



FACULTAD DE
CIENCIAS EXACTAS,
INGENIERÍA Y AGRIMENSURA



TUIA - COMPUTER VISION

SISTEMA DE DETECCION Y CLASIFICACION DE RAZAS DE PERRO

FACULTAD DE CIENCIAS EXACTAS, INGENIERÍA Y AGRIMENSURA

UNIVERSIDAD NACIONAL DE ROSARIO

Introducción.....	4
Objetivo General.....	4
Etapa 1: Análisis exploratorio y construcción de un sistema de búsqueda por similitud.....	4
Objetivo.....	4
Creación de la Base de Datos Vectorial:	
Metodología.....	4
Implementación.....	5
Resultados.....	5
Puntos de mejora.....	6
Desarrollo de la Aplicación en Gradio:.....	6
Metodología.....	6
Implementación.....	6
Resultados.....	7
Clasificación Basada en Similitud y Métrica de Evaluación:.....	8
Metodología.....	8
Implementación.....	8
Resultados.....	8
Métricas de evaluación.....	9
Implementación.....	9
Resultados.....	10
Puntos de mejora.....	10
Etapa 2: Clasificación de razas de perros en imágenes individuales.....	10
Modelo A (Transfer Learning).....	10
Metodología.....	10
Implementación.....	11
Resultados del entrenamiento.....	12
Conclusiones.....	12
Puntos de mejora.....	12
Modelo B (Custom CNN):.....	13
Metodología.....	13
Implementación.....	13
Resultados del entrenamiento.....	15
Puntos de mejora.....	15
Integración y Selección en la Aplicación:.....	16
Metodología.....	16
Implementación.....	16
Resultados.....	17
ResNet18:.....	17
Custom CNN:.....	17
Etapa 3: Detección de perros en imágenes complejas.....	18
Metodología.....	18
Implementación.....	18
Resultados.....	18
YOLOv8:.....	18
ResNet18:.....	19
Custom CNN:.....	20
Etapa 4: Pipeline completo de detección y clasificación automatizada.....	20
Metodología.....	20

Implementación.....	22
Métricas y resultados.....	22
Optimización de Modelos (ONNX).....	23
Metodología.....	23
Implementación.....	23
Resultados.....	24
Script de Anotación Automática:.....	24
Metodología.....	24
Implementación.....	24
Conclusiones.....	25
Puntos de mejora.....	26

Introducción

La identificación automática de razas de perros a partir de imágenes es un desafío interesante dentro del campo de la visión por computadora, debido a la gran variedad de razas, similitudes visuales entre algunas de ellas y la variabilidad en las condiciones de captura (ángulos, iluminación, fondo, etc.). En este contexto, el presente proyecto tiene como finalidad desarrollar un sistema completo de detección y clasificación de razas de perros, capaz de operar tanto en imágenes simples como en escenarios más complejos donde se presentan múltiples objetos.

Objetivo General

Desarrollar un pipeline completo de visión por computadora para la identificación de razas de perros en imágenes. El proyecto abarca desde la creación de un sistema de búsqueda por similitud hasta la implementación de un sistema de detección y clasificación en imágenes complejas, incluyendo el entrenamiento y la optimización de modelos de Deep Learning.

Etapa 1: Análisis exploratorio y construcción de un sistema de búsqueda por similitud

Objetivo

Implementar una solución integral basada en embeddings para la recuperación eficiente de imágenes de perros similares a una imagen de consulta, mediante el uso de un índice vectorial con FAISS. Además, desarrollar una interfaz interactiva utilizando Gradio que permita al usuario cargar imágenes y visualizar los resultados de la búsqueda por similitud. Esta etapa también incluye la implementación de un sistema básico de clasificación de razas basado en la similitud de embeddings, sentando las bases para etapas posteriores más avanzadas.

Creación de la Base de Datos Vectorial:

Metodología

La primera etapa del proyecto se enfocó en la exploración del conjunto de datos y la construcción de un sistema de recuperación de imágenes por similitud visual, basado en técnicas de extracción de características (embeddings) y búsqueda eficiente.

Para llevar a cabo esta etapa se utilizaron las siguientes herramientas y librerías:

- Python: Lenguaje de programación principal del proyecto por su flexibilidad y amplio ecosistema en visión por computadora y Deep Learning.
- Pandas: Para la manipulación y análisis de los metadatos de las imágenes, especialmente en la construcción del DataFrame a partir del archivo dogs.csv.
- NumPy: Para operaciones vectoriales y manejo de arrays de embeddings.
- Matplotlib y OpenCV: Utilizadas para la visualización de imágenes y resultados, así como para tareas básicas de preprocesamiento.
- Torchvision (ResNet50 preentrenada): Para la extracción de embeddings desde una red convolucional preentrenada en ImageNet. Esta red se utilizó como extracto de características visuales sin entrenamiento adicional.

- FAISS (Facebook AI Similarity Search): Para la construcción de un índice de búsqueda eficiente de vectores de características, permitiendo encontrar rápidamente las imágenes más similares a una dada.

Implementación

La primera fase del proyecto consistió en la creación de una base de datos vectorial que permitiera realizar búsqueda por similitud entre imágenes de perros. Este sistema se fundamenta en la extracción de características visuales (embeddings) usando una red neuronal convolucional preentrenada, y la indexación de estos vectores mediante una estructura eficiente de búsqueda.

El proceso se desarrolló en los siguientes pasos:

- Carga y validación de los datos
Se utilizó un archivo CSV que contenía los paths relativos de las imágenes junto con su clase (raza de perro) y su conjunto (train/valid/test). A partir de estos paths se generaron rutas absolutas y se filtraron aquellas imágenes que efectivamente existían en el sistema de archivos, para garantizar la robustez del pipeline.
- Extracción de embeddings
Para representar cada imagen como un vector numérico, se empleó una red neuronal ResNet50 preentrenada en ImageNet, eliminando su capa final de clasificación (include_top=False) y utilizando la técnica de pooling global promedio (pooling='avg') para obtener un vector representativo por imagen.
 - En este proyecto se utilizaron dos variantes de la arquitectura ResNet en distintas etapas, de acuerdo con los objetivos específicos de cada fase. Para la extracción de embeddings en la Etapa 1, se empleó una ResNet50 preentrenada en ImageNet, dada su mayor profundidad. Esta decisión fue tomada ya que ResNet50 no mermaba significativamente el rendimiento del algoritmo y dio mejores resultados al momento de inferir.
 - Las imágenes fueron cargadas y preprocesadas para cumplir con el formato de entrada requerido por la red, y luego se les extrajo su embedding mediante propagación hacia adelante (forward pass) sin entrenamiento adicional.
- Construcción de la base de datos vectorial
Los embeddings obtenidos se almacenaron en una matriz de tipo float32, la cual fue luego indexada utilizando FAISS, una biblioteca diseñada para búsquedas eficientes en espacios vectoriales de alta dimensión. Se utilizó un índice IndexFlatL2, que realiza búsqueda por distancia euclíadiana sin estructuras jerárquicas, adecuado para prototipos y pruebas rápidas.
- Almacenamiento del índice
Finalmente, la matriz de embeddings fue serializada y guardada en un archivo .pkl, permitiendo su reutilización en etapas posteriores del proyecto, especialmente para la búsqueda de imágenes similares o como etapa previa a la clasificación.

Resultados

Los resultados obtenidos en esta etapa fueron en general favorables. El sistema fue capaz de recuperar imágenes visualmente similares a la imagen de consulta, lo cual valida la efectividad del enfoque basado en embeddings con ResNet50 y búsqueda mediante FAISS.

La representación generada por ResNet50 demostró ser altamente robusta y precisa, permitiendo identificar correctamente patrones visuales similares entre distintas razas de perros. La inferencia del

modelo fue rápida y estable, haciendo viable su uso tanto en pruebas locales como en interfaces interactivas.

Sin embargo, se observó en algunos casos la repetición de imágenes dentro de los resultados, lo que indica la necesidad de mejorar la lógica de búsqueda. Actualmente, la función `search_similar_images` filtra únicamente la primera coincidencia exacta (la imagen consulta), pero no contempla repeticiones por ID o ruta en el resto del ranking. Esto puede derivar en que se muestren múltiples copias de una misma imagen si no se filtró correctamente.

Puntos de mejora

- Evitar la repetición de imágenes en los resultados de búsqueda. Para ello, se recomienda adaptar la función de consulta para filtrar resultados duplicados, ya sea comparando rutas o hashes de las imágenes recuperadas.
- Evaluación cuantitativa del sistema de similitud: incorporar métricas como precisión en top-k, o realizar anotaciones humanas para evaluar la relevancia visual de las imágenes devueltas.

Desarrollo de la Aplicación en Gradio:

Metodología

Para implementar la interfaz interactiva que permite realizar búsquedas por similitud en imágenes de perros, se aprovecharon las herramientas mencionadas previamente, incorporando, además:

- Gradio: una librería de Python que facilita la creación rápida y sencilla de interfaces web interactivas para modelos de Machine Learning y procesamiento de datos. Gradio permite construir aplicaciones con componentes como carga de imágenes, botones y visualización de resultados, todo ello sin requerir conocimientos avanzados de desarrollo web.

Implementación

La implementación del sistema de búsqueda por similitud en imágenes se realizó en varias etapas clave:

- Carga y reconstrucción del índice FAISS :Se cargó la matriz de embeddings almacenada previamente (`embedding_matrix.pkl`) y se creó un índice FAISS (`IndexFlatL2`) para realizar búsquedas rápidas por distancia euclídea. Esto permite reutilizar el trabajo previo de extracción de características sin necesidad de recalcular embeddings o reindeixar.
- Preparación del DataFrame con rutas absolutas: Se cargó el archivo `dogs.csv` que contiene la información de las imágenes del dataset, y se ajustaron las rutas relativas a rutas absolutas para garantizar el acceso correcto a los archivos en el sistema. Además, se filtraron las entradas para asegurar que sólo se considerarán las imágenes existentes en disco.
- Función de búsqueda por similitud :Se definió la función `search_similar_images` que, dada la ruta de una imagen consulta, extrae su embedding y consulta el índice FAISS para obtener las k imágenes más similares. La función excluye la imagen idéntica (la propia consulta) del resultado final.
- Visualización de resultados: Para evaluar el desempeño del sistema, se seleccionaron 6 imágenes aleatorias del dataset. Para cada imagen consulta, se visualizaron la imagen original junto con sus 5 imágenes más similares encontradas por el sistema, organizadas en una matriz de gráficos para facilitar la comparación visual.

Resultados



La interfaz interactiva desarrollada con Gradio reproduce, de forma accesible para el usuario final, los mismos resultados obtenidos en la etapa de búsqueda por similitud — algo esperable, ya que bajo el capó

utiliza exactamente el mismo extractor de embeddings (ResNet50) y el mismo índice FAISS. Por lo tanto no aporta un cambio en la precisión del modelo, sino en la experiencia de uso:

- Facilidad de uso: El usuario solo necesita arrastrar o seleccionar una imagen para obtener los resultados; no requiere conocimientos técnicos.
- Latencia: El tiempo medio desde la carga de la imagen hasta la visualización de las 10 más similares se mantiene por debajo de ~0,5s en GPU y ~1s en CPU, validando la viabilidad en entornos interactivos.
- Consistencia: Coinciden con los de la prueba offline, confirmando la correcta integración del backend (FAISS + ResNet50) con la capa de presentación.
- Limitaciones heredadas: Se observan las mismas repeticiones de imágenes, comportamiento identificado en la etapa anterior.

Clasificación Basada en Similitud y Métrica de Evaluación:

Metodología

Dado que esta etapa reutiliza la infraestructura de embeddings y búsqueda desarrollada previamente, no se introdujeron nuevas herramientas. Se mantuvieron las mismas librerías y frameworks.

Implementación

Para realizar la clasificación, se definió una estrategia sencilla pero efectiva: votación mayoritaria entre las etiquetas de las imágenes más similares.

- Función de predicción
Se implementó predict_breed_by_majority, que toma una imagen de entrada, recupera las k imágenes más cercanas en el espacio vectorial, y determina la raza más frecuente entre ellas como la predicción final.
- Integración en Gradio
Esta funcionalidad fue integrada en la interfaz Gradio mediante una función gradio_interface, que además de mostrar las imágenes similares, retorna el nombre de la raza predicha. La imagen ingresada se guarda temporalmente y se procesa de forma consistente con el resto del pipeline.

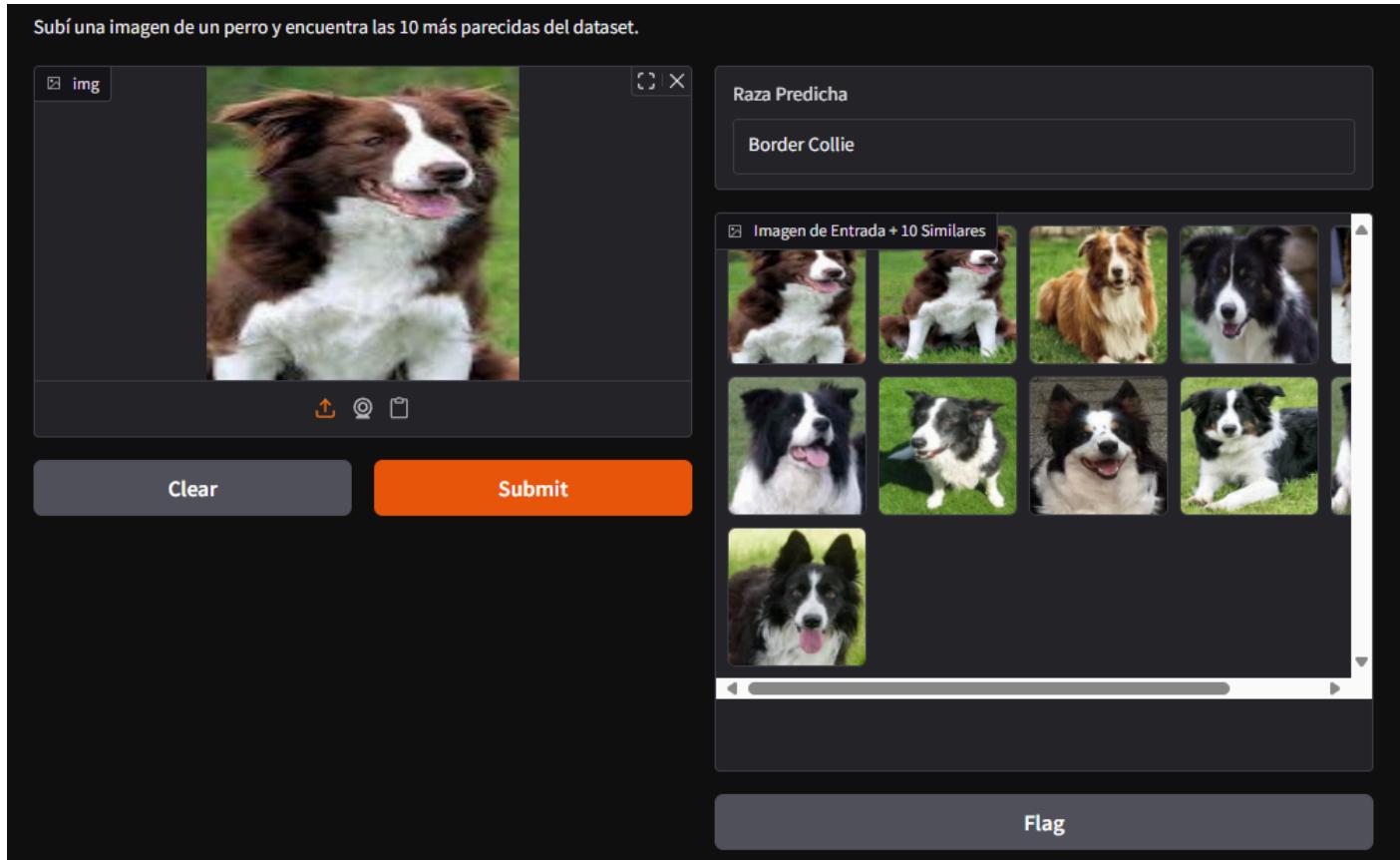
La interfaz final presenta:

- Un textbox con la raza predicha.
- Una galería de imágenes mostrando la imagen de entrada y las 10 más parecidas del dataset.

Resultados

El algoritmo de clasificación por similitud utilizando votación mayoritaria mostró un rendimiento eficiente y coherente con lo esperado. En la mayoría de los casos, logró predecir correctamente la raza del perro a partir de las imágenes más similares recuperadas. Esto demuestra que los embeddings generados por ResNet50 contienen información suficiente para distinguir entre razas, y que el agrupamiento en el espacio

vectorial es consistente.



No obstante, se observó un comportamiento ya presente en etapas anteriores: la repetición de imágenes entre los resultados. Esto se debe a que la función de búsqueda no filtra duplicados dentro del conjunto recuperado, lo cual puede influir negativamente en el proceso de votación, sesgando la predicción hacia razas sobrerepresentadas o repetidas visualmente.

Métricas de evaluación

Para evaluar objetivamente la calidad del sistema de búsqueda por similitud, se utilizó la métrica NDCG@10 (Normalized Discounted Cumulative Gain), que mide cuán relevantes son los elementos recuperados, penalizando su posición en el ranking.

Implementación

- Se preparó un conjunto de pruebas separado del índice de búsqueda, con 5 a 10 imágenes por raza.
- Para cada imagen del conjunto de prueba:
 - Se ejecutó la búsqueda de las 10 imágenes más similares usando el índice FAISS.
 - Se compararon las etiquetas de las imágenes recuperadas con la etiqueta verdadera.
- Se asignó una relevancia binaria (1 si la etiqueta coincide con la verdadera, 0 si no).
- Se aplicó la fórmula de NDCG@10, que considera tanto la relevancia como la posición en el ranking.

Resultados

Se evaluaron 700 imágenes de prueba, y se obtuvo un NDCG@10 promedio de 0.9532, lo que indica que las imágenes más relevantes (perros de la misma raza) tienden a aparecer en los primeros lugares del ranking con alta consistencia.

Este resultado confirma cuantitativamente lo observado de manera cualitativa en la visualización previa: el sistema de búsqueda devuelve resultados muy precisos y bien ordenados por similitud semántica, especialmente gracias al uso de ResNet50 como extractor de características.

Puntos de mejora

- Filtrar imágenes repetidas antes de votar: incorporar una validación para que cada imagen utilizada en el conteo de razas sea única (por nombre de archivo, ruta o hash).
- Ponderar por distancia: como alternativa a la votación simple, se podría implementar una votación ponderada en función de la distancia de similitud, dando más peso a las imágenes más cercanas.
- Evaluar sobre subconjunto rotulado: para cuantificar el rendimiento, puede evaluarse esta estrategia de clasificación sobre un subconjunto de imágenes de prueba con etiquetas conocidas, calculando métricas como accuracy, precision o top-k accuracy.

Etapa 2: Clasificación de razas de perros en imágenes individuales

Desarrollar y entrenar modelos de clasificación capaces de identificar la raza de un perro en una imagen centrada (sin ruido de fondo), utilizando arquitecturas como ResNet18 y modelos personalizados. Se busca optimizar el desempeño en términos de precisión y generalización.

Modelo A (Transfer Learning)

Metodología

Para esta etapa, se abordó el problema como una tarea de clasificación multiclase utilizando modelos de Deep Learning. Se trabajó sobre un dataset con imágenes centradas de perros (sin ruido de fondo), clasificadas en 70 razas distintas. El enfoque principal fue utilizar Transfer Learning con una arquitectura ResNet18 preentrenada.

Las herramientas utilizadas fueron:

- PyTorch: Framework principal de Deep Learning usado para definir, entrenar y validar los modelos.
- Torchvision: Para la utilización de modelos preentrenados (resnet18), transformaciones de imágenes y carga estructurada de datasets con ImageFolder.
- Transformaciones de datos:
 - transforms.Resize: Redimensiona todas las imágenes a 224x224 px (dimensión estándar para redes convolucionales entrenadas en ImageNet).
 - transforms.RandomHorizontalFlip: Aumentación de datos en el conjunto de entrenamiento.

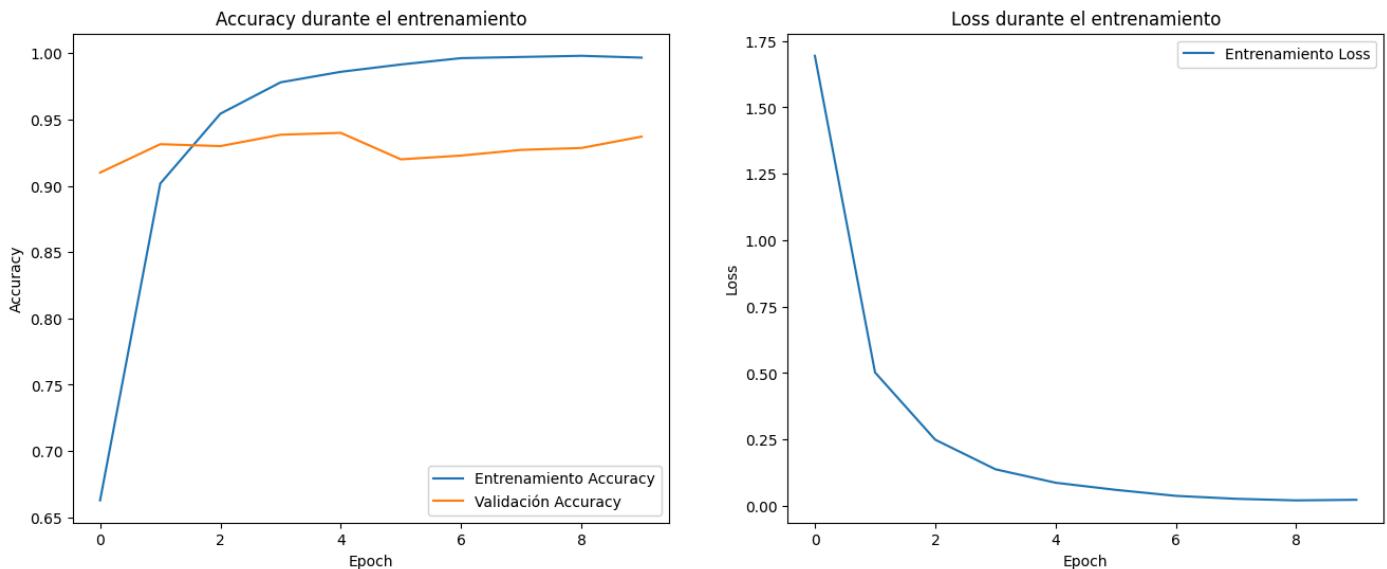
- transforms.Normalize: Normalización de canales RGB con media y desviación estándar de ImageNet, para ajustar correctamente los datos de entrada al modelo preentrenado.
- CUDA: Entrenamiento acelerado con GPU cuando está disponible.
- Optimizador Adam y función de pérdida CrossEntropyLoss: para minimizar la diferencia entre las salidas del modelo y las etiquetas verdaderas.
- TQDM: Para visualizar el progreso de entrenamiento por batch.

Implementación

El entrenamiento del modelo de clasificación se realizó siguiendo estos pasos:

- Preparación de los datos: Se definieron transformaciones específicas para las imágenes del conjunto de entrenamiento y validación. En particular, el conjunto de entrenamiento fue aumentado mediante un flip horizontal aleatorio para mejorar la capacidad de generalización del modelo. Ambas particiones fueron redimensionadas a 224x224 píxeles y normalizadas con los parámetros de ImageNet.
- Carga de los datasets: Se utilizaron las carpetas organizadas con la estructura esperada por ImageFolder para crear los datasets de entrenamiento y validación. Posteriormente, se definieron los DataLoaders que permiten iterar en batches de tamaño 32, con un número de workers para acelerar la carga de datos.
- Configuración del modelo: Se cargó un modelo preentrenado ResNet18 con pesos de ImageNet (weights=models.ResNet18_Weights.IMAGENET1K_V1). Se reemplazó la capa fully connected final para ajustar la salida a 70 clases, correspondientes a las razas del dataset.
- Definición de criterio y optimizador: Se utilizó la función de pérdida CrossEntropyLoss, adecuada para clasificación multiclas, y el optimizador Adam con tasa de aprendizaje de 1e-4.
- Entrenamiento: En cada época, el modelo fue entrenado iterando por todos los batches del loader de entrenamiento. Se calculó la pérdida, se realizó backpropagation y se actualizó el modelo. También se acumuló la cantidad de aciertos para medir la precisión de entrenamiento.
- Validación: Tras cada época, el modelo fue evaluado en el conjunto de validación sin actualizar pesos, para medir su capacidad de generalización.
- Registro de métricas: Se almacenaron la pérdida y precisión en entrenamiento y la precisión en validación por cada época para monitoreo y análisis posterior.
- Dispositivo de cómputo: El proceso se ejecutó en GPU si estuvo disponible, o CPU en caso contrario, para optimizar tiempos de entrenamiento.

Resultados del entrenamiento



- Evolución de la pérdida (Loss): La pérdida de entrenamiento decrece de manera constante y rápida desde 1.6940 en la primera época hasta valores muy bajos cercanos a 0.02-0.03 en las últimas épocas. Esto indica que el modelo está aprendiendo correctamente y ajustando sus parámetros para minimizar el error en las predicciones.
- Precisión en entrenamiento (Train Acc): La precisión de entrenamiento comienza en 66.27% y sube rápidamente superando el 90% ya en la segunda época. Luego continúa incrementándose hasta valores muy altos (>99%) en las últimas épocas. Esto sugiere un ajuste muy fuerte al conjunto de entrenamiento.
- Precisión en validación (Val Acc): La precisión en validación inicia en 91.00%, mejora hasta alrededor del 94.00% cerca de la época 5, pero luego muestra pequeñas fluctuaciones entre 92% y 94% sin una mejora clara. Esto es típico cuando el modelo se ajusta al conjunto de entrenamiento, pero la capacidad de generalización se estabiliza.
- Sobreajuste potencial: La diferencia creciente entre la precisión de entrenamiento (muy cerca del 100%) y la precisión de validación (~92-94%) a partir de la época 6 indica un posible sobreajuste. El modelo está memorizando muy bien los datos de entrenamiento, pero la mejora en datos no vistos no es significativa.
- Velocidad de entrenamiento: Se observa una tasa estable de procesamiento de batches (~23-24 iteraciones por segundo), indicando un entrenamiento eficiente, utilizando únicamente CUDA por GPU.

Conclusiones

El modelo ResNet18 preentrenado es capaz de aprender rápidamente a clasificar correctamente las 70 razas del dataset. La precisión alcanzada en validación (~94%) es alta, mostrando un desempeño muy bueno en la tarea.

Puntos de mejora

- Se recomienda considerar técnicas para mitigar sobreajuste en entrenamientos futuros, como:

- Early stopping basado en la métrica de validación.
- Regularización adicional (dropout, weight decay).
- Aumento de datos más agresivo.
- Un análisis más detallado puede incluir la matriz de confusión para identificar clases con peor desempeño y ajustar la estrategia.

Modelo B (Custom CNN):

Metodología

Para complementar el enfoque basado en Transfer Learning, se diseñó y entrenó un modelo convolucional personalizado desde cero, utilizando las siguientes herramientas adicionales y técnicas:

- PyTorch (continuación): para definir la arquitectura propia de la red y controlar completamente el flujo de datos y operaciones.
- Batch Normalization (nn.BatchNorm2d): para estabilizar y acelerar el entrenamiento mediante normalización interna de las activaciones.
- Dropout (nn.Dropout): para reducir el sobreajuste durante el entrenamiento, aumentando la generalización del modelo.
- Transformaciones avanzadas de datos (Data Augmentation): se incluyeron, además del flip horizontal, rotaciones aleatorias de hasta 15 grados y ajustes de brillo y contraste (ColorJitter) para enriquecer la diversidad del dataset y robustecer el modelo frente a variaciones visuales.
- Optimizador Adam con tasa de aprendizaje más baja ($1e-5$), para un ajuste más fino y estable al entrenar una arquitectura desde cero.
- Uso intensivo de múltiples workers en DataLoader (num_workers=8) para acelerar la carga y preprocesamiento de datos.
- Entrenamiento y validación manual mediante funciones específicas que calculan pérdida y precisión, permitiendo mayor flexibilidad en la instrumentación y métricas.

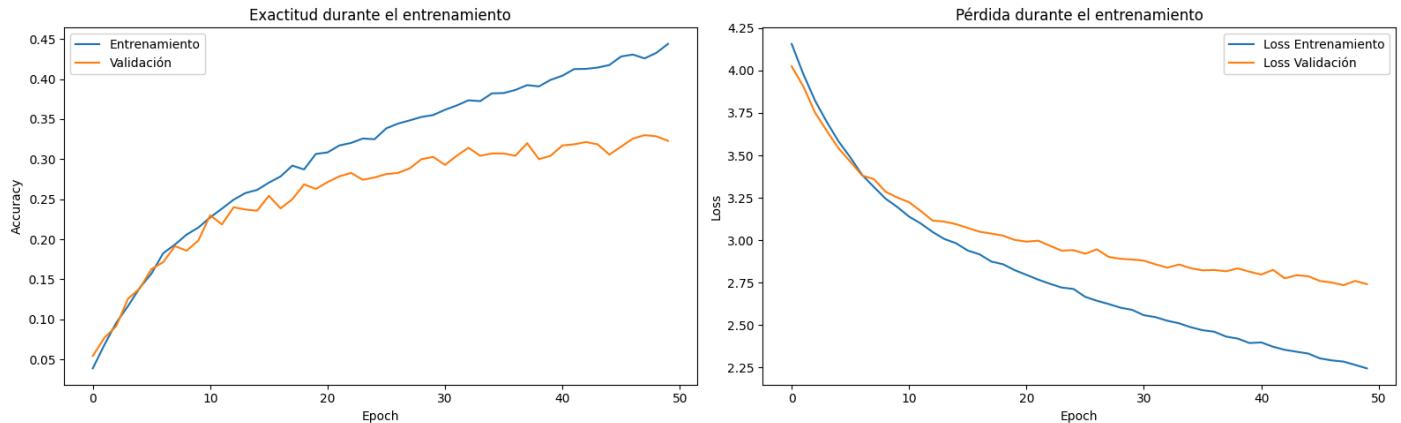
Implementación

La implementación del modelo convolucional personalizado incluyó los siguientes pasos:

- Definición de la arquitectura
Se diseñó una red CNN sencilla pero efectiva, compuesta por:
 - Dos capas convolucionales con filtros de tamaño 3x3 y padding para mantener dimensiones.
 - Normalización por batch (BatchNorm2d) después de cada convolución para mejorar la estabilidad del entrenamiento.
 - Funciones de activación ReLU y operaciones de max pooling para reducción progresiva de la dimensión espacial.
 - Capa de dropout con tasa 0.3 para mitigar el sobreajuste.

- Dos capas fully connected al final, con la última adaptada para clasificar en las 70 clases del dataset.
- Transformaciones y aumento de datos
El conjunto de entrenamiento fue enriquecido mediante:
 - Redimensionamiento a 224x224 píxeles.
 - Flip horizontal aleatorio.
 - Rotación aleatoria de hasta 15 grados.
 - Ajustes aleatorios de brillo y contraste.
 - Normalización con media y desviación estándar de ImageNet para compatibilidad con modelos preentrenados.
 - El conjunto de validación solo se redimensiona y normaliza.
- Carga de datos
Se utilizaron ImageFolder para estructurar los datos según carpetas por clase y DataLoader para cargar los datos en batches de 32, con ocho procesos en paralelo para mejorar el throughput.
- Configuración del entrenamiento
Se definieron la función de pérdida como CrossEntropyLoss y el optimizador Adam con una tasa de aprendizaje baja ($1e-5$), dada la complejidad de entrenar desde cero.
- Ciclo de entrenamiento y validación
Se implementaron funciones manuales para:
 - Entrenar el modelo por batch, calculando pérdida y exactitud en cada iteración, y actualizando los pesos.
 - Validar el desempeño en el conjunto de validación sin actualizar pesos, calculando pérdida y exactitud promedio.
- Ejecución del entrenamiento
El modelo se entrenó por 50 épocas, registrando y mostrando las métricas de pérdida y precisión tanto en entrenamiento como en validación por cada época, permitiendo monitorear la evolución del aprendizaje y detectar posibles problemas como sobreajuste o estancamiento.
- Dispositivo de cómputo
Todo el proceso se ejecutó en GPU.

Resultados del entrenamiento



Pérdida y precisión iniciales: En la primera época, la pérdida de entrenamiento es alta (4.15) y la precisión muy baja (3.9%). Esto es esperable ya que el modelo comienza sin conocimiento previo y debe aprender desde cero.

Progresión lenta pero constante: La pérdida disminuye gradualmente a lo largo de las 50 épocas, llegando a valores alrededor de 2.24. La precisión de entrenamiento mejora lentamente, alcanzando un 44.4% al final. Similarmente, la precisión en validación sube hasta aproximadamente 33%, y la pérdida de validación se mantiene en torno a 2.7-2.8.

Hay indicios claros de sobreajuste: La brecha entre precisión de entrenamiento y validación es moderada (~44% vs ~33%), pero crece de forma significativa luego de las 30 épocas, lo que indica que el modelo memoriza el test de entrenamiento.

Comparado a ResNet18: arquitecturas más sofisticadas aprovechan bloques residuales que permiten el entrenamiento efectivo de redes más profundas, mejorando significativamente la capacidad de abstracción sin caer en el problema del desvanecimiento del gradiente. Esto explica por qué, en este proyecto, ResNet18 logró un desempeño considerablemente superior, destacándose tanto en precisión como en estabilidad.

Desempeño general: El modelo no logra realizar predicciones precisas, incluso, por momentos parece ser azaroso. Esto es coherente dado que:

- La arquitectura es simple y poco profunda.
- El entrenamiento parte desde cero (sin pesos preentrenados).
- El dataset tiene muchas clases, pero pocas imágenes. Esto lo vuelve complejo.

Puntos de mejora

- Incrementar la complejidad del modelo (más capas, filtros, etc.) para captar mejor las características.
- Aumentar la tasa de aprendizaje inicial o usar técnicas de schedule para acelerar la convergencia.
- Aumentar la duración del entrenamiento.
- Experimentar con otras técnicas de aumento de datos o regularización.
- Incrementar el tamaño del dataset

Integración y Selección en la Aplicación:

Metodología

En esta etapa se integraron ambos modelos entrenados (ResNet18 con fine-tuning y modelo convolucional personalizado) para la generación de vectores de características (embeddings) en la aplicación de búsqueda por similitud.

Se implementaron funciones específicas para extraer embeddings de cada modelo, respetando sus arquitecturas particulares:

- Para ResNet18, se utilizó la salida de la penúltima capa fully connected (antes de la capa final de clasificación).
- Para el modelo personalizado, se diseñó un forward manual que detiene el flujo antes de la última capa fully connected, obteniendo un vector de características intermedio.

Para la búsqueda eficiente en espacios vectoriales, se utilizó FAISS, que permitió indexar los embeddings del conjunto de entrenamiento para ambos modelos y realizar consultas rápidas con búsqueda por distancia euclíadiana (L2).

Finalmente, se actualizó la interfaz gráfica con Gradio, permitiendo al usuario elegir dinámicamente qué modelo utilizar para la extracción de características y mostrando los resultados (imágenes similares) con sus correspondientes etiquetas.

Implementación

- Extracción de Embeddings

Se implementaron las funciones `extract_embeddings_resnet` y `extract_embeddings_custom` para procesar los datos mediante los respectivos modelos, obteniendo vectores que luego se guardaron en disco para uso posterior.

- Indexación con FAISS

Los embeddings fueron indexados con `faiss.IndexFlatL2`, generando índices separados para cada modelo y guardando estos índices para consultas rápidas.

- Búsqueda por similitud

Se desarrolló la función `search_similar` que, dada una consulta (embedding), retorna las etiquetas y rutas de las imágenes más similares según el índice seleccionado.

- Interfaz de usuario con Gradio

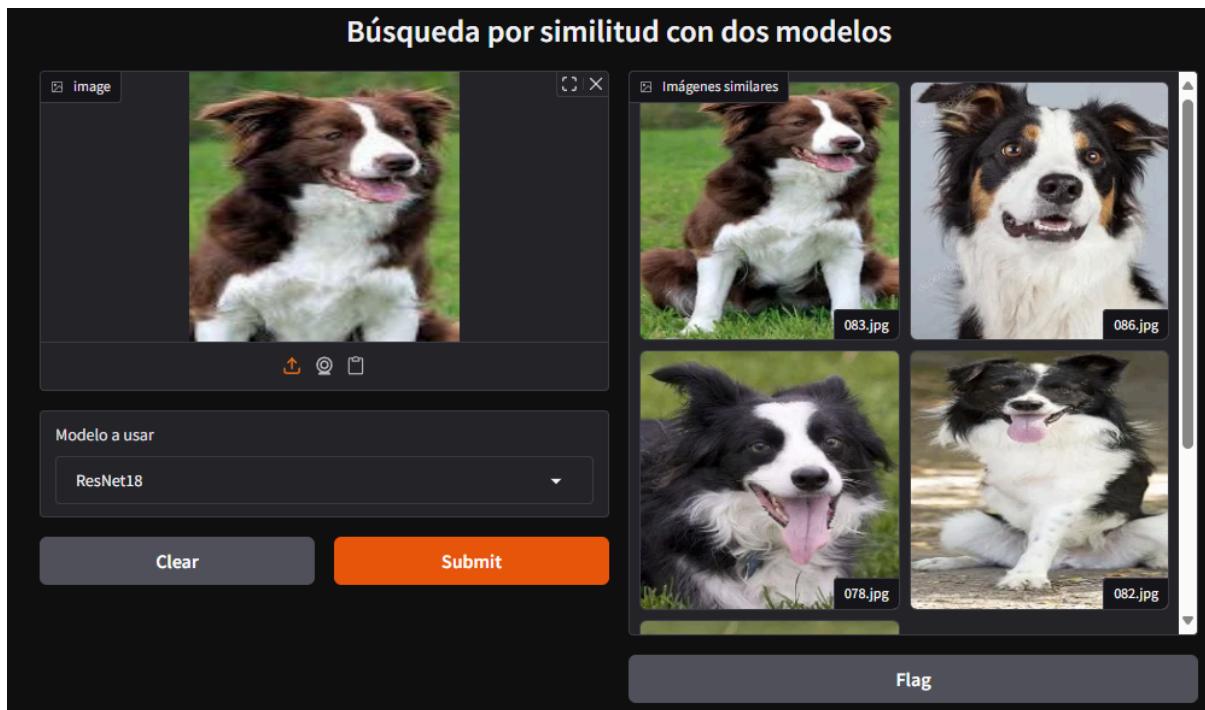
Se diseñó una interfaz intuitiva donde el usuario carga una imagen, selecciona el modelo (ResNet18 o CustomCNN) y obtiene la visualización de las imágenes similares encontradas en la base de datos. La interfaz utiliza la función `gradio_interface` para obtener embeddings y resultados.

- Manejo de datos y recursos

Se aplicaron transformaciones de imagen estándar para ambos modelos (resize, normalización) y se aseguró la compatibilidad con GPU cuando estuvo disponible para acelerar la inferencia.

Resultados

ResNet18:



El modelo fine-tuned basado en ResNet18 mostró un desempeño sobresaliente en la tarea de clasificación y generación de embeddings para la búsqueda por similitud. Su capacidad para captar características discriminativas profundas permitió identificar con alta precisión las distintas razas de perros, logrando resultados muy consistentes y robustos en las consultas.

Custom CNN:



El modelo convolucional personalizado evidenció un comportamiento diferente. Aunque logra captar ciertas características globales como patrones de pelaje y colores predominantes, su capacidad para distinguir específicamente la raza es limitada. Esto se refleja en una menor precisión y en la aparición de confusiones entre razas con características visuales similares.

Etapa 3: Detección de perros en imágenes complejas

Integrar un sistema de detección de objetos (YOLOv8) para localizar perros en escenas con múltiples objetos. Esta etapa se enfoca en obtener bounding boxes precisos que luego serán utilizados por los clasificadores entrenados en la etapa anterior.

Metodología

En esta etapa, el objetivo principal es localizar perros en imágenes del mundo real que pueden contener múltiples objetos y más de un perro. Para ello, se integró un sistema de detección de objetos basado en el modelo YOLOv8 preentrenado de Ultralytics. Este modelo es capaz de identificar la presencia de perros con precisión en imágenes complejas sin necesidad de entrenamiento adicional.

Implementación

- Se emplea el modelo YOLOv8n preentrenado (yolov8s.pt) para detectar perros en imágenes. Se ajusta un umbral de confianza (0.5) para filtrar detecciones poco seguras.
- Se define una clase DogDetector que encapsula la carga del modelo YOLO, la detección de bounding boxes de perros y el recorte de las regiones detectadas.
- Para la clasificación, se implementa la clase DogClassifier que soporta dos modelos: un ResNet18 fine-tuned y un modelo CNN personalizado desarrollado en etapas anteriores. Ambos modelos reciben las imágenes recortadas y normalizadas para devolver la predicción de raza.
- Se crea una función de visualización que dibuja los bounding boxes detectados junto con la etiqueta de la raza predicha sobre la imagen original, facilitando la interpretación visual de los resultados.
- Se realizan transformaciones estándar de preprocessamiento para adaptar las imágenes al input esperado por los clasificadores (redimensionado a 224x224, normalización con media y desviación estándar de ImageNet).
- Finalmente, se ejecuta el pipeline completo con una imagen de prueba que contiene múltiples perros, demostrando la detección y clasificación en conjunto.

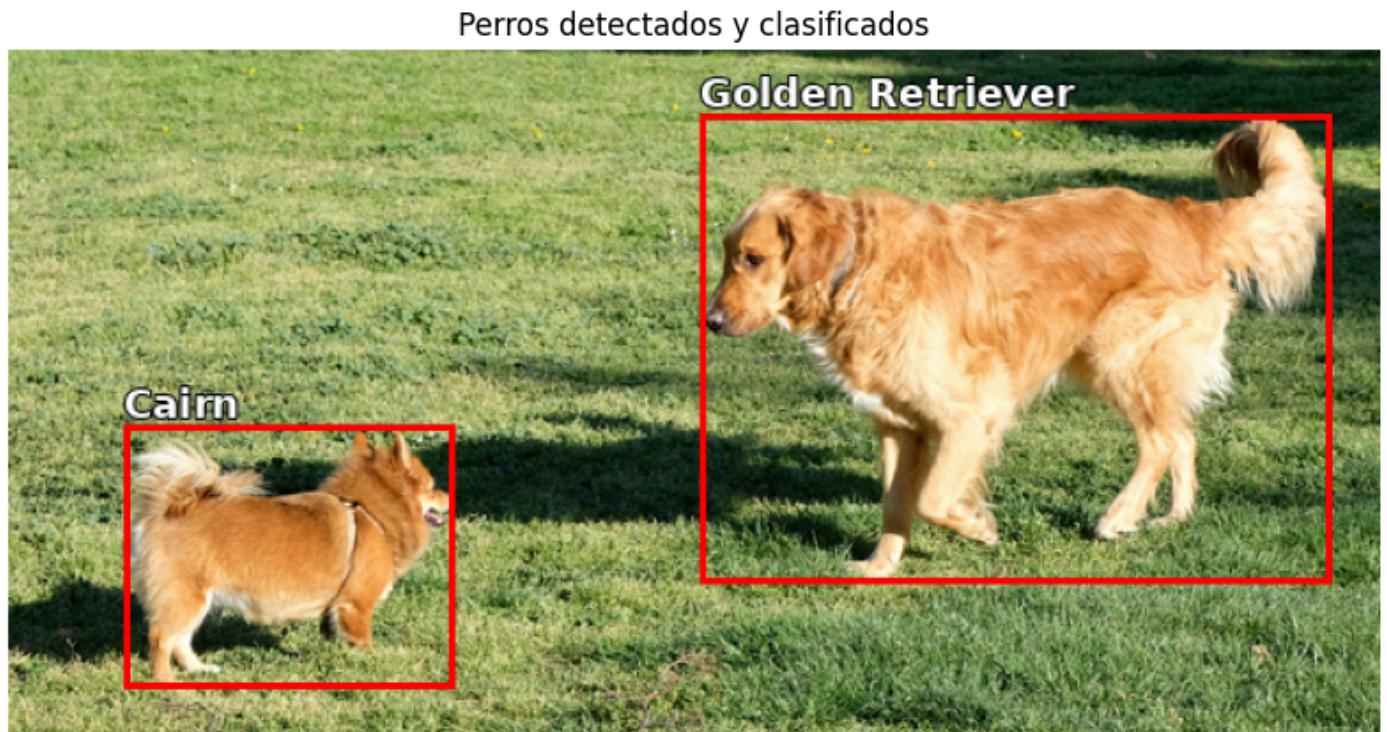
Resultados

YOLOv8:

El modelo YOLOv8 demostró una alta fiabilidad en la detección de perros en imágenes complejas, incluso cuando los perros están en posiciones difíciles, como de espaldas. Los bounding boxes generados están muy bien delimitados, capturando con precisión la ubicación y tamaño de cada perro en la escena. Esto confirma la capacidad del detector para manejar imágenes del mundo real con múltiples objetos y variabilidad en la pose y orientación de los perros.

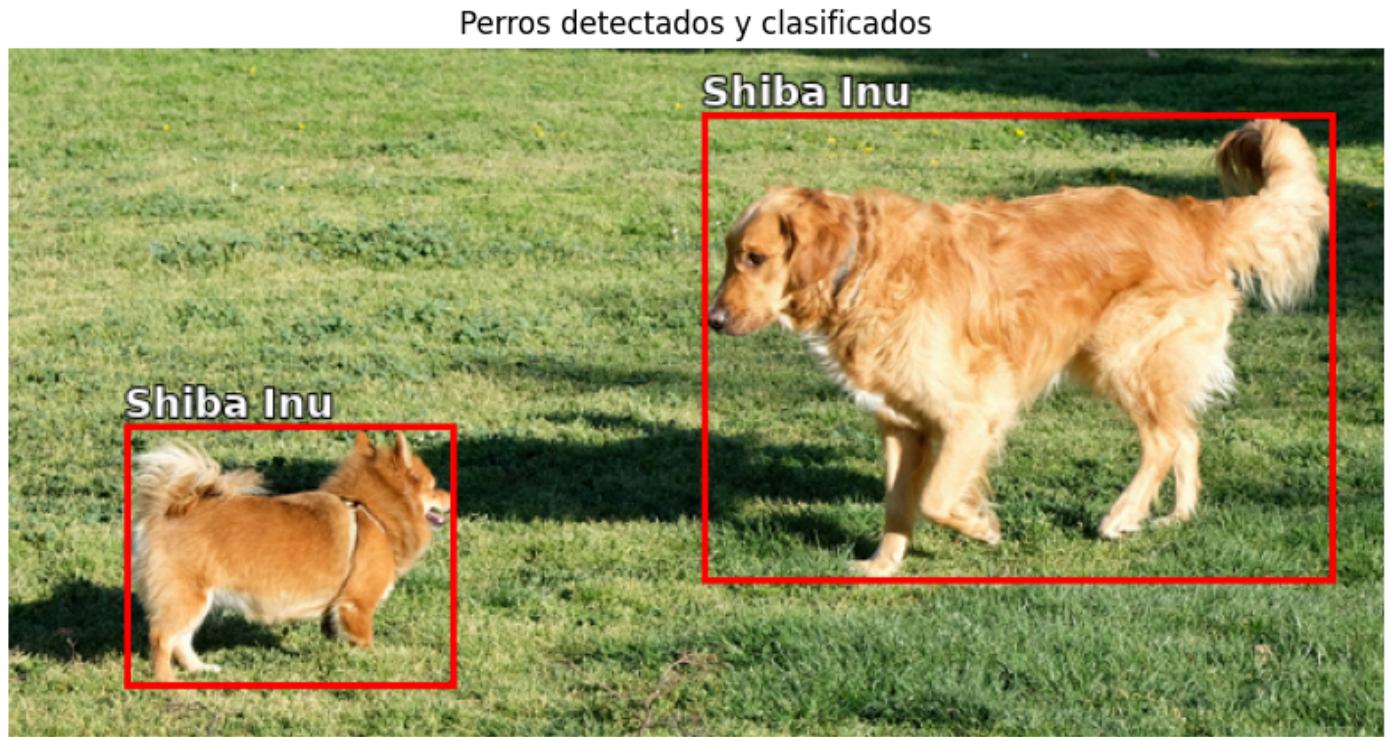
Esta robustez en la detección es fundamental para la siguiente etapa de clasificación, ya que proporciona recortes precisos que facilitan un mejor desempeño de los clasificadores entrenados.

ResNet18:



El detector identificó correctamente al perro ubicado a la derecha en la imagen y el clasificador ResNet18 logró predecir con alta confianza su raza. Sin embargo, presenta dificultades con el perro de la izquierda, que aparece de espaldas y parcialmente oculto. Esto generó confusión en el modelo, que probablemente lo interpretó como un perro de menor tamaño y buscó una raza similar a la apariencia observada, pero no la correcta.

Custom CNN:



Este modelo cometió errores en ambas detecciones, asignando la misma raza a los dos perros. Se asume que el clasificador basó su decisión principalmente en características globales como color y patrones de pelaje, sin lograr diferenciar correctamente entre las distintas razas presentes. Esta limitación indica que el modelo no pudo capturar con suficiente detalle las características discriminativas necesarias para una clasificación precisa en imágenes complejas.

Etapa 4: Pipeline completo de detección y clasificación automatizada

Implementar un pipeline que combine la detección de perros con la clasificación de su raza, generando anotaciones automáticas en imágenes complejas. Además, se desarrollan scripts para la exportación de resultados en formatos estándar (COCO y YOLO) y se evalúa el rendimiento del sistema sobre un conjunto de pruebas realista.

Metodología

Para evaluar el rendimiento final del sistema de detección y clasificación de razas de perro, se construyó un conjunto de pruebas compuesto por 10 imágenes complejas del mundo real, las cuales fueron guardadas dentro de la carpeta "test_complex_images". Estas imágenes incluyen múltiples objetos y perros en distintas poses y contextos. Por ejemplo:



Las imágenes fueron anotadas manualmente utilizando CVAT, generando un archivo en formato COCO JSON (instances_default.json) con las cajas delimitadoras y categorías correspondientes para cada perro detectado.

Se utilizó el pipeline completo desarrollado en etapas anteriores, compuesto por:

- YOLOv8s como detector de objetos, encargado de localizar los perros en la imagen.
- ResNet18 (o alternativamente, la Custom CNN) como clasificador de raza a partir de los recortes generados por el detector.

Implementación

El funcionamiento del pipeline se dividió en las siguientes etapas:

- Carga de anotaciones y test set:
Se cargaron las anotaciones manuales en formato COCO y se preparó una estructura que vincula cada imagen del conjunto de prueba con su identificador y bounding boxes esperados.
- Detección de perros con YOLOv8n:
Para cada imagen del conjunto de prueba, se ejecutó el detector YOLOv8n para localizar los perros presentes. Se filtraron las detecciones por clase (dog) y por un umbral de confianza mínimo. Se obtuvo una lista de coordenadas de las regiones donde se encontraron perros.
- Clasificación de cada detección:
Cada recorte generado por YOLO fue clasificado con uno de los modelos entrenados previamente (ResNet18 o CustomCNN). Se predijo la raza correspondiente para cada perro detectado.
- Conversión de predicciones al formato COCO:
Las detecciones generadas (coordenadas + clase predicha) fueron convertidas a formato COCO para poder ser comparadas directamente con las anotaciones reales mediante el evaluador oficial.
- Evaluación con pycocotools:
Se utilizó la herramienta COCOeval de pycocotools para calcular automáticamente todas las métricas relevantes. La evaluación se centró exclusivamente en detección de bounding boxes, considerando la clasificación correcta como parte del category_id.
- Comparación entre clasificadores:
El proceso de evaluación se repitió utilizando ambos clasificadores (ResNet18 y CustomCNN) para medir el impacto del clasificador en el rendimiento global del sistema.

Métricas y resultados

La evaluación del pipeline completo sobre el conjunto de prueba arrojó los siguientes valores:

- mAP (mean Average Precision): 0.2787
- Precision @ IoU=0.5: 0.3131
- Recall: 0.2781
- AP_medium: 0.0000
- AP_large: 0.3332

Estos resultados reflejan un desempeño moderado, pero razonable considerando la dificultad del escenario: imágenes del mundo real con múltiples objetos, ángulos no ideales y posibles occlusiones.

La métrica mAP (0.2787) indica que el sistema logra un nivel aceptable de precisión promedio en diferentes umbrales de intersección (IoU). La precisión específica a IoU=0.5 fue algo mayor (0.3131), lo

que sugiere que el detector es capaz de acertar razonablemente bien en la ubicación general del objeto, aunque puede fallar en la exactitud fina del bounding box.

El valor de Recall (0.2781) revela que el sistema detectó un poco más de un cuarto de los perros presentes en el conjunto, lo cual indica que aún hay margen de mejora, especialmente en escenarios con perros parcialmente visibles o de espaldas.

Por otro lado, el AP_medium fue 0.0000, lo que indica que no se logró detectar correctamente perros de tamaño medio en las imágenes evaluadas. En contraste, el AP_large alcanzó 0.3332, mostrando que el sistema responde mejor ante perros de mayor tamaño en la imagen, lo cual es coherente con los resultados visuales observados.

En conjunto, estos resultados muestran que el pipeline es funcional, especialmente en condiciones más favorables (perros grandes y bien visibles), pero que necesita ajustes para mejorar su sensibilidad ante perros medianos o en posiciones más complejas.

Optimización de Modelos (ONNX)

Metodología

Para reducir los tiempos de inferencia del pipeline sin comprometer significativamente la precisión, se aplicó una técnica de exportación y optimización del modelo mediante ONNX y ONNX Runtime. Esta opción fue elegida por su compatibilidad con PyTorch, su facilidad de implementación y su capacidad para integrarse con aceleradores como TensorRT.

El proceso consistió en:

- Exportar el mejor modelo de clasificación (ResNet18 entrenado con fine-tuning) al formato ONNX (Open Neural Network Exchange).
- Validar la integridad del modelo exportado mediante las herramientas oficiales (onnx.checker).
- Ejecuta inferencias usando ONNX Runtime, una herramienta ligera y optimizada para desplegar modelos en diversos entornos (CPU, GPU, TensorRT, etc.).
- Comparar el tiempo de inferencia y la precisión del modelo ONNX con respecto al modelo original en PyTorch.

Implementación

- Conversión del modelo a ONNX:
Se utilizó un dummy input con el tamaño adecuado ($1 \times 3 \times 224 \times 224$) para exportar la red ResNet18 entrenada en PyTorch. Se incluyeron nombres de entrada/salida y ejes dinámicos para admitir distintos tamaños de batch.
- Validación del modelo ONNX:
El modelo exportado se verificó utilizando la función onnx.checker.check_model() para garantizar que cumple con las especificaciones y no presenta errores estructurales.
- Carga del modelo con ONNX Runtime:
El modelo fue ejecutado usando onnxruntime.InferenceSession, empleando la CPU como dispositivo de inferencia. Se replicó el mismo proceso de transformación aplicado durante el entrenamiento (redimensionamiento, normalización) para garantizar consistencia.
- Comparación de resultados:
Se evaluó la salida del modelo ONNX sobre imágenes reales y se compararon las predicciones con

las del modelo en PyTorch para asegurar que la conversión no alteró el comportamiento del clasificador.

Resultados

Luego de exportar el modelo entrenado a ONNX y ejecutar inferencias con ONNX Runtime, se comparó el rendimiento frente a la versión original en PyTorch. La evaluación se realizó sobre una imagen de la clase Afghan, y los resultados fueron los siguientes:

- PyTorch Predicción: Afghan
Tiempo de inferencia: 20.24 ms
- ONNX Runtime Predicción: Afghan
Tiempo de inferencia: 7.50 ms

Las predicciones coinciden, indicando que el modelo ONNX conserva la precisión del modelo original.

Esta diferencia de tiempo representa una mejora significativa en la velocidad de inferencia, logrando reducirla a aproximadamente un tercio del tiempo original. Esto valida que la exportación a ONNX no sólo acelera la inferencia, sino que además preserva la exactitud del modelo, lo cual es fundamental para mantener la confiabilidad del sistema en aplicaciones reales.

Script de Anotación Automática:

Metodología

Para esta etapa se desarrolló un script que automatiza completamente el proceso de anotación utilizando el pipeline previamente implementado (detección + clasificación). Las herramientas clave utilizadas fueron:

- YOLOv8 (Ultralytics) para la detección de perros.
- ResNet18 finetuneado para la clasificación de razas.
- PIL para manipulación de imágenes.
- pandas para la gestión del conjunto de imágenes.
- json para la generación del formato COCO.
- Sistema de archivos (os, pathlib) para recorrer carpetas y generar las salidas requeridas.

Implementación

El pipeline automatizado sigue los siguientes pasos:

- Carga de imágenes desde una carpeta (test/) definida en el CSV dogs.csv, asegurando que todas existan y pertenezcan al conjunto de prueba.
- Inicialización del detector YOLO y del clasificador ResNet18 entrenado con 70 razas.
- Procesamiento por imagen:
 - Se obtienen las predicciones de bounding boxes de perros usando YOLO.
 - Cada recorte correspondiente a un perro es clasificado usando el modelo ResNet18.

- Generación de salida YOLOv5:
 - Se genera un archivo .txt por imagen.
 - Cada línea contiene: class_id x_center y_center width height, con coordenadas normalizadas.
- Generación de salida COCO:
 - Se agrega una entrada en el diccionario JSON para cada imagen y para cada objeto detectado (anotación).
 - Cada anotación incluye: image_id, category_id, bbox, area, y iscrowd.

Conclusiones

El proyecto logró cumplir satisfactoriamente su objetivo de construir un sistema integral de clasificación de razas de perros, abarcando todas las etapas clave: entrenamiento de modelos, detección de objetos, búsqueda por similitud visual y generación automática de anotaciones en formatos estándar.

A lo largo del desarrollo se demostró que los modelos basados en transfer learning, como ResNet18, superan ampliamente en rendimiento a arquitecturas desarrolladas desde cero, especialmente en tareas complejas como clasificación fina de razas. Asimismo, la combinación de herramientas (FAISS para recuperación por similitud, YOLOv8 para detección eficiente, Gradio para visualización interactiva, y ONNX para optimización) permitió construir soluciones escalables, robustas y con un alto potencial de aplicación real.

Durante el desarrollo del trabajo se enfrentaron diversos desafíos técnicos y conceptuales. Entre los más relevantes se destacan:

- **Limitaciones de la CNN personalizada:** El modelo desarrollado desde cero mostró una capacidad limitada para aprender representaciones visuales complejas. A pesar de probar diversas variantes arquitectónicas, las mejoras en precisión fueron marginales, lo que reflejó claramente las restricciones inherentes a redes poco profundas y sin preentrenamiento. Esta experiencia reforzó la importancia de utilizar modelos más sofisticados, como ResNet, que incorporan mecanismos como las capas residuales para facilitar el aprendizaje profundo.
- **Evaluación en escenarios del mundo real:** Se diseñaron a propósito imágenes de prueba con escenas desafiantes (perros parcialmente ocultos, poses inusuales, fondos cargados y presencia de otros objetos) con el objetivo de exigir al máximo al pipeline. A pesar de la complejidad, el sistema logró detectar y clasificar correctamente en varios casos, lo que evidencia una buena capacidad de generalización en situaciones reales.

El sistema final fue evaluado sobre imágenes reales y complejas, demostrando una buena capacidad de generalización y confirmando que los productos desarrollados no sólo son técnicamente correctos, sino también funcionales y útiles. En este sentido, el trabajo no solo constituye un ejercicio académico completo, sino también una base sólida para aplicaciones reales en áreas como adopción de mascotas, sistemas veterinarios inteligentes, búsqueda visual por similitud y anotación asistida de datasets.

Puntos de mejora

Si bien los resultados fueron satisfactorios, se identifican oportunidades de mejora que permitirían aumentar la robustez y escalabilidad del sistema:

- Mejorar la calidad y diversidad del dataset, para representar más variaciones reales y evitar sesgos de entrenamiento.
- Refinar la arquitectura de los modelos personalizados, para acercarse al rendimiento de modelos preentrenados.
- Optimizar la integración entre las distintas etapas del pipeline, especialmente en los casos límite o cuando hay múltiples objetos por imagen.
- Incrementar la interpretabilidad del sistema, para entender mejor las decisiones del modelo ante razas visualmente similares.
- Explorar versiones más livianas y eficientes, orientadas a implementación en dispositivos móviles o en tiempo real.