



Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

Visión por Computadora
Trabajo Práctico Final
29 de junio de 2025

Pace, Bruno E. P-5295/7.

Docentes: Manson Juan Pablo, Ferrucci Costantino, Brugé Lucas.

Resumen

El presente informe detalla el desarrollo de un pipeline completo de visión por computadora, cuyo objetivo principal es la identificación automática de razas de perros en imágenes. El proyecto abarca desde la creación de un sistema de búsqueda por similitud de imágenes hasta la implementación de un sistema robusto de detección y clasificación en imágenes complejas. Se incluyen los procesos de entrenamiento, fine-tuning y optimización de modelos de Deep Learning, así como la evaluación de su desempeño mediante métricas estándar de la industria.

Contenidos

1. Introducción	3
2. Abordaje General y Buenas Prácticas	3
3. Buscador de Imágenes por Similitud	4
3.1. Preparación del Entorno y Datos	4
3.2. Creación de la Base de Datos Vectorial	4
3.3. ¿Por qué Faiss en lugar de FiftyOne?	5
3.4. Desarrollo de la Aplicación en Gradio y Clasificación Basada en Similitud con ResNet50	5
4. Entrenamiento y Comparación de Modelos de Clasificación	5
4.1. Fine tuning con ResNet18	6
4.2. Modelo Custom: Arquitectura y Resultados	6
4.3. Métrica NDCG@10	7
4.3.1. Conclusiones	7
4.4. Interfaz de Gradio	8
5. Detección y clasificación de perros en imágenes generales	9
6. Evaluación, Optimización y Anotación	10
6.1. Elección de imágenes	10
6.2. Implementación de detección y clasificación	10
6.2.1. Métricas IoU, Precision, F1-score, Recall	10
7. Exportado a ONNX	12
7.1. Explicación detallada de la función de detección y clasificación de perros	12
7.1.1. Detección con YOLOv5	12
7.1.2. Clasificación de la raza	13
7.2. Generación de anotaciones	13
8. Dificultades	14
9. Conclusiones Finales	15

1. Introducción

El **objetivo general** de este trabajo práctico es **desarrollar un pipeline completo de visión por computadora para la identificación de razas de perros en imágenes**. Este sistema abarca la creación de un buscador de imágenes por similitud y un sistema de detección y clasificación en imágenes complejas, incluyendo el entrenamiento y la optimización de modelos de Deep Learning. Para el desarrollo se utilizó el **Dataset: 70 Dog Breeds Image Dataset de Kaggle**. El proyecto se estructura en cuatro etapas incrementales, donde cada una construye sobre la anterior.

Este informe detalla las decisiones técnicas tomadas, los resultados obtenidos y los desafíos enfrentados durante el desarrollo, junto con un análisis crítico de los resultados y su aplicabilidad en escenarios reales.

2. Abordaje General y Buenas Prácticas

El trabajo sigue buenas prácticas de programación, desde un paradigma funcional. No sólo por prolijidad, sino por escalabilidad. Las funciones son de fácil interpretabilidad y mediante ellas se logra modularización.

3. Buscador de Imágenes por Similitud

El propósito de esta etapa fue **crear una aplicación que, dada una imagen de un perro, encuentre las imágenes más similares en el dataset**.

3.1. Preparación del Entorno y Datos

Antes de comenzar, se realizó la instalación de librerías necesarias como `faiss-cpu`, `onnx`, `onnxruntime`, y se importaron diversas librerías estándar y especializadas para procesamiento de datos, imágenes y frameworks de Deep Learning como PyTorch y TensorFlow. Se incluyó una función para descargar datos y modelos para reducir el tiempo de ejecución.

Se obtienen los modelos y datos necesarios de una carpeta en Google Drive, debido a que GitHub no permitía subir el modelo custom por su peso (más de 1Gb). Esta carpeta tiene los mismos datos del repositorio, más el modelo custom. Puede accederse a través del siguiente enlace: Carpeta de drive.

El dataset fue descargado desde KaggleHub. Se realizó un Análisis Exploratorio de Datos (EDA) simple que reveló la **cantidad de clases (70 razas únicas)** y la distribución de imágenes por clase. Se observó un **claro desbalance** en la cantidad de imágenes por raza, con algunas clases teniendo menos ejemplos. Las estadísticas mostraron que la **media es de aproximadamente 114 imágenes por clase**, el Q1 de 99 y el Q3 de 125. Es destacable también el mínimo (65) y el máximo (198). Es de destacar que al ser un dataset prácticamente listo para trabajar, las imágenes son de 224x224 y no es necesario aplicar operaciones en ese sentido.

Para solucionar el desbalance, se aplicó una estrategia de **balanceo simple** mediante **oversampling para clases minoritarias y undersampling para clases mayoritarias**. Se definió un `target_count_per_class` basado en el Q2 (mediana) y un `max_count_per_class` basado en el Q3.

En cuanto a las transformaciones: aleatoriamente se rota cada imagen hasta 15 grados, lo que ayuda al modelo a generalizar frente a variaciones en la orientación. También se volteá horizontalmente la imagen de forma aleatoria.

Luego, se aplican cambios en el brillo, contraste, saturación y tono de color, simulando diferentes condiciones de iluminación o cámaras. Después, se distorsiona la perspectiva de la imagen con cierta probabilidad, como si se hubiese tomado desde otro ángulo. Todo esto contribuye a que el modelo sea más robusto frente a variaciones del mundo real.

Finalmente, la imagen se convierte a un tensor (el formato que PyTorch utiliza para operar) y luego de nuevo a una imagen en formato PIL, posiblemente para seguir trabajando con transformaciones que requieren ese tipo de objeto.

3.2. Creación de la Base de Datos Vectorial

Para construir la base de datos vectorial, se utilizó un **modelo pre-entrenado en ImageNet, específicamente ResNet50**, para extraer vectores de características (embeddings) de cada imagen del dataset de perros. El modelo `keras_embedding_model` consiste en la base `ResNet50` (sin la capa superior, `include_top=False`) seguida de una capa `GlobalAveragePooling2D()` para reducir las características a un vector.

El proceso para generar estos embeddings involucró:

- Cargar y preprocesar cada imagen, convirtiéndola a formato RGB, convirtiéndola a un arreglo numpy y expandiendo sus dimensiones.
- Normalizar los embeddings resultantes utilizando `faiss.normalize_L2` para que la búsqueda por producto interno (IP) en FAISS sea equivalente a la similitud coseno.
- Finalmente, estos vectores se indexaron en una **base de datos vectorial FAISS (IndexFlatIP)**, que permite búsquedas eficientes de vecinos más cercanos. Se informó que el índice FAISS se creó con un total de vectores cargados.

Los embeddings generados fueron guardados en un archivo `dog_embeddings_resnet50.npz` para su posterior carga y evitar recomputaciones.

3.3. ¿Por qué Faiss en lugar de FiftyOne?

Se utiliza Faiss en lugar de FiftyOne que fue la herramienta preferida por la materia, debido a su complicación para implementarse en conjunto con Gradio, requisito de este trabajo.

3.4. Desarrollo de la Aplicación en Gradio y Clasificación Basada en Similitud con ResNet50

Se diseñó una interfaz con Gradio que permitía al usuario subir una imagen de un perro. El flujo de la aplicación `buscar_similares` es el siguiente:

1. La imagen de entrada es preprocesada y su embedding se extrae utilizando la misma lógica del `keras_embedding_model`.
2. Se realiza una búsqueda en el índice FAISS para encontrar las **10 imágenes más similares (vecinos más cercanos)** en el dataset.
3. A partir de las razas de estas 10 imágenes recuperadas, se implementa una lógica de **”voto mayoritario”** para predecir la raza del perro de la imagen de entrada. La raza predicha se muestra en la interfaz junto con las imágenes similares.

Aunque se implementó la interfaz Gradio para esta etapa, la misma no se lanzó en este punto porque se decidió no hacer tantas interfaces separadas.

4. Entrenamiento y Comparación de Modelos de Clasificación

Esta etapa se enfocó en el entrenamiento de modelos propios para mejorar la representación de características y la clasificación.

4.1. Fine tuning con ResNet18

El modelo ResNet18 fue seleccionado para realizar fine-tuning debido a su equilibrio entre capacidad de aprendizaje y eficiencia computacional. La arquitectura original fue modificada reemplazando su capa fully connected final (fc) por una nueva capa lineal con 70 unidades de salida (una por cada raza de perro). Durante el entrenamiento:

- Se congelaron todas las capas excepto la última, permitiendo que solo esta se adapte a nuestro problema específico.
- Se utilizó el optimizador SGD (gradiente descendiente estocástico), el cual evita actualizaciones repentinas. Momentum (0.9) para reducir oscilaciones y una tasa de aprendizaje de 0.01, evitando overfitting.
- El entrenamiento se limitó a 10 épocas, suficiente para lograr una adaptación sin overfitting.

Los resultados obtenidos demuestran la efectividad de este enfoque:

- **Precisión en entrenamiento:** 95.91 % (loss: 0.1475)
- **Precisión en validación:** 91.57 % (loss: 0.6476)
- **Pérdida en entrenamiento:** 0.1475
- **Pérdida en validación:** 0.6476

4.2. Modelo Custom: Arquitectura y Resultados

Se diseñó una red convolucional personalizada con la siguiente arquitectura detallada:

- **Capas convolucionales:**
 - Bloque 1: Conv2D (64 filtros 3x3) + BatchNorm + MaxPooling2D + Dropout (20 %)
 - Bloque 2: Conv2D (128 filtros 3x3) + BatchNorm + MaxPooling2D + Dropout (30 %)
 - Bloque 3: Conv2D (256 filtros 3x3) + BatchNorm + MaxPooling2D + Dropout (40 %)
- **Capas densas:**
 - Flatten + Dense (512 unidades) + BatchNorm + Dropout (50 %)
 - Capa de salida con activación softmax

El modelo fue entrenado con:

- Optimizador Adam (lr=0.01)
- Early Stopping (pacientia=5). Corta entrenamiento si luego de 5 épocas no hay mejora.

- ReduceLROnPlateau (factor=0.1, paciencia=3). Reduce la velocidad de aprendizaje si el modelo se estanca. Multiplica el LR por 0.1.

A pesar de su diseño y arquitecturas pensadas para este problema, no se obtienen buenos resultados. Hace overfitting en el caso de train y en validación las métricas no son optimas: 38.57 % de precisión en validación y loss alto de 2.8186. Este modelo directamente no es tenido en cuenta más que para un fin analítico. Pareciera tener una arquitectura demasiado poco compleja para el set de datos.

4.3. Métrica NDCG@10

El Normalized Discounted Cumulative Gain (NDCG) es una métrica que evalúa la calidad del ranking en sistemas de recuperación de información. Para este caso compara el ranking de imágenes similares devuelto por el sistema contra un ranking ideal.

En cuanto a su interpretación, los valores cercanos a 1 indican que las imágenes más relevantes (misma raza) aparecen primero.

Al analizar los tres modelos evaluados, observamos diferencias significativas en su desempeño. El ResNet50 pre-entrenado obtuvo el mejor resultado con un NDCG@10 promedio de 0.8955, demostrando una alta capacidad para generar rankings precisos, aunque con un tiempo considerable de procesamiento de más de 4 minutos para generar los embeddings de las 700 imágenes de prueba. Su baja desviación estándar de 0.1179 indica que mantuvo un rendimiento consistente en la mayoría de los casos.

El modelo ResNet18 con fine-tuning mostró un rendimiento muy cercano al ResNet50, alcanzando un NDCG@10 de 0.8715, con una ligera mayor variabilidad (desviación estándar de 0.1341). Sin embargo, su ventaja principal radica en la eficiencia, ya que completó el procesamiento en solo 1 minuto y medio, menos de un tercio del tiempo requerido por ResNet50. Esta combinación de buen rendimiento y velocidad lo convierte en la opción más equilibrada para aplicaciones prácticas.

Por otro lado, el modelo personalizado (custom) presentó el desempeño más bajo con un NDCG@10 de solo 0.5553, acompañado de una alta variabilidad (desviación estándar de 0.2421) que indica resultados inconsistentes. Además, requirió 3 minutos y 13 segundos para procesar las imágenes, un tiempo considerablemente mayor al ResNet18 a pesar de su menor efectividad. Solo 192 de las 700 imágenes contribuyeron significativamente al cálculo del NDCG, lo que sugiere que en muchos casos el modelo no encontró imágenes lo suficientemente relevantes para incluirlas en la evaluación. Confirma lo que se analizaba con las métricas anteriores.

4.3.1. Conclusiones

Se puede concluir que aunque ResNet50 tiene una ligera ventaja en precisión (aproximadamente 2.4 % mejor en NDCG), esta diferencia no justifica su mayor costo computacional en la mayoría de escenarios prácticos. ResNet18 ofrece un equilibrio casi ideal entre precisión y eficiencia, siendo significativamente más rápido mientras mantiene una calidad de rankings muy cercana a la de su contraparte más pesada.

Por eso se utiliza ResNet18 de aquí en adelante como preferencia, por más que figure en algunas interfaces de Gradio ResNet50.

4.4. Interfaz de Gradio

En esta instancia sí se deja constancia de una interfaz de Gradio que compara una imagen ingresada por el usuario con elección de cualquiera de los modelos. Fue necesario crear un índice también para el modelo ResNet18 para la visualización de las 10 imágenes más similares.

5. Detección y clasificación de perros en imágenes generales

Se realiza una detección de los perros (clase 16) con YOLOv8. Este modelo "You only look once." encierra al animal en un bounding box y luego se recorta, para clasificarlo. Nuevamente, se setea un menú desplegable para dar a elegir al usuario qué modelo utilizar.

En grandes rasgos, se puede evaluar que dependiendo de la complejidad de la imagen, si bien el perro es correctamente detectado, la clasificación no es óptima.

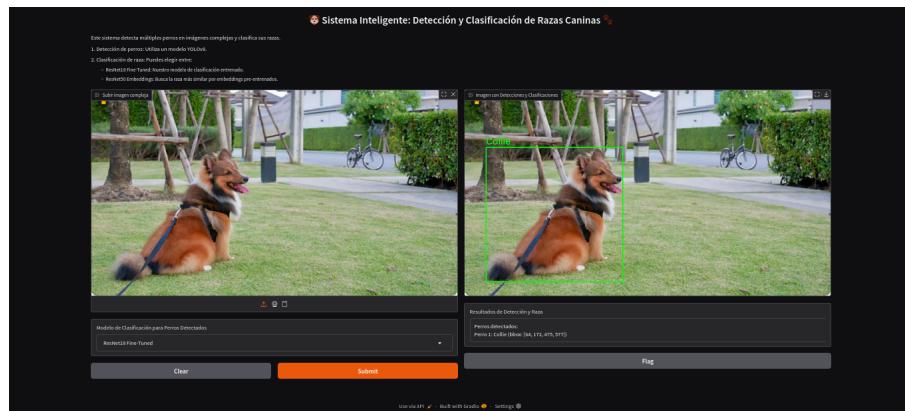


Figura 1: Caso correcto con interfaz

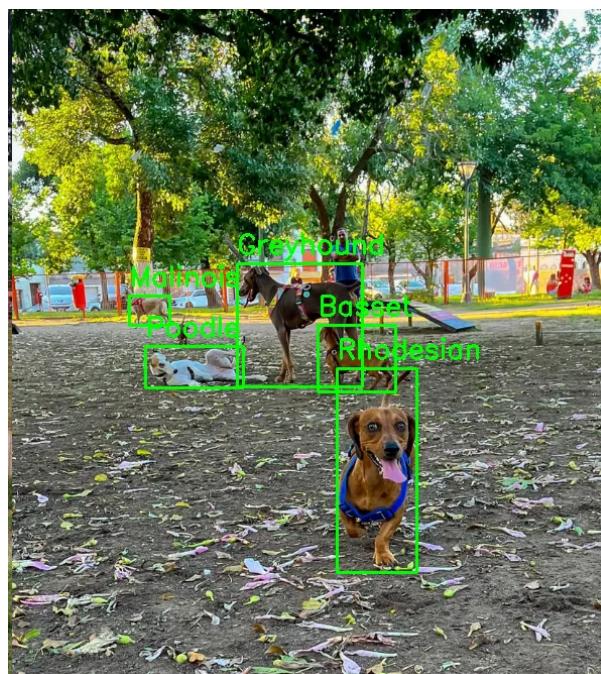


Figura 2: Caso errático cuando los perros no están tan claros o cerca

6. Evaluación, Optimización y Anotación

6.1. Elección de imágenes

Para la construcción del "dataset" de 10 imágenes se navegó por la web, buscando imágenes donde haya un escenario un poco más complejo que los del dataset original del clasificador. Conforme la imagen contiene más perros, objetos o colores el rendimiento disminuye.



(a) Perros en poses de carrera y con bozales



(b) Escena con múltiples objetos



(c) Difícil diferenciación

Figura 3: Ejemplos representativos del dataset de evaluación con escenarios complejos

Como herramienta de anotación se utilizó CVAT online e importado en formato COCO. Esto fue útil e intuitivo.

6.2. Implementación de detección y clasificación

Se emplea lo realizado anteriormente con una modificación en la salida de la función para incluir el bounding box.

En cuanto a las métricas, es destacable mencionar que la IoU obtiene buenos valores, lo que indica que la detección de YOLOv8 funciona muy bien.

6.2.1. Métricas IoU, Precision, F1-score, Recall

```

1 AP @[IoU=0.50:0.50 | area=all    | maxDets=100] = 0.144
2 AP @[IoU=0.50      | area=all    | maxDets=100] = 0.144
3 AP @[IoU=0.75      | area=all    | maxDets=100] = -1.000
4 AP @[IoU=0.50:0.50 | area=small   | maxDets=100] = -1.000
5 AP @[IoU=0.50:0.50 | area=medium  | maxDets=100] = 0.250
6 AP @[IoU=0.50:0.50 | area=large   | maxDets=100] = 0.084
7 AR @[IoU=0.50:0.50 | area=all    | maxDets=1 ] = 0.143
8 AR @[IoU=0.50:0.50 | area=all    | maxDets=10 ] = 0.143
9 AR @[IoU=0.50:0.50 | area=all    | maxDets=100] = 0.143
10 AR @[IoU=0.50:0.50 | area=small   | maxDets=100] = -1.000
11 AR @[IoU=0.50:0.50 | area=medium  | maxDets=100] = 0.250
12 AR @[IoU=0.50:0.50 | area=large   | maxDets=100] = 0.083
13
14 Resumen:
15 Precision (AP@0.50): 0.144
16 Recall (AR@0.50):    0.143
17 F1-Score:            0.144

```

Listing 1: Métricas detalladas de evaluación

En la evaluación de modelos de detección de objetos, tres métricas clave son precisión, recall y F1-Score. La precisión indica cuántas de las detecciones realizadas por el modelo fueron correctas, mientras que el recall mide cuántos de los objetos realmente presentes fueron efectivamente detectados. El F1-Score combina ambas en una única métrica para reflejar el equilibrio entre falsos positivos y falsos negativos.

En este caso, los valores bajos en las tres métricas indican que el modelo comete muchos errores al clasificar e identificar objetos, aunque logra localizar sus posiciones con cierto éxito. Esto sugiere que el detector basado en YOLOv8 funciona razonablemente bien para encontrar regiones de interés, pero el clasificador asociado no logra distinguir correctamente las clases, lo que limita el rendimiento global del sistema.

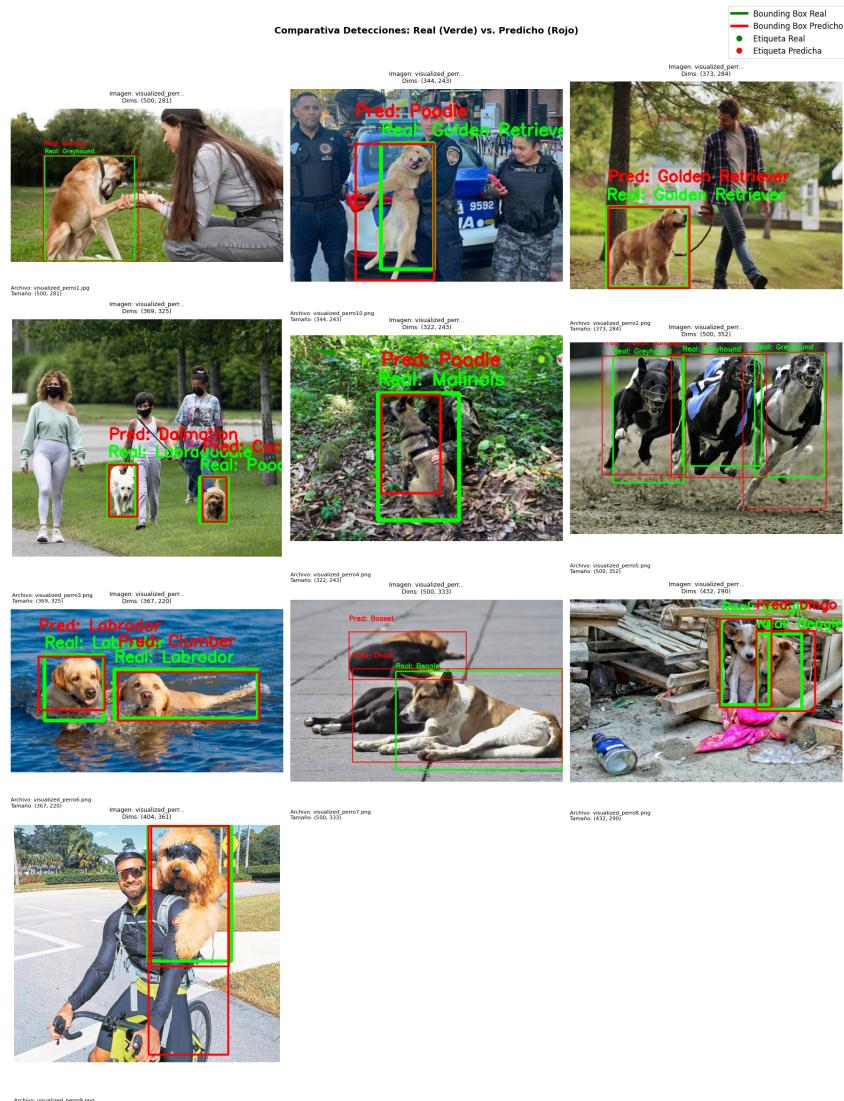


Figura 4: Resultado visual de la etapa 4

En la imagen notamos que las áreas de los animales son bastante precisas pero las clases no corren la misma suerte.

7. Exportado a ONNX

Se pone el modelo en modo evaluación (`eval()`). Esto es muy importante porque durante el entrenamiento, ciertas capas como *Dropout* o *BatchNorm* se comportan distinto para ayudar al modelo a generalizar. Pero cuando se usa el modelo para predecir (inferencia), esas capas deben comportarse de manera fija. Así que, `eval()` garantiza que el modelo no haga cambios aleatorios y produzca resultados consistentes.

Luego, se crea una entrada de ejemplo, un tensor ficticio con las mismas dimensiones que las imágenes que el modelo espera recibir (por ejemplo, 1 imagen, 3 canales de color, 224x224 píxeles). Esta entrada sirve para que PyTorch trace cómo fluye la información a través del modelo y pueda generar una versión equivalente en ONNX.

Con la entrada ficticia, PyTorch exporta el modelo a un archivo `.onnx`. Este archivo contiene toda la información necesaria: las operaciones, cómo se conectan, los pesos, y detalles sobre las entradas y salidas.

Al usar ONNX Runtime, se acelera el tiempo de ejecución porque está específicamente optimizado para hacer inferencia. Por eso, en las mediciones realizadas, el tiempo promedio por imagen pasó de:

- **Modelo original en PyTorch:** 207.58 ms por imagen.
- **Modelo exportado y ejecutado con ONNX Runtime:** 71.08 ms por imagen.

Esta mejora se debe a que ONNX Runtime optimiza el modelo para inferencia, fusiona operaciones, elimina overhead innecesario y aprovecha mejor el hardware disponible, haciendo la predicción mucho más rápida sin perder precisión.

7.1. Explicación detallada de la función de detección y clasificación de perros

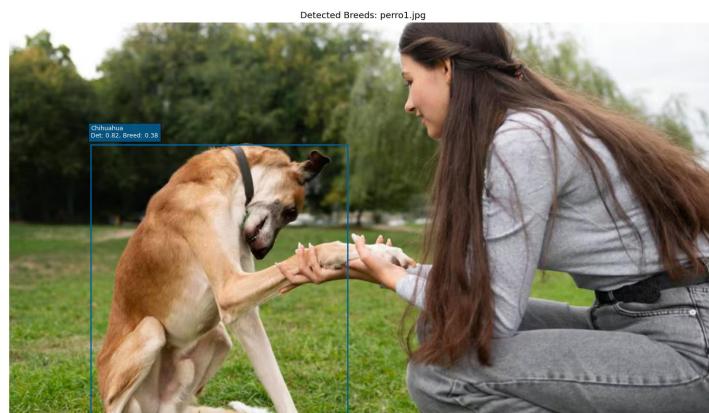


Figura 5: Ejemplo de resultado del detector (YOLOv5) y clasificador (utilizando el modelo onnx).

7.1.1. Detección con YOLOv5

Se utiliza el modelo YOLOv5 para detectar objetos en la imagen completa. El detector devuelve una lista de objetos con sus coordenadas en píxeles y la clase detectada. En este

caso, la función filtra únicamente las detecciones cuya clase corresponde a perros (clase 16 en el conjunto COCO). Esto asegura que solo se procesen regiones de interés relevantes.

7.1.2. Clasificación de la raza

Para cada perro detectado, se extrae su región (bounding box) de la imagen original. Se verifica que las coordenadas sean válidas y que el recorte no esté vacío. Esta región se preprocesa y se pasa por el modelo clasificador ONNX, que devuelve un vector de probabilidades para cada raza posible.

El índice de la clase con mayor probabilidad se identifica como la raza predicha. Para interpretar la confianza de la predicción, se calcula la probabilidad suavizada con la función `softmax` para esa clase. En caso de error durante la clasificación, se maneja la excepción y se marca la raza como "desconocida".



Figura 6: Visualización de la detección y clasificación de la raza en una imagen.

7.2. Generación de anotaciones

Para cada perro identificado, se generan anotaciones en dos formatos estándar en visión por computadora:

- **Formato YOLOv5:** coordenadas normalizadas respecto a la anchura y altura de la imagen, con la clase numérica seguida por los valores de centro y tamaño del bounding box.
- **Formato COCO:** un diccionario con la información completa, que incluye las coordenadas en píxeles, el ID de categoría, nombre de la raza, puntuación de confianza del detector y la confianza de la clasificación.

8. Dificultades

La principal dificultad fue la carencia de un entorno de uso de GPU con una cuota considerable para entrenar (o re-entrenar) los modelos e ir ajustando los parámetros a medida de cómo respondían las métricas.

Considero que el modelo custom, podría haberse mejorado de haber tenido más tiempo, capacidad de cómputo y buscar algún punto simple de diferenciación entre clases que marque la diferencia.

Los párrafos anteriores explican también por qué se cargan los modelos desde Drive y no se corren en cada ejecución del notebook. Se terminó utilizando la CPU de Colab para el desarrollo del trabajo práctico.

También es destacable que, debido a un error persistente y que no fue posible modificar en un tiempo razonable, no pude lograr calcular la métrica ndcg con GPU.

Otra dificultad fue la elección de las imágenes complejas. Considero que fueron lo suficientemente complejas para YOLO, mostrándose bastante consistentes, mientras que para los clasificadores fue demasiado. No se logró encontrar un balance perfecto en ese sentido ya que, cuando ambos funcionaban bien, las imágenes eran demasiado simples.

Por último, ejecutar el TP de forma local fue complejo porque no cuenta con una computadora lo suficientemente potente para que no me lleve demasiado tiempo. Por eso se optó por trabajar en Colab.

9. Conclusiones Finales

Puedo remarcar una instancia realmente de aprendizaje donde me enfrenté con un problema extrapolable a la vida real. Considero que el funcionamiento de la detección es adecuado pero la clasificación es deficiente. En imágenes complejas no cumple con lo esperado y no lo recomendaría para una aplicación en la vida real. Puedo recalcar las herramientas utilizadas para un entorno productivo como faiss, la exportación a ONNX y CVAT que incluso hasta puede usarse online.