Trabajo Practico

Visión por Computadora

Autor: Iair Borgo Elgart

Universidad Nacional de Rosario, TUIA 2025

Fecha de entrega: 29/6/25

Índice

1. Resumen:

En este trabajo práctico final, desarrollamos un **sistema completo de visión por computadora** diseñado para la **identificación automática de razas de perros en imágenes**. El *pipeline* abarca desde la creación de un robusto sistema de búsqueda por similitud basado en *embeddings*, hasta la detección y clasificación de perros en escenas complejas del mundo real. A lo largo del desarrollo, integramos diversas técnicas avanzadas de **Deep Learning**, incluyendo la **transferencia de aprendizaje** con modelos preentrenados de vanguardia como **ResNet**, el **entrenamiento personalizado de clasificadores convolucionales** desde cero, y la **detección de objetos** utilizando el modelo **YOLOv12**.

El proceso de desarrollo se estructuró en cuatro etapas incrementales, lo que nos permitió validar cada componente de forma aislada y progresiva. Las primeras tres etapas fueron implementadas y exploradas extensivamente en un entorno de *notebook* interactivo:

1. **Buscador de imágenes similares en *embedding* preentrenado con una interfaz en Gradio**: Esta etapa inicial se centró en la creación de un sistema de recuperación de imágenes basado en la similitud semántica. Utilizamos *embeddings* generados por un modelo preentrenado (ResNet50) para indexar imágenes y permitir búsquedas eficientes a través de una interfaz de usuario intuitiva desarrollada con Gradio.

2. **Entrenamiento y comparación de modelos de clasificación**: Aquí, exploramos dos enfoques principales para la clasificación de razas de perros. Por un lado, aplicamos **transferencia de aprendizaje** a un modelo ResNet18, ajustando las capas finales para nuestra tarea específica. Por otro lado, diseñamos y entrenamos un **modelo convolucional personalizado** desde cero. Ambos modelos fueron comparados en términos de rendimiento y se les asoció una interfaz Gradio para demostraciones interactivas.

3. **Un *pipeline* de detección de perros con YOLO y clasificación para imágenes del mundo real**: Esta etapa crucial se centró en la creación de un *pipeline* integral capaz de operar en imágenes no curadas. Primero, utilizamos YOLO para detectar y localizar perros dentro de escenas complejas, extrayendo los *bounding boxes* correspondientes. Luego, las regiones detectadas fueron clasificadas por raza utilizando los modelos desarrollados en la etapa anterior.

4. **Modelo YOLO optimizado en formato ONNX y llevado a un archivo .py**: Para optimizar el rendimiento y facilitar la implementación en entornos de producción, el modelo YOLO fue exportado al formato ONNX. Esto permitió una inferencia

significativamente más rápida. Posteriormente, todo el código fue refactorizado y estructurado en un proyecto Python coherente, con definiciones de clases en módulos separados y *scripts* auxiliares.

Finalmente, todo el código fue estructurado en un **proyecto utilizable**, creando definiciones de clases en módulos separados y desarrollando *scripts* auxiliares para facilitar la automatización, incluyendo un **script de anotación automática** de imágenes.

2. Introducción

La **clasificación de razas de perros a partir de imágenes** representa un desafío paradigmático en el campo de la visión por computadora. La complejidad radica en la **gran variedad de razas**, muchas de las cuales comparten **similitudes morfológicas sutiles**, así como en las diversas **condiciones de captura de las imágenes**, que pueden incluir variaciones en la iluminación, el ángulo de la toma, o la presencia de fondos complejos que añaden ruido y dificultan la detección. Este trabajo tiene como objetivo principal el desarrollo de un **sistema integral** capaz de identificar razas de perros en imágenes, tanto simples como complejas, mediante un **enfoque progresivo de visión por computadora**.

Nuestra propuesta metodológica parte de un **sistema de recuperación por similitud** utilizando *embeddings* generados con un modelo preentrenado (ResNet50), los cuales son almacenados eficientemente en una base de datos **ChromaDB**. Este enfoque inicial nos permitió explorar la riqueza semántica de las imágenes. Progresamos hacia el entrenamiento de modelos clasificadores, incluyendo un modelo **fine-tuned** (ResNet18) y un modelo **custom** al cual "desconectamos la cabeza" (la capa clasificadora final) para obtener *embeddings* genéricos, lo que nos permitió crear nuevas colecciones para comparaciones y análisis de similitud más avanzados. El *pipeline* culmina en un sistema capaz de **detectar perros con YOLO en escenas del mundo real**, para luego **predecir su raza** y generar **anotaciones automáticas**. Este enfoque multifacético nos permitió abordar tareas tanto de **clasificación supervisada** como de **detección multiobjeto**, integrando diversas herramientas y metodologías fundamentales exploradas a lo largo de la materia.

La estructura del informe se adhiere a la secuencia cronológica en la que se abordaron las consignas del proyecto. Comenzamos con un **análisis exploratorio detallado en un entorno de *notebook***, donde se sentaron las bases para la definición de los *datasets* y la selección inicial de modelos. Para la fase final de implementación, todo el código fue refactorizado y trasladado a un **proyecto Python modular y escalable**, apto para un desarrollo y despliegue más profundo.

3. Metodología

Durante las primeras etapas del desarrollo, utilizamos un entorno de trabajo en **Google Colab** para la experimentación rápida y eficiente con los diversos componentes del sistema. El *notebook* exploratorio, disponible en el siguiente enlace [Notebook Colab], documenta en detalle el flujo de pruebas iniciales y la creación de objetos fundamentales que sirvieron como base para el código final del proyecto.

En esta fase inicial:

Descargamos el *dataset* denominado **"70-dog-breeds-image-data-set" desde Kaggle**.

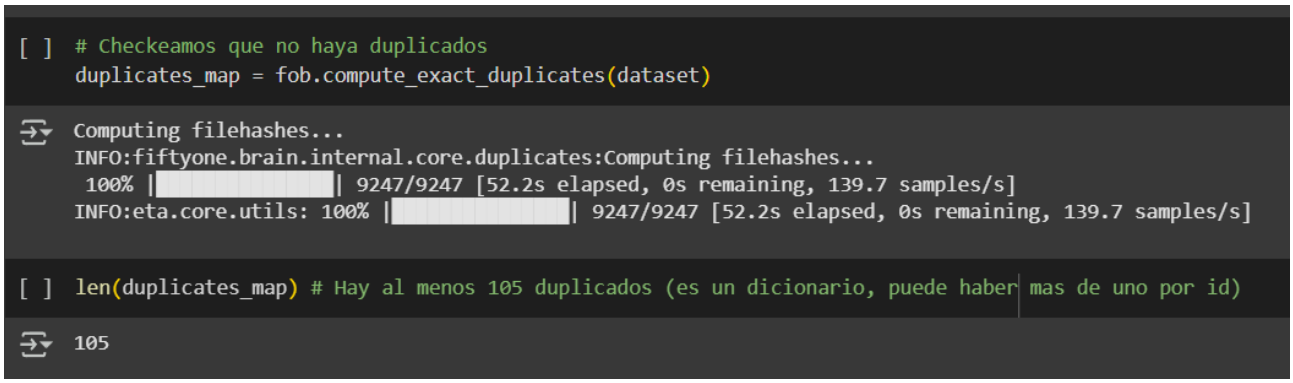
	filepaths	labels	data set
0	train/Afghan/001.jpg	Afghan	train
1	train/Afghan/002.jpg	Afghan	train
2	train/Afghan/003.jpg	Afghan	train
3	train/Afghan/004.jpg	Afghan	train
4	train/Afghan/005.jpg	Afghan	train
...	...	...	...
9341	valid/Yorkie/06.jpg	Yorkie	valid
9342	valid/Yorkie/07.jpg	Yorkie	valid
9343	valid/Yorkie/08.jpg	Yorkie	valid
9344	valid/Yorkie/09.jpg	Yorkie	valid
9345	valid/Yorkie/10.jpg	Yorkie	valid

Este *dataset* se presenta como un DataFrame que nos proporciona la ruta de la imagen original (filepaths), la raza del perro (labels), y la división del *dataset* a la que pertenece (data set). Inicialmente, el *dataset* comprendía un total de **9346 imágenes**.

Se identificaron 71 etiquetas de razas diferentes. Sin embargo, tras un análisis exploratorio, decidimos **eliminar una de las clases que presentaba un número extremadamente bajo de observaciones (solo 10 imágenes)**. La siguiente clase con menor representatividad contaba con 80 imágenes, lo que evidenciaba una disparidad significativa y justificaba la eliminación para evitar sesgos en el entrenamiento.

Posteriormente, aprovechando **FiftyOne**, una herramienta ampliamente utilizada en el curso, creamos un **Dataset propio** dentro de su paquete. FiftyOne demostró ser invaluable, facilitando tareas cruciales como la **generación de *embeddings***, la **detección de imágenes duplicadas** y la **migración a otros formatos** de *dataset*.

Mediante el método .compute_exact_duplicates de FiftyOne, detectamos la existencia de al menos **105 imágenes exactamente iguales (por *hash* de contenido)**. Estas imágenes fueron eliminadas del *dataset* para garantizar la consistencia y evitar el sobreajuste.

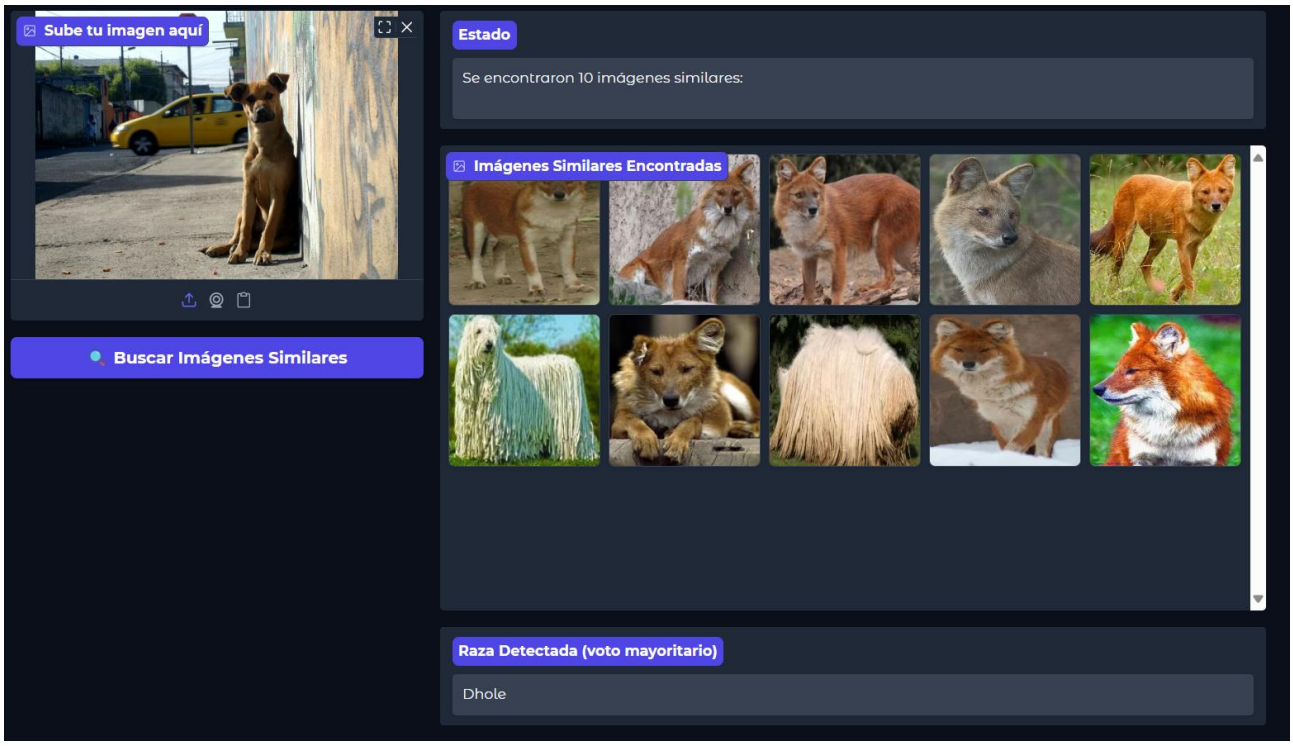


A través del método .compute_leaky_splits, identificamos un número considerable de imágenes que, aunque no eran idénticas, presentaban **ligeras transformaciones** (rotaciones, cambios de escala, etc.). Si bien esta es una práctica común en el *data augmentation*, su presencia en un *dataset* crudo resultó peculiar. A pesar de esto, se decidió **mantener estas imágenes** para evitar una pérdida excesiva de datos.

El *dataset* final, después de la limpieza y curación, quedó distribuido de la siguiente manera: **7811 datos para entrenamiento, 653 para testeo y 664 para validación**.

Utilizando nuevamente las potentes herramientas de FiftyOne, procedimos a **crear los *embeddings* de las imágenes utilizando el modelo ResNet50 preentrenado por defecto**. Estos *embeddings* de alta dimensionalidad fueron posteriormente almacenados de manera eficiente en una base de datos **ChromaDB**, que nos permitió realizar búsquedas de similitud vectorial de forma rápida.

Una vez que tuvimos todos los componentes necesarios, procedimos a desarrollar una **aplicación interactiva en Gradio** para demostrar y probar el sistema de búsqueda por similitud de los *embeddings*.



Aquí se evidenciaron algunas de las limitaciones del enfoque inicial. Las imágenes utilizadas para el entrenamiento, provenientes del *dataset*, tenían todas una **dimensión estandarizada de 224x224 píxeles** y presentaban a los perros perfectamente recortados y centrados. Al proporcionarle al sistema una imagen en la que el perro no estaba centrado o estaba acompañado de un fondo complejo, el modelo de *embedding* procesaba la imagen completa, lo que resultaba en **problemas de detección de similitud** ya que la información del fondo podía dominar sobre la del perro.

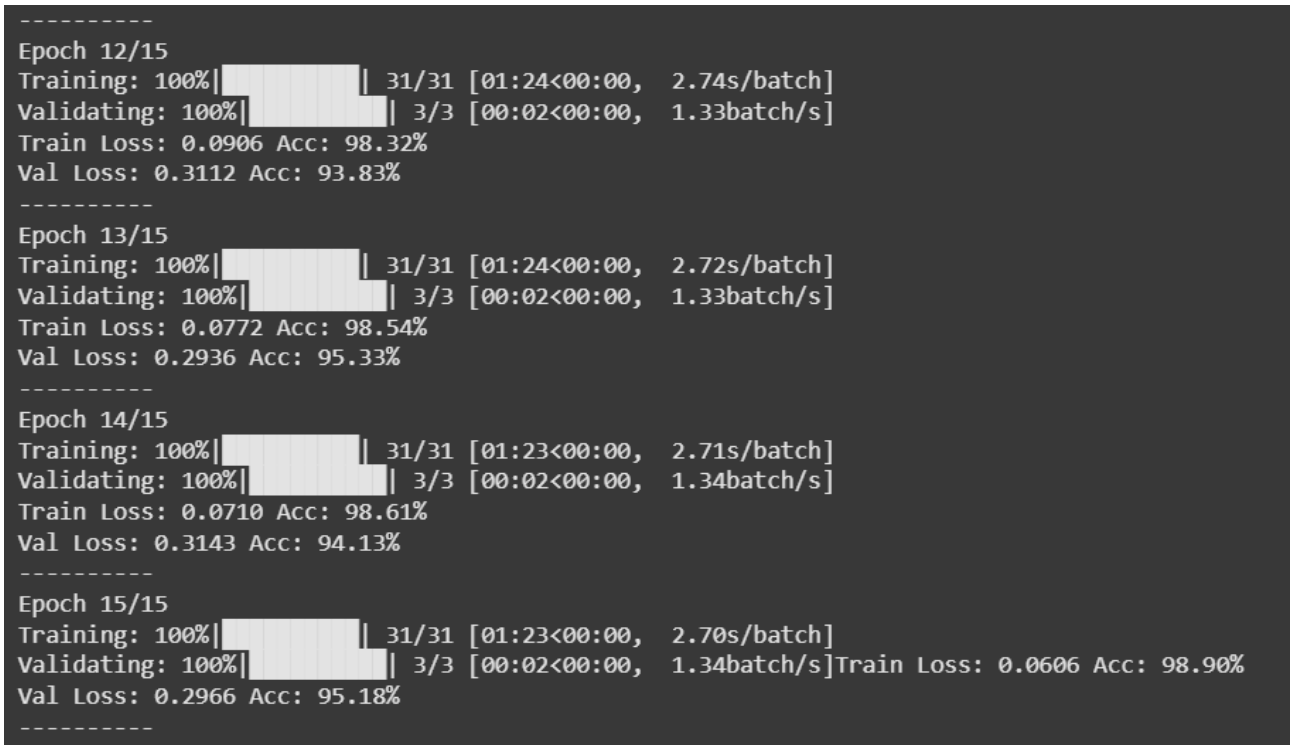
A pesar de esta limitación con imágenes del mundo real, al proporcionarle imágenes del propio set de validación o testeo, el sistema lograba **distinguir la raza sin ningún problema**. Esto se corroboró al calcular la métrica **NDCG@10 (Normalized Discounted Cumulative Gain at 10)** sobre el *dataset* de validación completo, obteniendo un **impresionante puntaje de 0.9635**. Esto indica que el sistema es altamente efectivo para recuperar imágenes relevantes dentro de su dominio de entrenamiento.

Para la siguiente etapa, surgieron desafíos al intentar cargar el *dataset* de FiftyOne en el *framework* **PyTorch**. Tras consultar la documentación, encontramos la solución en el módulo utils.torch de FiftyOne. Utilizando la clase **TorchImageClassificationDataset**, logramos integrar sin problemas ambos paquetes, permitiendo que PyTorch accediera directamente a los datos gestionados por FiftyOne.

Al *dataset* de entrenamiento le aplicamos técnicas de **aumento de datos (data augmentation)** para mejorar la robustez y la capacidad de generalización del modelo. Estas

transformaciones incluyeron giros horizontales y verticales aleatorios de la imagen, rotaciones de hasta 10 grados y leves modificaciones en los colores (jitters). Este *data augmentation* es crucial para simular variaciones en las condiciones de captura y reducir el sobreajuste, lo que a su vez mejora las predicciones en los conjuntos de testeo y validación.

Para el modelo de **transferencia de aprendizaje**, utilizamos la arquitectura **ResNet18**. Como **capa clasificadora (conocida como *head*)**, implementamos una simple capa lineal con una activación que mapeaba el *embedding* resultante a las **70 clases** de razas de perros que estábamos prediciendo. Para optimizar aún más la predicción y permitir que el modelo aprendiera características más finas relevantes para nuestro *dataset*, **descongelamos la capa final convolucional (conocida como *neck*)** de ResNet18. A esta capa descongelada se le aplicó un ***learning rate* más bajo** que al *head*, dado que ya estaba preinicializada con pesos útiles de la tarea original de ImageNet, mientras que el *head* se inicializó desde cero.



En la última época de entrenamiento, este modelo de transferencia de aprendizaje alcanzó un **95.18% de *accuracy*** en el conjunto de validación, lo que demuestra la efectividad de la transferencia de conocimiento de modelos preentrenados.

Como experimento adicional y para comparar enfoques, decidimos entrenar un **modelo convolucional personalizado (custom model)** desde cero. Este modelo constaba de **seis capas convolucionales**, donde cada bloque repetía la secuencia de **convolución, normalización por lotes (*Batch Normalization*), activación ReLU y *MaxPooling***.

Comenzamos con 8 filtros en la primera capa y duplicamos este número en cada capa subsecuente. Para la clasificación final, se aplicó una operación de *flattening* seguida de una **capa oculta de 512 neuronas** con activación ReLU, y finalmente una **capa lineal de salida con las 70 clases** a predecir.

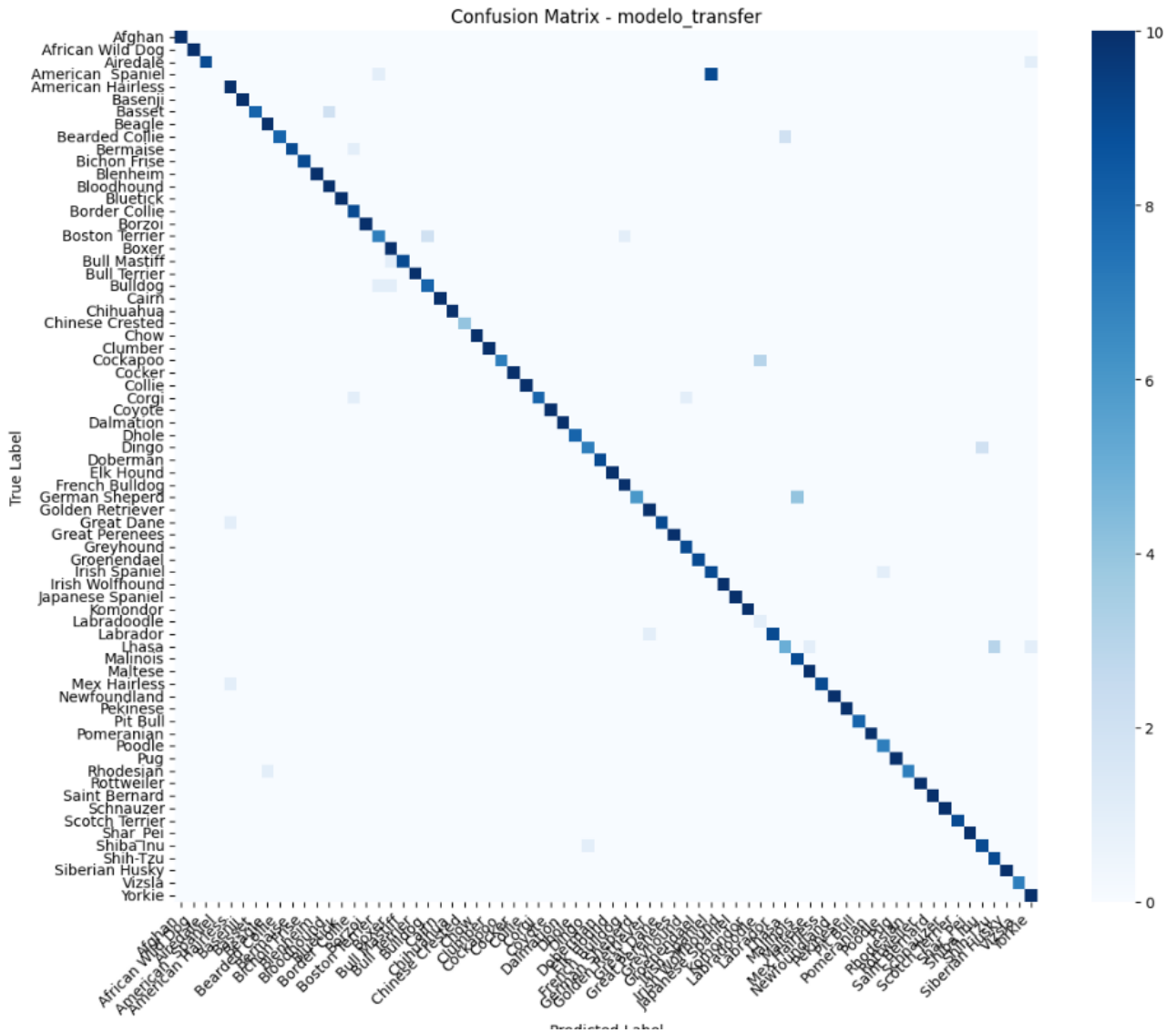
```
----------
Epoch 27/30
Training: 100%|██████████| 31/31 [01:21<00:00,  2.62s/batch]
Validating: 100%|██████████| 3/3 [00:01<00:00,  1.52batch/s]
Train Loss: 1.3724 Acc: 59.52%
Val Loss: 1.4327 Acc: 60.54%
----------
Epoch 28/30
Training: 100%|██████████| 31/31 [01:20<00:00,  2.61s/batch]
Validating: 100%|██████████| 3/3 [00:01<00:00,  1.52batch/s]
Train Loss: 1.3563 Acc: 60.27%
Val Loss: 1.3300 Acc: 61.90%
----------
Epoch 29/30
Training: 100%|██████████| 31/31 [01:21<00:00,  2.61s/batch]
Validating: 100%|██████████| 3/3 [00:01<00:00,  1.59batch/s]
Train Loss: 1.3297 Acc: 61.48%
Val Loss: 1.5539 Acc: 58.89%
----------
Epoch 30/30
Training: 100%|██████████| 31/31 [01:20<00:00,  2.61s/batch]
Validating: 100%|██████████| 3/3 [00:01<00:00,  1.53batch/s]Train Loss: 1.2753 Acc: 62.19%
Val Loss: 1.4622 Acc: 61.75%
----------
```

Este modelo personalizado, si bien tuvo un desempeño aceptable, **no logró igualar la *performance* del modelo basado en transferencia de aprendizaje**. Aunque es posible que manipulando la arquitectura (añadiendo más capas, cambiando el número de filtros, etc.) se pudiera haber logrado un mejor resultado, por restricciones de tiempo y poder computacional, decidimos seleccionar al modelo de ResNet18 con transferencia de aprendizaje como nuestro modelo preferido debido a su superior rendimiento.

```
Evaluating modelo_custom...
Results for modelo_custom:
  Exactitud (Accuracy): 0.6175
  Precisión (Precision): 0.6764
  Sensibilidad (Recall): 0.6175
  Especificidad (Specificity): 0.9945
  F1-Score: 0.6122

Evaluating modelo_transfer...
Results for modelo_transfer:
  Exactitud (Accuracy): 0.9337
  Precisión (Precision): 0.9330
  Sensibilidad (Recall): 0.9337
  Especificidad (Specificity): 0.9990
  F1-Score: 0.9284
```

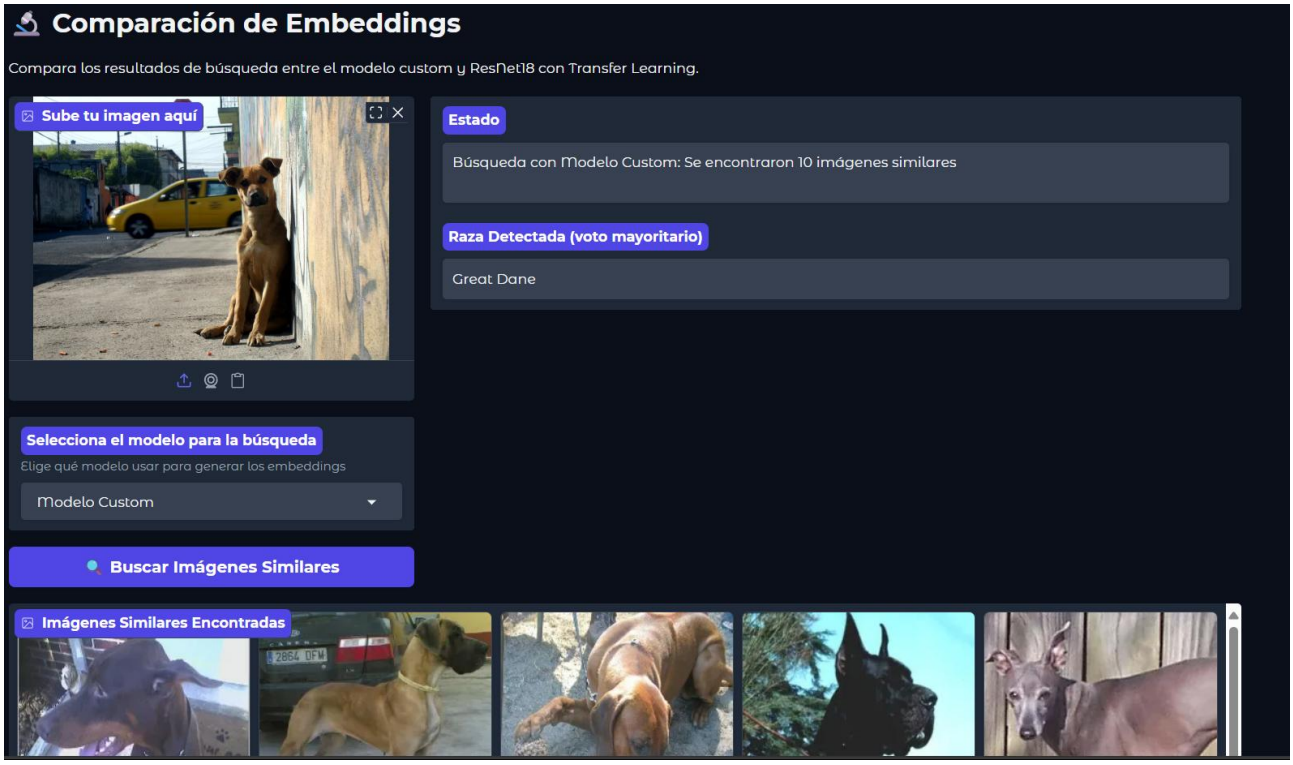
La **especificidad del modelo *custom***, como métrica aislada, puede resultar engañosa en un problema multiclase. Al analizar todas las métricas relevantes, el **modelo de clasificación basado en transferencia de aprendizaje (ResNet18 fine-tuned) superó ampliamente al modelo *custom*** en términos de precisión general y capacidad de generalización.



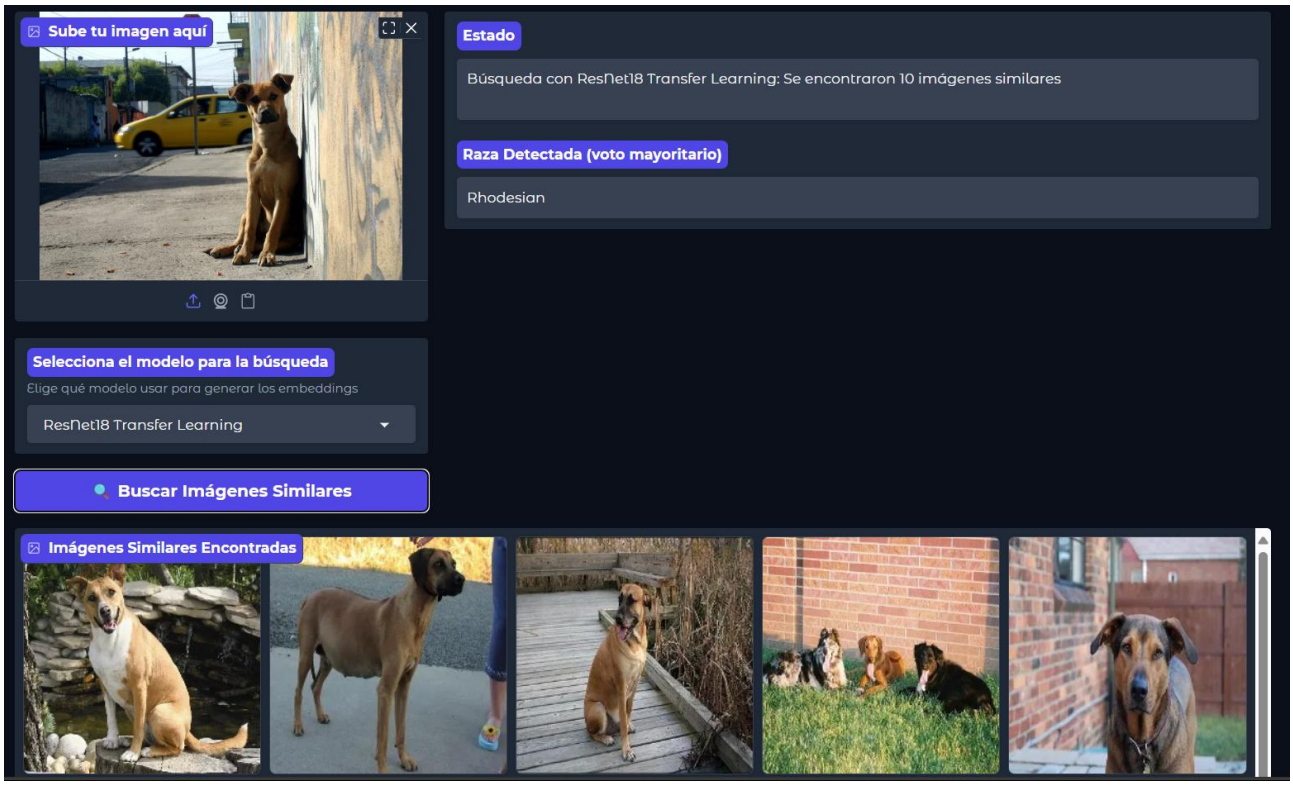
En la **matriz de confusión** generada, el punto más prominente y de mayor confusión se asoció con la etiqueta que habíamos eliminado previamente del *dataset* (pero que por alguna razón el LabelEncoder mantuvo). Excluyendo este artefacto, el modelo no parecía tener confusiones significativas entre las razas restantes, lo que sugiere una buena capacidad de discriminación.

Para poder avanzar con la etapa 3 del proyecto, fue necesario **desconectar la capa clasificadora (*head*) de ambos modelos** (razón por la cual la capa convolucional en el modelo de transferencia fue descongelada). Esto nos permitió obtener los **embeddings de**

las imágenes a partir de las características aprendidas por las capas profundas de los modelos. Una vez obtenidos, estos *embeddings* fueron añadidos a **colecciones separadas en ChromaDB** para cada modelo. Con estas nuevas colecciones, desarrollamos una **nueva versión de la aplicación Gradio** que permitía comparar los resultados de búsqueda de similitud entre los *embeddings* generados por el modelo fine-tuned y el modelo custom.



En esta demostración utilizando una foto de un perro callejero, el **modelo *custom*** infiere que se trata de un Gran Danés, lo cual visualmente no parece ser correcto. Al comparar con el otro modelo:



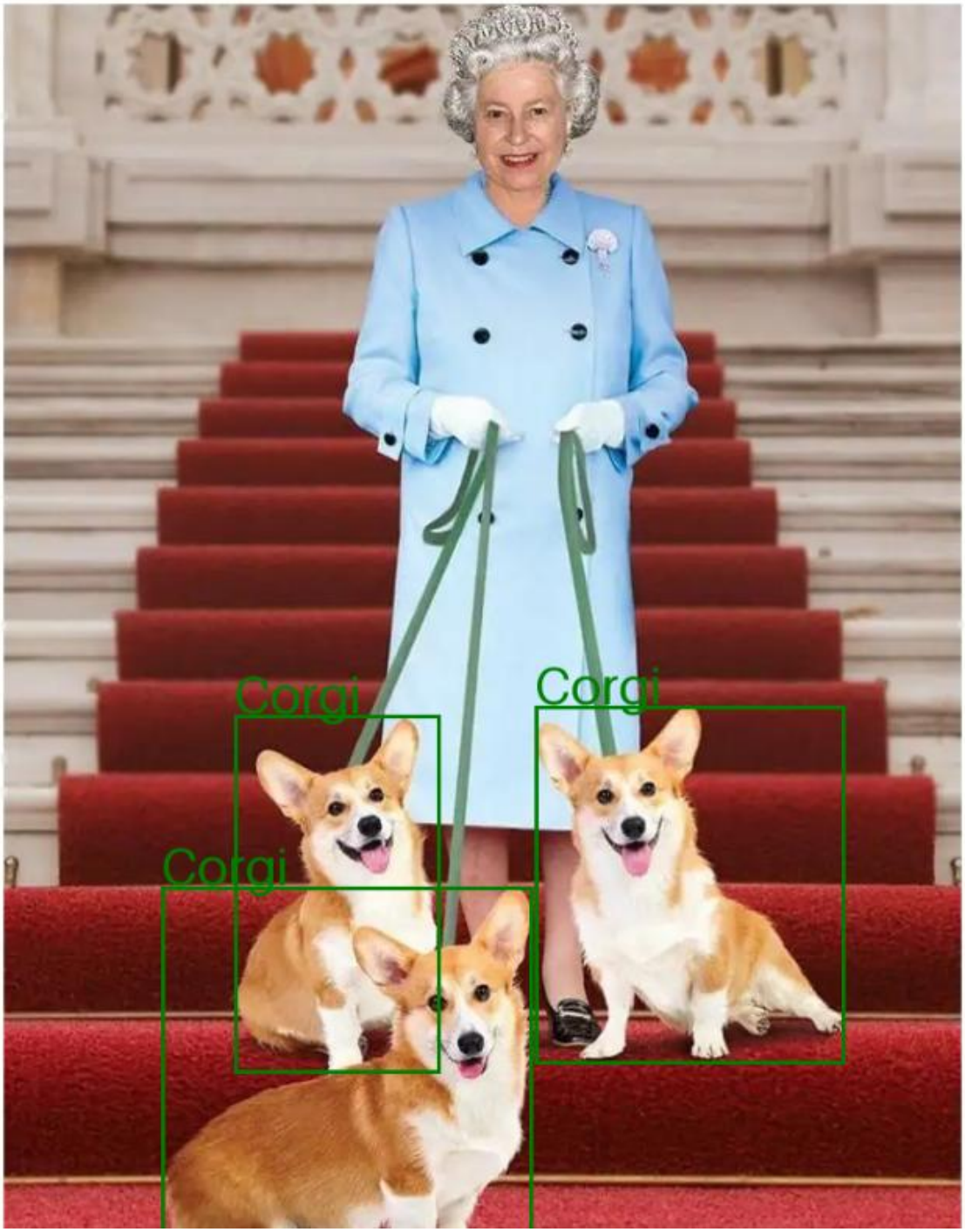
El **modelo basado en transferencia de aprendizaje** lo clasifica como un Rodesiano, lo cual podría ser una predicción más plausible considerando las características del perro callejero, aunque sigue siendo una inferencia en un contexto de incertidumbre.

Es importante destacar una mejora significativa en el tiempo de inferencia: la búsqueda de similitud con esta nueva aplicación basada en los modelos fine-tuned y custom tarda **menos de 2 segundos** en un dispositivo equipado con una CPU Ryzen 5700u, mientras que la aplicación anterior (basada solo en ResNet50 para *embeddings*) rondaba los 8 segundos. Esta diferencia de velocidad podría atribuirse a la optimización de las operaciones en PyTorch o a la naturaleza de los *embeddings* generados por los modelos entrenados.

Para la **etapa 3** de este proyecto, optamos por utilizar **YOLOv12 nano**, siendo la versión más reciente disponible en Ultralytics al día de la entrega. Una investigación previa nos indicó que esta versión no solo es **más precisa** que sus predecesoras, sino que también ofrece un **menor tiempo de inferencia** para tamaños de modelo similares, lo que lo convierte en una opción ideal para aplicaciones en tiempo real.

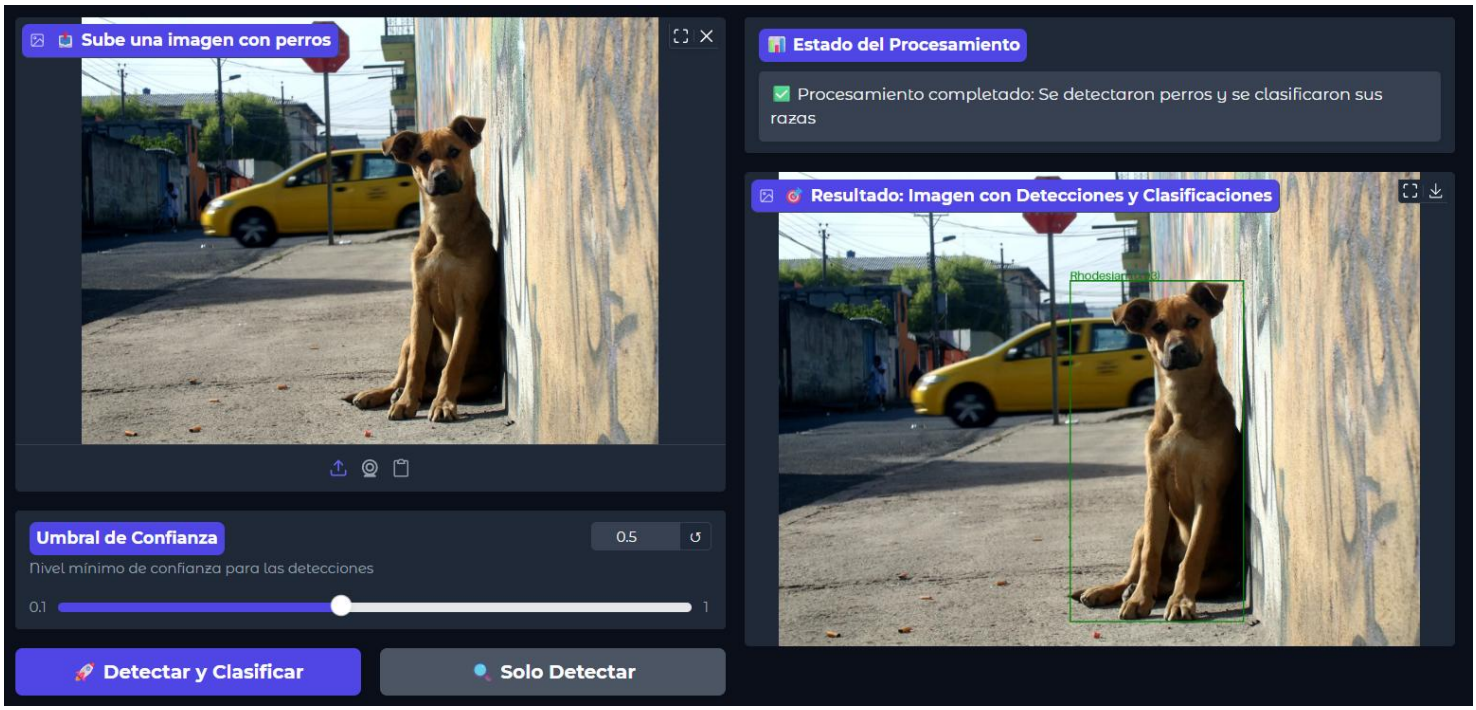
El *pipeline* para la detección y clasificación en imágenes del mundo real funciona de la siguiente manera: primero, la imagen se procesa con YOLO, que se encarga de detectar los **objetos de la clase "perro" (clase 16)** y generar sus respectivos *bounding boxes*. Luego, la región de la imagen dentro de cada *bounding box* es **recortada** y se le aplica una

transformación para ajustar su tamaño al esperado por nuestro modelo de clasificación de razas. Observamos que al realizar este recorte y redimensionamiento, la predicción de la raza mejoraba significativamente. Aunque se podría haber utilizado *pooling* adaptativo para aceptar tamaños variables de recortes, esto presentaba el problema de que todas nuestras imágenes de entrenamiento estaban estandarizadas a un tamaño fijo, lo que, en la práctica, **empeoraba el rendimiento** del clasificador al introducir variabilidad no vista durante el entrenamiento.



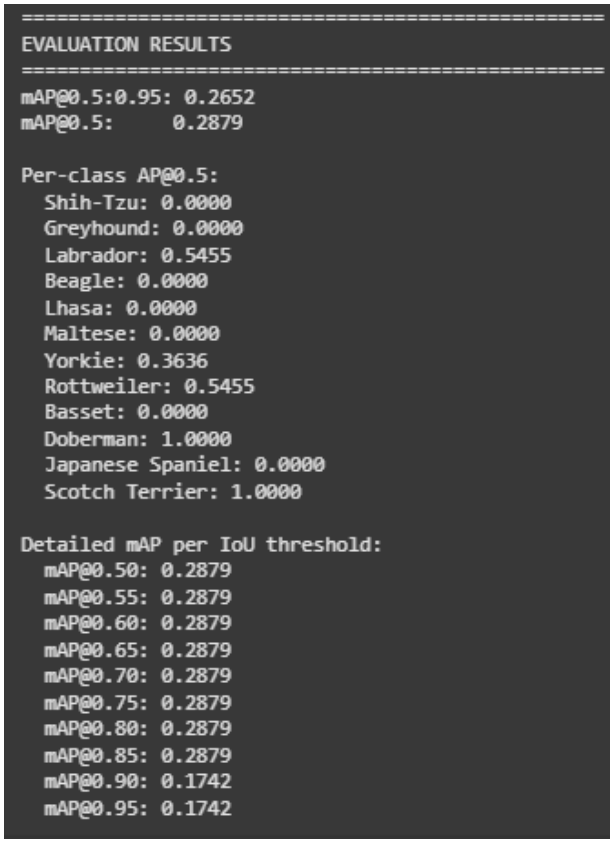
Las salidas de este modelo (YOLO + clasificador) son las mejores obtenidas hasta el momento. **Recortar la imagen al *bounding box* del perro simula el tipo de imágenes con las que se entrenó el modelo de clasificación**, mejorando así drásticamente la

predicción de la raza. Al aplicar este *pipeline* a la misma imagen del perro callejero utilizada previamente:

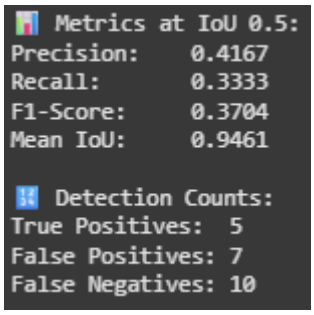


El sistema detecta una vez más un Rodesiano, esta vez con una **confianza de 0.93**. Esto demuestra la efectividad de la detección y el recorte para aislar el objeto de interés. Podemos ajustar el umbral de confianza para eliminar detecciones dudosas si es necesario, aunque en este caso la confianza es alta.

Para evaluar el rendimiento del *pipeline* completo en un entorno más "realista", recopilamos **10 imágenes de Reddit que contenían un total de 8 clases de perros diferentes**. Calculamos la métrica **mAP (mean Average Precision) 0.5:0.95** sobre este pequeño conjunto de prueba:



Obtuvimos un resultado de **0.2652**. Este valor puede ser engañoso. Es importante destacar que estas imágenes son **muy diferentes a las utilizadas durante el entrenamiento** del modelo, presentando variaciones significativas en composición, iluminación y contexto. Además, el **tamaño extremadamente pequeño del conjunto de prueba** (solo 10 imágenes) introduce una variabilidad muy alta en la métrica, haciendo que el resultado sea menos representativo de la *performance* general. Idealmente, necesitaríamos al menos 20 perros de cada una de las razas para obtener una evaluación más robusta, pero por limitaciones de tiempo, nos mantuvimos con este conjunto. Este número, de manera aproximada, indica que, **en promedio, nuestro *pipeline* completo acierta la raza correcta y su localización en un 26% de las veces** en este entorno "real".



Por otro lado, tanto la **precisión como el *recall* resultaron ser bajos**. Esto sugiere que el modelo **omite la detección de muchos perros** que debería encontrar (*recall* bajo) y también **detecta incorrectamente objetos que no son perros** o asigna razas erróneas (*precisión* baja). Sin embargo, el **IoU (Intersection over Union)** es del **94%**, lo que es un

resultado excelente. Esto significa que **cuando el modelo detecta correctamente un perro, su localización (el *bounding box*) es extremadamente precisa**, gracias a la robustez del modelo YOLO.

La razón detrás de la baja precisión y *recall* probablemente radica en la naturaleza de las **imágenes de entrenamiento, que son "ideales"** (perros centrados, fondos limpios, poses estandarizadas). En contraste, las imágenes del mundo real proporcionadas al modelo presentan perros en **poses muy diferentes, con oclusiones, fondos complejos y variaciones de iluminación** que el modelo no ha visto previamente, lo que limita su capacidad de generalización.

Para ganar experiencia con la **optimización de modelos**, decidimos exportar el modelo YOLO a un formato **ONNX (Open Neural Network Exchange)**. La siguiente imagen muestra el tiempo de inferencia del modelo antes de ser exportado:

```
Average inference time: 0.013611733998914133 ms
Average preprocess time: 0.002309732999492553 ms
Average postprocess time: 0.0012685530000453582 ms
Average total process time per frame: 40.82208677499875 ms
```

Al principio, subestimamos el impacto de esta optimización, creyendo que los modelos ya venían en un formato suficientemente comprimido para la ejecución en computadoras domésticas y que, si se requería un modelo más preciso, este se buscaría específicamente. Sin embargo, para nuestra sorpresa, el modelo en el formato ONNX resultó en una **inferencia el doble de rápida en promedio**:

```
Average inference time: 0.010166214000491891 ms
Average preprocess time: 0.0039760630006639985 ms
Average postprocess time: 0.002385044999755337 ms
Average total process time per frame: 23.417404460000398 ms
```

4. Implementacion:

Como consigna final, fue necesario crear un *script* que permitiera **anotar automáticamente una carpeta de imágenes** y exportar estas anotaciones en formatos estándar. A continuación, se explicará brevemente la estructura del proyecto, organizada para maximizar la modularidad y la reutilización de código:

- **prepare_project.py**: Este *script* se encarga de la configuración inicial del entorno. Descarga las bases de datos necesarias para el funcionamiento del proyecto y crea las estructuras de carpetas donde se almacenarán los datos y resultados. Esto asegura que el entorno esté correctamente configurado antes de ejecutar cualquier otro componente.
- **src/**: Este directorio contiene el código fuente principal del proyecto, dividido en módulos lógicos:
    - **models/**: Contiene la clase ModelManager, que es responsable de cargar y gestionar los diferentes modelos de *deep learning* utilizados en el trabajo (clasificadores, detectores). También maneja la separación de la capa clasificadora (*head*) en los casos en que solo se necesita el *embedding* del modelo (por ejemplo, para la búsqueda por similitud).
    - **search/**: Contiene la clase VectorSearch, la cual encapsula la lógica para realizar búsquedas de similitud de *embeddings* utilizando las colecciones almacenadas en ChromaDB, aprovechando los varios modelos generadores de *embeddings* desarrollados.
    - **detection/**: Contiene la clase DogDetectionClassifier, que orquesta el *pipeline* completo de detección de perros con YOLO y posterior clasificación de la raza. Es la encargada de integrar el modelo de detección con el clasificador para procesar imágenes del mundo real.
    - **annotation/**: Contiene la clase AnnotationExporter, que se encarga de formatear y guardar las anotaciones generadas por el sistema en los formatos requeridos, como YOLO o COCO.
- **apps/**: Este directorio contiene el código fuente para las **tres aplicaciones Gradio** interactivas que se desarrollaron a lo largo del proyecto, permitiendo la demostración de cada etapa (buscador de similitud, clasificadores, *pipeline* de detección).

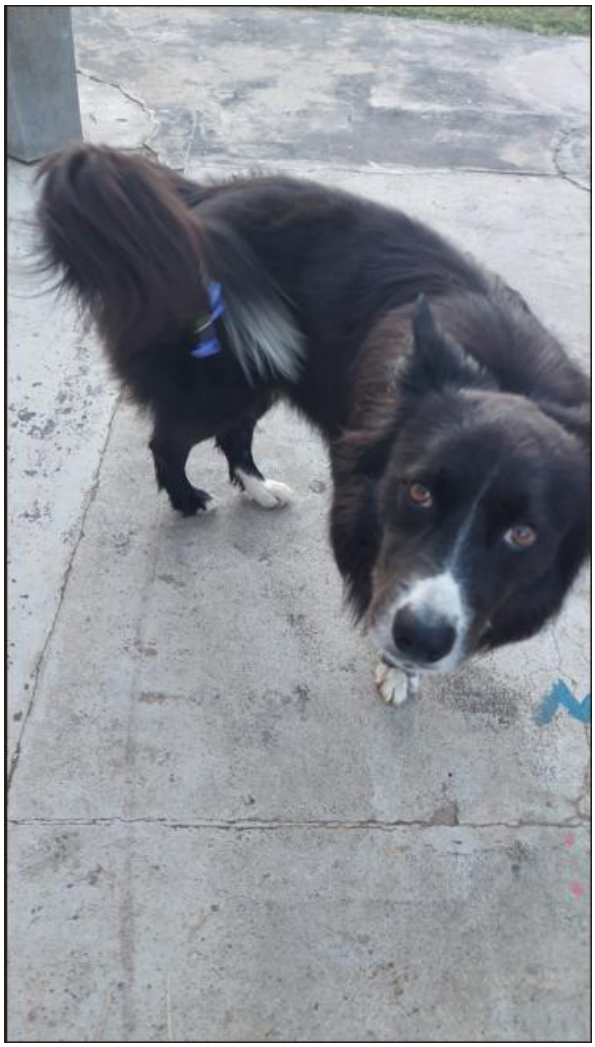
- **auto_annotate.py**: Este es el *script* principal que responde directamente a la consigna final del proyecto. Es el punto de entrada para el proceso de anotación automática.

Este último *script* (auto_annotate.py) ha sido diseñado para ser flexible y permite al usuario ingresar diversos argumentos a través de la línea de comandos para personalizar su funcionamiento:

- input_folder: Especifica la ruta de la carpeta que contiene las imágenes que se desean procesar y anotar.
- output_folder: Especifica la ruta de la carpeta de destino donde se guardarán las anotaciones. Si la carpeta no existe, será creada automáticamente.
- --confidence: Un argumento opcional que permite al usuario establecer un umbral de confianza. El modelo YOLO filtrará las detecciones cuya confianza esté por debajo de este valor. El valor predeterminado es 0.5.
- --format: Un argumento opcional para especificar el formato en el que se guardarán las anotaciones. Las opciones disponibles son yolo o coco. El valor predeterminado es both, lo que significa que las anotaciones se guardarán en ambos formatos.
- --device: Un argumento opcional que permite al usuario cambiar el dispositivo (CPU o GPU) en el que se realizará la inferencia. Esto es útil para optimizar el rendimiento en diferentes configuraciones de hardware.

5. Resultados:

Para evaluar la funcionalidad del *script* de anotación automática en un escenario práctico, contamos con la ayuda de un amigo que nos proporcionó **27 imágenes de su Border Collie llamado "Luca"**. A continuación, se presenta un ejemplo de una de las fotos utilizadas:



Al procesar estas 27 imágenes con nuestro programa, el sistema fue capaz de **reconocer la presencia de un perro en 18 de ellas (aproximadamente el 66%)**. De esas 18 detecciones confirmadas, las clasificaciones de raza obtenidas fueron las siguientes: **Groenendael en 5 ocasiones, Border Collie en 3 ocasiones, Collie en 3 ocasiones**, y varias confusiones aisladas con otras razas para el resto de las detecciones.

Al buscar imágenes de la raza **Groenendael**, se hizo evidente la razón detrás de las frecuentes confusiones:

Las similitudes morfológicas entre el Border Collie y el Groenendael son notables, especialmente en cuanto a la forma de la cabeza, el pelaje y la estructura corporal, lo que explica por qué el modelo pudo haber tenido dificultades para distinguir consistentemente entre ambas razas.

6. Conclusiones:

Este trabajo práctico ha sido una exploración exhaustiva de diversas **herramientas y técnicas de Computer Vision** que son actualmente utilizadas en la industria. Se ha puesto de manifiesto la **importancia crítica de ajustar y refinar los modelos de visión** preentrenados, así como los entrenados desde cero, para poder extraer el máximo rendimiento de sus arquitecturas y adaptarlos eficazmente a tareas específicas. La **transferencia de aprendizaje**, en particular, demostró ser una estrategia excepcionalmente potente para lograr alta precisión con un volumen de datos y tiempo de entrenamiento considerablemente menor en comparación con el entrenamiento de un modelo convolucional desde cero.

También se ha logrado tomar una dimensión clara del **poder de cómputo y la vasta cantidad de datos necesarios** para poder entrenar un modelo convolucional robusto desde 0. Este proceso no solo exige recursos computacionales significativos, sino que también implica una laboriosa y costosa tarea de **anotación manual de *datasets***, un aspecto crucial para la supervisión del aprendizaje.

Finalmente, se ha comprendido la **importancia fundamental de la optimización de modelos**, ya sea mediante técnicas de **cuantización** o la **exportación a formatos eficientes como ONNX**. Estas prácticas no solo son esenciales para el despliegue de modelos en entornos con recursos limitados, sino que también permiten extraer la **mejor *performance* en cuanto a tiempo de inferencia**, logrando una **reducción casi nula en la precisión** del modelo. Esto es vital para aplicaciones en tiempo real donde la latencia es un factor crítico.

Para poder lograr que el modelo final llegue a ser un producto es necesario pulir muchas cosas, principalmente enriquecer la ligera base de datos con la que contamos. Al añadir imágenes de tamaños variables se podría mejorar el rendimiento del modelo frente a situaciones reales.

7. Bibliografia:

Voxel51. (n.d.). *FiftyOne Documentation*. Recuperado de https://docs.voxel51.com/

Shaikh, A. (2020). *70-dog-breeds-image-data-set*. Kaggle. Recuperado de https://www.kaggle.com/datasets/ashishkumar468/70-dog-breedsimage-data-set

Ultralytics. (n.d.). *YOLO by Ultralytics Documentation*. Recuperado de https://docs.ultralytics.com/

PyTorch. (n.d.). *PyTorch Documentation*. Recuperado de https://pytorch.org/docs/stable/index.html

ChromaDB. (n.d.). *Chroma Documentation*. Recuperado de https://docs.trychroma.com/

Contenido dictado por la catedra de Computer Vision, TUIA UNR 2025.