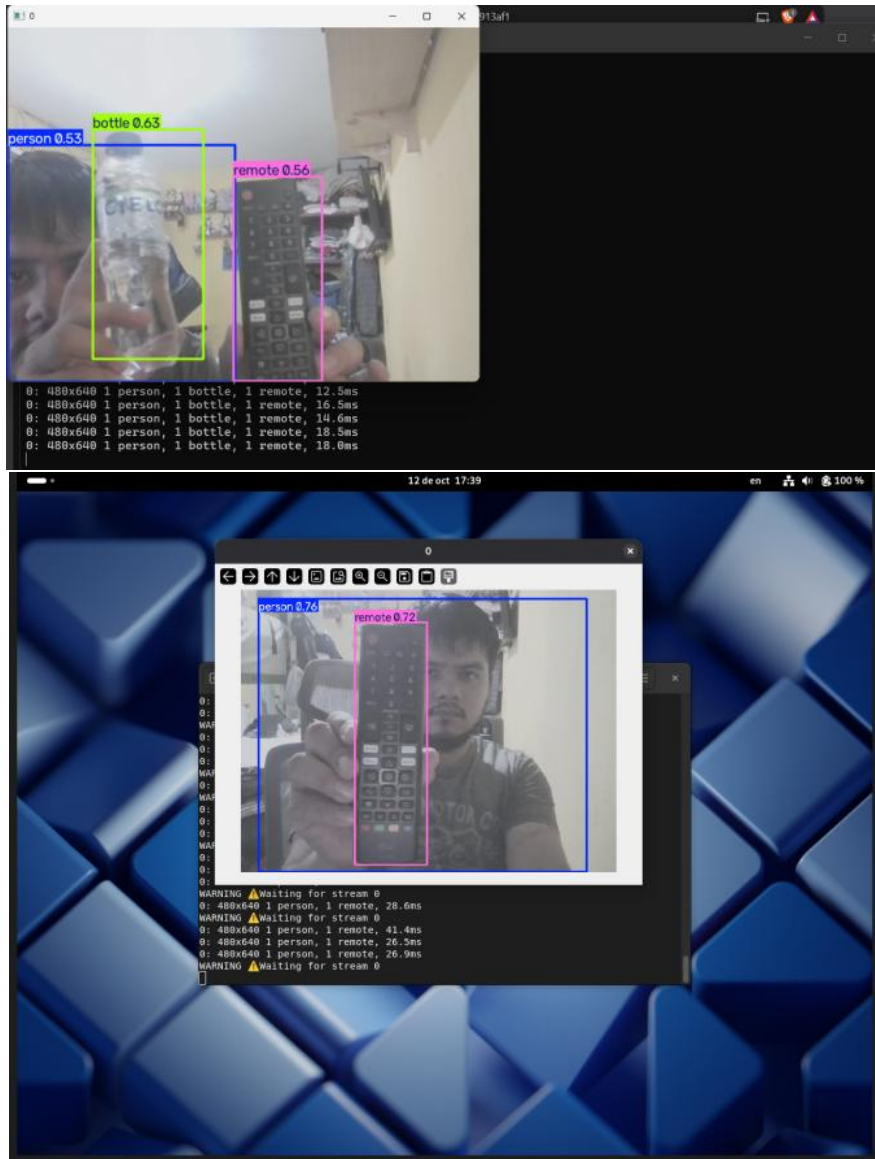
Versión	Clases	Imágenes de entrenamiento	Qué se agregó	Resultado principal
Versión 1	137	17161	COCO con 200 imágenes por clase, manos y primeros productos	mAP50 de 0.491 sobre 1353 imágenes de validación
Versión 2	178	36651	COCO con 500 imágenes por clase, MVTec D2S y objetos cotidianos	mAP50 de 0.689 sobre 4128 imágenes de validación
Versión 3	183	44839	Supermercado y lácteos	Manzana de 0.34 a 0.97 y pan de 0.05 a 0.96; total de 0.668 sobre la validación de la versión 2
Versión 4	194	46618	Snacks	Snacks en 0.820; total de 0.671 sobre la validación de la versión 3 y supermercado de 0.707 a 0.777

El modelo final es la versión 4, que reconoce más clases y es la mejor para detectar productos de supermercado y snacks, sin pérdida en el total. Para consolidarlo, cuando ya no se agreguen más datos conviene un entrenamiento final desde cero con todo el dataset.

Pruebas de inferencia

El modelo se probó con la cámara en Windows y en Debian, y el mismo archivo funcionó en ambos sin ningún cambio. Las capturas y los tiempos de esta sección se tomaron con la versión 2; la versión 4, que es el modelo final, tiene la misma arquitectura, con 121 capas, 2436634 parámetros y 5.6 GFLOPs, frente a 2408932 parámetros y 5.5 GFLOPs del modelo base, y se ejecuta con el mismo comando, cambiando el nombre del archivo:

```
yolo detect predict model=~/modelos/unificado_v4.pt source=0 show=True save=False conf=0.5
```



Entorno	Tiempo por cuadro	Cuadros por segundo
Windows, GPU	12.5 a 18.5 ms	55 a 80
Debian, CPU con 4 núcleos virtuales	26.5 a 50 ms	20 a 35

En Debian solo con el procesador el modelo procesa entre 20 y 35 cuadros por segundo, suficiente para detección en tiempo real. Este es el escenario más parecido al de los nodos del clúster. En las pruebas, algunos objetos fueron más difíciles de detectar con poca luz o con fondos con muchos elementos. Esto se debe a la diferencia entre las fotos de los datasets públicos, tomadas casi siempre con buena luz y fondos limpios, y las condiciones reales de la cámara. Estos tiempos se midieron con la versión 2, y como la versión 4 tiene casi los mismos parámetros y operaciones, se espera una velocidad equivalente.

Despliegue en el clúster

Con el modelo final listo, se armó el sistema completo del Caso 1: una caja de autoservicio de supermercado. La cámara ve los productos, el modelo los reconoce y la caja arma el ticket con los precios y cobra. Todo funciona dentro de la red local, sin internet.

Arquitectura

El sistema usa dos nodos conectados a la misma red local. La Raspberry Pi 400 es la caja: recibe la imagen de la cámara y muestra la página de cobro. La máquina virtual Debian es el nodo de inferencia: tiene el modelo unificado_v4.pt y responde qué productos hay en cada imagen.

Equipo	Función	Dirección IP
Raspberry Pi 400	Caja de autoservicio y página web	192.168.1.60
Máquina virtual Debian en VirtualBox, con 4 núcleos y 10 GB de RAM	Servicio de inferencia con YOLO26 Nano versión 4	192.168.1.78
iPhone con la app DroidCam	Cámara	192.168.1.6

La Raspberry Pi 400 es una Raspberry Pi 4 dentro de un teclado. Tiene el mismo procesador Broadcom BCM2711 de 4 núcleos, 4 GB de RAM y Ethernet Gigabit.

Red

Todos los equipos están en la red 192.168.1.0/24, que es una red privada.

A la máquina virtual se le dio la IP fija 192.168.1.78.

```
sudo ip addr add 192.168.1.78/24 dev enp0s3  
hostname -I
```

El segundo comando muestra las direcciones de la máquina virtual, y ahí debe aparecer 192.168.1.78. Para comprobar la conexión se hizo ping desde la máquina virtual a la Raspberry, y llegaron los 3 paquetes.

Servicio de inferencia en la máquina virtual

El modelo se convirtió en un servicio web con FastAPI, un framework de Python. El programa servicio.py carga el modelo, recibe una imagen y responde la lista de objetos que encontró, con su clase, su confianza y su recuadro. Corre con Uvicorn en el puerto 8001 y usa imágenes de 640 px, el mismo tamaño con el que se entrenó el modelo.

Para arrancarlo se usa el script iniciar-inferencia.sh, que activa el entorno virtual yolo-env, carga el modelo y deja el servicio corriendo en segundo plano:

```
bash ~/iniciar-inferencia.sh
```

Cuando el modelo termina de cargar, el script muestra el nodo, el modelo y el número de clases, que son 194 en la versión 4. Para detener el servicio se usa `kill -f "uvicorn servicio:app"`.

Caja de autoservicio en la Raspberry

La caja es otro programa en Python con FastAPI, llamado app.py, que corre en la Raspberry en el puerto 8000. Toma la imagen de la cámara, la envía a la máquina virtual con la librería httpx y busca cada producto detectado en una tabla de precios. Las ventas se guardan en una base de datos SQLite.

La configuración está en 2 archivos:

- config.json: la cámara y la dirección del nodo de inferencia.
- precios.json: el nombre y el precio en quetzales de cada producto. Tiene 44 productos, como Coca-Cola, Pepsi, Doritos, Takis, leche, aguacate, tomate, teclado y mouse. Las clases que no tienen precio no se cobran, y las personas y las manos se ignoran.

La página web está hecha con HTML, CSS y JavaScript, sin frameworks, para que sea liviana en la Raspberry. Las fuentes se guardaron dentro del proyecto para que la página funcione sin internet.

El proyecto se copió a la Raspberry, en la carpeta /home/grupo5/caja-autoservicio. Las librerías se instalaron una sola vez con el script instalar-pi.sh. Después, con el modelo ya corriendo en la máquina virtual, la caja se arranca desde la terminal de la Raspberry:

```
curl http://192.168.1.78:8001/salud
bash ~/caja-autoservicio/iniciar-caja.sh
chromium --kiosk http://localhost:8000
```

El primer comando comprueba que la Raspberry ve el modelo de la máquina virtual, el segundo arranca la caja en segundo plano y el tercero abre la página en pantalla completa.

Cámara

Primero se probó una cámara USB conectada a la Raspberry. Después se usó la cámara de un iPhone con la app DroidCam, que transmite el video por la red local en la dirección http://192.168.1.6:4747/video. Para volver a la cámara USB basta con poner el valor 0 en la cámara de config.json.

Funcionamiento de la caja

Al presionar Escanear se siguen estos pasos:

1. La caja toma la última imagen de la cámara.
2. La envía por la red a la máquina virtual.
3. El modelo detecta los productos y responde la lista.
4. La caja busca cada producto en precios.json.
5. La página dibuja los recuadros sobre la foto y suma los productos al ticket.

Cada escaneo suma productos al ticket, y el ticket solo se vacía al pagar. Si algo se escanea por error, se puede quitar con el botón de menos o con Vaciar ticket. Si el modelo no reconoce un producto, se puede agregar a mano desde una lista. El pago es simulado: en efectivo la caja calcula el cambio, y con tarjeta muestra un número de autorización sin pedir datos de ninguna tarjeta. Al final se genera un comprobante que se puede imprimir.

En la parte de abajo, la página muestra el estado de cada nodo, que se revisa cada 2 segundos, y qué nodo procesó el último escaneo y cuánto tardó.

Resultados del sistema completo

Medida	Valor
Tiempo de inferencia en la máquina virtual	252 ms
Tiempo total de un escaneo	331 ms
Latencia de red entre la Raspberry y la máquina virtual	5 a 6 ms
Clases del modelo	194
Productos con precio	44

El tiempo total incluye tomar la imagen, enviarla por la red, la inferencia y la respuesta. En la primera prueba el modelo detectó una persona con 87% de confianza, y la caja no la cobró porque las personas están en la lista de clases ignoradas.

Problemas encontrados y soluciones

Problema	Causa	Solución
pip no permitía instalar paquetes en Debian	Debian 12 o superior protege el Python del sistema	Instalar en un entorno virtual llamado venv
Debian no detectaba la cámara	VirtualBox no comparte la webcam por defecto	Activarla en Devices y Webcams, en cada sesión
El entrenamiento en Debian era muy lento	Solo CPU, dentro de una máquina virtual sin GPU	Entrenar el modelo de productos en Windows con la GPU
Nombres de clases inconsistentes entre datasets	Cada dataset usa sus propios nombres como Drinks- Coke	El script unifica nombres sin importar mayúsculas ni signos
Tres datasets de snacks no se podían descargar	No tenían versiones publicadas en Roboflow, por lo que no se pueden bajar con la API	Hacer un Fork a una cuenta propia y generar una versión sin aumento de datos
Algunas clases empeoraron en la versión 3, como pan, frijol y detergente	El mismo nombre agrupa objetos que se ven distinto: pan servido y empacado, frijoles frescos y en lata	Subir el repaso al 80% en la versión 4 y, a futuro, separar los nombres, por ejemplo pan-empacado y frijol-lata
COCO baja un poco en cada versión incremental	Cada ronda vuelve a ajustar el modelo anterior	Un entrenamiento final desde cero con todo el dataset, cuando ya no se agreguen datos
Takis, Ruffles, Cheetos y Pringles se detectan con poca fiabilidad	Los datasets públicos tienen muy pocas fotos de esos productos	Agregar fotos propias; mientras tanto, sus métricas no son confiables

Problema	Causa	Solución
La Raspberry no detectaba la cámara	No había una cámara conectada	Conectar una cámara USB y después usar la del iPhone con DroidCam
La Raspberry no se comunicaba con la máquina virtual	La red de la máquina virtual estaba en modo NAT	Cambiar el adaptador de VirtualBox a modo puente
La máquina virtual no tenía dirección IPv4	NetworkManager no aplicaba la IP fija	Asignarla a mano con ip addr add cada vez que se enciende
La caja se detenía al cerrar la terminal	El programa dependía de la terminal que lo abrió	Arrancarla en segundo plano con nohup desde un script
OpenCV no abría el video del iPhone	No reconoce el formato de video de DroidCam	Leer el video a mano y separar cada imagen JPEG
DroidCam respondía que estaba ocupado	Solo acepta una conexión y se quedó una abierta	Cerrar la app en el iPhone y volver a abrirla
Cada escaneo borraba el ticket	La primera versión reemplazaba el ticket en cada escaneo	Sumar los productos hasta que se paga

Conclusiones

- **El objetivo se cumplió.** Tras 4 versiones se obtuvo un modelo YOLO26 Nano de 194 clases que reconoce productos de consumo, incluidos los de supermercado y los snacks, además de objetos cotidianos, personas y manos, con un mAP50 de 0.671 sobre las 7873 imágenes de validación de la versión 3 y de 0.820 sobre las de snacks.
- **El modelo es viable para el borde.** Pesa menos de 6 MB y procesa entre 20 y 35 cuadros por segundo solo con el procesador, sin GPU ni conexión a internet, como lo requiere el Caso 1.
- **Separar entrenamiento e inferencia fue clave.** Entrenar en Windows con la GPU tomó 6 horas en lugar de días, sin contradecir el enunciado, que solo restringe la etapa de inferencia.
- **El entrenamiento incremental funciona, con límites.** Agregar supermercado en la versión 3 y snacks en la versión 4 tomó 1 hora 44 minutos y 2 horas 27 minutos, en lugar de las 6 horas de un entrenamiento completo, y mejoró mucho los productos nuevos sin perder lo anterior en el total. A cambio, COCO baja un poco en cada ronda, de 0.471 a 0.457 y después a 0.439.
- **La calidad de los datos pesa más que la cantidad.** Las clases genéricas o ambiguas dieron los peores resultados, y las fotos de laboratorio inflan las métricas frente a las condiciones reales.
- **Importancia de la conectividad de red.** La conexión Ethernet mediante el switch permite la comunicación entre la Raspberry Pi 400 y la computadora que ejecuta la máquina virtual, el router doméstico proporciona acceso a Internet cuando es necesario, mientras que el intercambio de imágenes y resultados puede realizarse dentro de la red local sin requerir servicios externos de inteligencia artificial.
- **Desarrollo de una aplicación de supermercado inteligente.** Se implementó una interfaz web que permite escanear productos, actualizar el ticket de compra, calcular el total, gestionar artículos y generar comprobantes. También se incorporó un flujo de pago simulado, con cálculo de cambio para

efectivo y una autorización ficticia para tarjeta, demostrando la integración entre visión artificial y lógica comercial.

- **Importancia de la calidad del conjunto de datos.** Las evaluaciones demostraron que la precisión depende de la variedad y representatividad de las imágenes utilizadas durante el entrenamiento. Las clases visualmente ambiguas y las diferencias entre las condiciones controladas y las condiciones reales pueden afectar el reconocimiento, por lo que es necesario continuar mejorando el dataset y evaluar el modelo en escenarios diversos.
- **Eficacia y limitaciones del entrenamiento incremental.** La incorporación progresiva de nuevas categorías permitió ampliar el modelo sin repetir completamente el entrenamiento inicial. Sin embargo, también se observó una disminución gradual del mAP50 en las clases de COCO, de 0.471 a 0.457 y posteriormente a 0.439, lo que evidencia la necesidad de evaluar el impacto del aprendizaje incremental sobre las clases previamente entrenadas.
- **Potencial de ampliación del sistema.** La arquitectura permite incorporar nuevos productos, actualizar los precios y mejorar el modelo sin tener que reconstruir completamente la aplicación. Como trabajo futuro, se podrían implementar mecanismos de supervisión automática de los servicios, recuperación ante fallos y validación adicional de las detecciones para reducir errores durante el registro de compras.

Referencias

- Ultralytics. Documentación de Ultralytics YOLO. <https://docs.ultralytics.com/>
- Ultralytics. Hand Keypoints Dataset. <https://docs.ultralytics.com/datasets/pose/hand-keypoints>
- Lin, T.-Y. et al. COCO: Common Objects in Context. <https://cocodataset.org>
- MVTec. MVTec D2S: Densely Segmented Supermarket Dataset. Licencia CC BY-NC-SA 4.0. <https://www.mvtec.com/company/research/datasets/mvtec-d2s>
- Datasets de Roboflow Universe con licencia CC BY 4.0:
 - soft drinks 2: <https://universe.roboflow.com/sukru-can-ercoban/soft-drinks-2>
 - multi food detection model: <https://universe.roboflow.com/asdas-3oje1/multi-food-detection-model>
 - vgvvggvvgv: <https://universe.roboflow.com/abbas-ugfrv/vgvvggvvgv>
 - Robotica_2024: https://universe.roboflow.com/objectrecognitionbahiart/robotica_2024
 - Vegetables: <https://universe.roboflow.com/vegetable-object-detection/vegetables-kacga-lvr1d>
 - headphones: <https://universe.roboflow.com/aihat/headphones-moskk>
 - Box Detection: <https://universe.roboflow.com/emperium/box-detection-sz4gh>

- glasses_detector_robo: https://universe.roboflow.com/waimea/glasses_detector_robo
 - Object Detection HW: <https://universe.roboflow.com/robotics-lab-3-nryht/object-detection-hw>
 - ieee: <https://universe.roboflow.com/bdu-b0bls/ieee-umc5f>
 - YOLO-based Fine-Tuning: <https://universe.roboflow.com/hilcps/yolo-based-fine-tuning>
- Datasets de Roboflow Universe agregados en las versiones 3 y 4, con licencia CC BY 4.0:
 - grocery_picking_2: https://universe.roboflow.com/openhousews/grocery_picking_2
 - Retail object detection: <https://universe.roboflow.com/polyu-9ziss/retail-object-detection-bqtzf>
 - chips: <https://universe.roboflow.com/meanzaza/chips-slo4r>
 - cpvis: <https://universe.roboflow.com/compvis-4pxvs/cpvis>
 - x2: <https://universe.roboflow.com/ai-for-internship/x2-e78hc>
 - lays: <https://universe.roboflow.com/mehar/lays-txw7q>