Informe Final del Proyecto: Sistema Integral de Detección, Clasificación y Búsqueda de Razas de Perros

Materia: Visión por Computadora - TUIA 2025 **Alumno:** Roberto Orazi **Fecha:** Diciembre 2024

Tabla de Contenidos

---

1. Resumen Ejecutivo

Este documento presenta el desarrollo completo de un sistema de visión artificial diseñado para identificar razas de perros. El proyecto abarca desde la ingesta de datos crudos hasta el despliegue de una aplicación funcional, pasando por etapas críticas de entrenamiento de modelos profundos (Deep Learning), integración de sistemas de detección de objetos y optimización para la inferencia.

El sistema final combina la potencia de **YOLOv8** para la localización de objetos con la precisión de una **ResNet18** ajustada mediante *Transfer Learning* para la clasificación de 70 razas distintas. Además, se implementó un motor de búsqueda por similitud visual utilizando **FAISS**, capaz de recuperar imágenes semánticamente cercanas en milisegundos.

Los resultados demuestran la viabilidad técnica de la solución, alcanzando un **NDCG@10 de 0.9874** en tareas de búsqueda y un **F1-Score de 0.9483** en clasificación, con una arquitectura optimizada que reduce el consumo de memoria en un 75% mediante técnicas de cuantización dinámica.

---

2. Introducción y Contexto del Problema

2.1 El Desafío de la Clasificación Fine-Grained

La clasificación de imágenes genéricas (ej. "gato" vs "perro" vs "avión") es un problema largamente resuelto en la visión por computadora. Sin embargo, la clasificación "fine-grained" (de grano fino) presenta desafíos únicos. En nuestro caso, distinguir entre 70 razas de perros implica manejar dos dificultades estadísticas fundamentales:

1. **Baja Varianza Inter-clase:** Muchas razas son fenotípicamente similares. Por ejemplo, distinguir un *Siberian Husky* de un *Alaskan Malamute*, o un *American Hairless* de un *Mexican Hairless*, requiere que el modelo aprenda características extremadamente sutiles (forma de las orejas, estructura del hocico, textura de la cola) que no son evidentes en un modelo genérico.
2. **Alta Varianza Intra-clase:** Los perros de una misma raza pueden tener apariencias muy diversas debido a la edad, color del pelaje, condiciones de iluminación, pose y oclusión. Un *Labrador Retriever* puede ser negro, chocolate o amarillo; puede estar corriendo, sentado o nadando.

El sistema debe ser robusto a estas variaciones intra-clase mientras mantiene la sensibilidad necesaria para discriminar las sutiles diferencias inter-clase.

2.2 Objetivos del Proyecto

El proyecto se estructuró en cuatro etapas incrementales, cada una con objetivos técnicos específicos:

1. **Etapa 1 (Exploración y Retrieval):** Implementar un sistema de búsqueda visual que, dada una imagen de consulta (*query*), recupere las imágenes más similares del dataset. Esto permite validar la calidad de los datos y la capacidad de las redes pre-entrenadas para extraer características relevantes sin re-entrenamiento.

2. **Etapa 2 (Especialización):** Entrenar un clasificador discriminativo. Se busca superar el desempeño de los modelos genéricos mediante técnicas de *Transfer Learning* y *Fine-Tuning*, adaptando una red neuronal a nuestro dominio específico.
3. **Etapa 3 (Integración):** Construir un pipeline *end-to-end* que resuelva el problema del mundo real: las fotos de usuarios no son recortes perfectos de perros. El sistema debe primero localizar al animal en la escena y luego clasificarlo.
4. **Etapa 4 (Producción):** Validar el sistema con datos "in-the-wild" anotados manualmente y optimizar los modelos para su despliegue, reduciendo latencia y tamaño en disco.

---

3. Marco Teórico Fundamentado

Antes de profundizar en la implementación, es crucial establecer las bases teóricas que sustentan las decisiones de diseño.

3.1 Redes Neuronales Convolucionales (CNN)

Las CNN (*Convolutional Neural Networks*) son la arquitectura dominante en visión artificial. Se inspiran en la organización del córtex visual biológico, donde las neuronas responden a estímulos en regiones limitadas del campo visual (campos receptivos).

Operación de Convolución: Matemáticamente, una convolución discreta en 2D se define como: $$ (I * K)(i, j) = \sum_m \sum_n I(i+m, j+n) K(m, n) $$ Donde I es la imagen de entrada y K es el núcleo o filtro. Esta operación permite a la red aprender características:

- **Locales:** Detecta patrones en vecindarios pequeños de píxeles.
- **Invariantes a la Traslación:** Si una característica (ej. una oreja) se mueve en la imagen, el mapa de características resultante simplemente se desplaza, pero la activación se mantiene.

Capas de Pooling: El *submuestreo* (usualmente Max Pooling) reduce progresivamente la resolución espacial de los mapas de características.

- Reduce la cantidad de parámetros y el costo computacional.
- Introduce invariancia a pequeñas distorsiones y traslaciones.
- Aumenta el campo receptivo de las neuronas en capas posteriores, permitiéndoles "ver" una porción más grande de la imagen original.

3.2 Arquitecturas Profundas: ResNet

A medida que las redes neuronales se hacen más profundas, surge el problema del **Desvanecimiento del Gradiente** (*Vanishing Gradient*). Durante la retropropagación (*Backpropagation*), el gradiente de la función de pérdida se multiplica repetidamente por la derivada de la función de activación. Si estas derivadas son pequeñas (< 1), el gradiente tiende a cero exponencialmente al llegar a las primeras capas, impidiendo que estas aprendan.

La Solución Residual: La arquitectura ResNet (*Residual Network*) introduce conexiones de salto (*skip connections*) o atajos. En lugar de aprender un mapeo directo $H(x)$, las capas residuales aprenden un mapeo residual $F(x) := H(x) - x$. La salida original se suma a la salida de las capas apiladas: $$ y = F(x, {W_i}) + x $$ Esto tiene dos efectos profundos:

1. Si la función óptima es la identidad (x), es más fácil para la red llevar los pesos de $F(x)$ a cero que aprender la identidad desde cero.
2. Durante la retropropagación, el gradiente puede fluir directamente a través del atajo $+x$ sin atenuarse, permitiendo entrenar redes de cientos de capas (como ResNet50 o ResNet152).

3.3 Transfer Learning y Fine-Tuning

El entrenamiento de una CNN profunda desde cero requiere millones de imágenes etiquetadas y semanas de cómputo en GPUs. **Transfer Learning** permite reutilizar el conocimiento adquirido por una red en una tarea masiva (como ImageNet, con 1.2 millones de imágenes y 1000 clases) para una nueva tarea con menos datos.

Existen dos estrategias principales que utilizamos:

1. **Feature Extraction (Etapa 1):** Utilizamos la red pre-entrenada como un extractor de características fijo. Congelamos todos los pesos de la red y solo utilizamos la salida de la penúltima capa (antes de la clasificación) como un vector descriptor de la imagen.
2. **Fine-Tuning (Etapa 2):** Inicializamos la red con los pesos pre-entrenados, pero "descongelamos" algunas o todas las capas. Entrenamos la red con una tasa de aprendizaje (learning rate) muy baja. Esto permite ajustar los pesos para que los filtros aprendan características específicas de nuestro dominio (texturas de pelo de perro) en lugar de características genéricas de ImageNet.

3.4 Detección de Objetos en Tiempo Real (YOLO)

Mientras que la clasificación asigna una etiqueta a una imagen completa, la detección implica localizar objetos y asignarles etiquetas.

YOLO (You Only Look Once): YOLO revolucionó el campo al plantear la detección como un único problema de regresión, a diferencia de enfoques anteriores (R-CNN) que requerían múltiples etapas (propuesta de regiones + clasificación).

- **Grid Cell:** La imagen se divide en una grilla $S \times S$. Si el centro de un objeto cae en una celda, esa celda es responsable de detectarlo.
- **Bounding Box Regression:** Cada celda predice B cajas delimitadoras y sus puntuaciones de confianza. Una caja se define por $(x, y, w, h, confidence)$.
- **Anchor-Free (YOLOv8):** A diferencia de versiones anteriores que usaban cajas de anclaje predefinidas (*anchor boxes*), YOLOv8 predice directamente el centro de los objetos, reduciendo la complejidad de diseño y mejorando la generalización a formas de objetos inusuales.
- **Loss Function:** YOLO utiliza una función de pérdida compuesta que penaliza errores en la localización de la caja (IoU Loss), errores en la clase del objeto (Classification Loss) y errores en la "objectness" (si hay objeto o no).

3.5 Búsqueda Vectorial y Embeddings

Un **Embedding** es una representación vectorial de baja dimensión de un dato de alta dimensión (como una imagen). La idea central es que imágenes semánticamente similares deben mapearse a puntos cercanos en el espacio vectorial.

- **Espacio Vectorial:** Si extraemos el vector de características de una ResNet50, obtenemos un vector en $\mathbb{R}^{2048}$.
- **Distancia L2 (Euclidiana):** La similitud entre dos imágenes A y B se mide generalmente como la distancia Euclidiana entre sus embeddings u y v: $$ d(u, v) = ||u - v||_2 = \sqrt{\sum_{i=1}^{n} (u_i - v_i)^2} $$
- **Similitud Coseno:** Alternativamente, si normalizamos los vectores para que tengan norma 1, la distancia Euclidiana es equivalente a medir el ángulo entre ellos (Similitud Coseno).

FAISS (Facebook AI Similarity Search): Buscar el vecino más cercano en un dataset de millones de vectores usando fuerza bruta es computacionalmente inviable. FAISS implementa estructuras de datos eficientes (como índices invertidos y cuantización de producto) para acelerar estas búsquedas en órdenes de magnitud.

3.6 Métricas de Evaluación en Visión

Para cuantificar el desempeño, utilizamos métricas estándar rigurosas:

- **IoU (Intersection over Union):** Mide el solapamiento entre la caja predicha (P) y la caja real (G). $$ IoU = \frac{Area(P \cap G)}{Area(P \cup G)} $$ Un IoU > 0.5 generalmente se considera una detección correcta.
- **Precision (Precisión):** De todas las detecciones positivas del modelo, ¿cuántas son reales? $$ Precision = \frac{TP}{TP + FP} $$
- **Recall (Sensibilidad):** De todos los objetos reales en la escena, ¿cuántos encontró el modelo? $$ Recall = \frac{TP}{TP + FN} $$
- **F1-Score:** La media armónica entre precisión y recall. Proporciona una métrica única balanceada. $$ F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} $$
- **mAP (Mean Average Precision):** Es el estándar de oro en detección. Se calcula el área bajo la curva Precision-Recall para cada clase y luego se promedia entre todas las clases. mAP@0.5 significa que se calcula considerando correctas las detecciones con IoU > 0.5.
- **NDCG (Normalized Discounted Cumulative Gain):** Usada en recuperación de información. Valora que los resultados relevantes aparezcan en las primeras posiciones del ranking.

---

4. Ingeniería de Software y Arquitectura Modular

Uno de los pilares del proyecto fue el diseño de software. Se evitó la creación de "scripts espagueti" monolíticos en favor de una arquitectura modular basada en el principio DRY (*Don't Repeat Yourself*).

4.1 Estructura del Repositorio

El proyecto se organiza en directorios funcionales:

```
tp-cv-orazi/
├── dataset/                # Datos crudos divididos en train, valid, test
├── utils/                  # Paquete Python con lógica compartida
│   ├── __init__.py         # Inicializador del paquete
│   ├── dataset.py          # Clase DogDataset
│   ├── models.py           # Constructores de modelos (Factory)
│   └── transforms.py       # Pipelines de preprocesamiento
├── etapa-1/                # Scripts de Búsqueda Visual
│   ├── extraer_embeddings.py
│   ├── evaluar_ndcg.py
│   └── app_etapa1.py       # Aplicación Demo
├── etapa-2/                # Scripts de Entrenamiento
│   ├── entrenar_resnet18.py
│   └── resnet18_best.pth   # Checkpoint del modelo
├── etapa-3/                # Pipeline Integrado
│   ├── app_deteccion.py
│   └── metricas_pipeline.csv
├── etapa-4/                # MLOps y Finalización
│   ├── etiquetador_manual.py # Herramienta custom
│   ├── optimizar_modelos.py
│   └── evaluar_pipeline.py
├── informe/                # Documentación
└── requirements.txt        # Dependencias
```

4.2 Análisis del Módulo `utils`

Este módulo centraliza el código crítico para asegurar consistencia entre las etapas de entrenamiento, evaluación e inferencia.

`utils/dataset.py`: Ingesta de Datos Robusta

Aquí se define la clase `DogDataset`, que hereda de `torch.utils.data.Dataset`.

```
class DogDataset(Dataset):
    def __init__(self, csv_file, root_dir, dataset_type=None, transform=None):
        # ... carga de dataframe con pandas ...
        # Construcción dinámica y determinística del mapeo de etiquetas
        self.label_to_idx = {
            label: idx for idx, label in enumerate(sorted(df["labels"].unique()))
        }
```

Análisis de Diseño:

- **Determinismo:** El uso de `sorted()` al crear `label_to_idx` es crítico. Asegura que la raza "Afghan Hound" siempre sea el ID 0, independientemente de cómo se carguen los datos. Sin esto, un modelo entrenado hoy sería incompatible con un script de inferencia mañana si el orden de carga cambia.
- **Filtrado de Subsets:** El constructor acepta un argumento `dataset_type` ('train', 'valid', 'test') para filtrar el CSV maestro sin necesidad de dividir físicamente los archivos en disco.

```
    def __getitem__(self, idx):
        # ... obtención de rutas ...
        image = Image.open(img_path).convert("RGB")
        # ... transformaciones ...
```

- **Conversión Explicita:** La llamada a `.convert("RGB")` es una defensa contra datos corruptos o inesperados. Algunas imágenes en datasets públicos pueden ser monocromáticas (1 canal) o CMYK (4 canales). Si pasamos estas imágenes directamente a una CNN que espera 3 canales (RGB), el programa fallaría en tiempo de ejecución. Esta línea garantiza la integridad del tensor de entrada.

`utils/transforms.py` : Preprocesamiento de Imágenes

Define las políticas de transformación de datos.

Transformación de Entrenamiento (`get_train_transform`): Implementa *Data Augmentation* agresivo para combatir el overfitting.

1. `RandomResizedCrop(224)` : Recorta una porción aleatoria de la imagen y la redimensiona. Esto simula zoom y diferentes encuadres, obligando a la red a no depender de ver al perro completo.
2. `RandomHorizontalFlip()` : Duplica efectivamente el tamaño del dataset invirtiendo las imágenes horizontalmente.
3. `ColorJitter(...)` : Altera brillo, contraste y saturación. Hace al modelo robusto a cambios de iluminación.
4. `Normalize(...)` : Resta la media y divide por la desviación estándar de ImageNet `mean=[0.485, ...]` . Esto centra los datos en torno a cero, lo cual es matemáticamente necesario para la convergencia estable del Descenso de Gradiente Estocástico.

Transformación de Validación (`get_val_transform`): Debe ser determinística.

1. `Resize(256)` : Escala la imagen.
2. `CenterCrop(224)` : Toma el centro exacto. Esto es el estándar en benchmarks de ImageNet para evaluación justa.

`utils/models.py` : Factory de Arquitecturas

Abstrae la complejidad de instanciar y modificar modelos pre-entrenados.

```
def get_resnet18(num_classes, pretrained=True):
    # Carga la arquitectura base
    model = models.resnet18(weights=...)
    # Reemplaza la 'cabeza' de clasificación
    num_features = model.fc.in_features  # 512 para ResNet18
    model.fc = nn.Linear(num_features, num_classes) # Nueva capa 512 -> 70
    return model
```

Esta función encapsula la lógica de *surgery* (cirugía) de la red, donde amputamos la última capa original (que clasifica 1000 clases) e injertamos una nueva capa no entrenada para nuestras 70 razas.

Capítulo I: Búsqueda de Imágenes por Similitud (Etapa 1)

Objetivo

Crear una aplicación que, dada una imagen de un perro, encuentre las imágenes más similares en el dataset y prediga la raza mediante voto mayoritario.

---

1. Imports y Configuración del Dispositivo

```
import torch
import torch.nn as nn
from torchvision import models, transforms
from torch.utils.data import Dataset, DataLoader
import numpy as np
import pandas as pd
from PIL import Image
from pathlib import Path
import faiss
from collections import Counter
import gradio as gr
```

¿Por qué estos imports?

- `torch` y `torchvision` : Framework de deep learning para cargar modelos pre-entrenados
- `numpy` y `pandas` : Manipulación eficiente de datos y arrays
- `PIL` : Carga y procesamiento de imágenes
- `faiss` : Librería de Facebook para búsquedas eficientes de similitud en vectores de alta dimensión
- `gradio` : Crea interfaces web interactivas de forma simple
- `Counter` : Cuenta frecuencias para el voto mayoritario

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

¿Por qué? Detecta si hay GPU disponible para acelerar el procesamiento. En Mac M1 usará CPU o MPS si está configurado.

---

2. Dataset Personalizado

```
class DogDataset(Dataset):
    def __init__(self, csv_file, root_dir, transform=None):
        self.data = pd.read_csv(csv_file)
        self.root_dir = Path(root_dir)
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        img_path = self.root_dir / self.data.iloc[idx]['filepaths']
        image = Image.open(img_path).convert('RGB')
        label = self.data.iloc[idx]['labels']

        if self.transform:
            image = self.transform(image)

        return image, label, str(img_path)
```

¿Por qué crear esta clase?

- PyTorch necesita un formato específico para cargar datos eficientemente
- Permite procesar imágenes en lotes (batches) para acelerar la extracción
- `__getitem__` carga una imagen, la transforma y devuelve la imagen, su etiqueta (raza) y la ruta
- `convert('RGB')` asegura que todas las imágenes tengan 3 canales (algunas pueden ser escala de grises)

---

3. Transformaciones de Imagen

```
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

¿Por qué estas transformaciones?

- `Resize(256)` : Redimensiona la imagen manteniendo el aspect ratio
- `CenterCrop(224)` : Recorta el centro a 224x224, el tamaño que espera ResNet50
- `ToTensor()` : Convierte la imagen PIL a un tensor de PyTorch con valores [0,1]
- `Normalize()` : Normaliza con la media y desviación estándar de ImageNet (dataset con el que se entrenó ResNet50). Esto es crítico para que el modelo funcione correctamente.

---

4. Carga del Dataset

```
dataset = DogDataset('dataset/dogs.csv', 'dataset', transform=transform)
dataloader = DataLoader(dataset, batch_size=32, shuffle=False, num_workers=0)
```

¿Por qué?

- `DataLoader` carga imágenes en lotes de 32 para procesar múltiples imágenes simultáneamente
- `shuffle=False` : No mezclamos porque necesitamos mantener el orden para asociar embeddings con rutas
- `num_workers=0` : En Mac M1 a veces hay problemas con multiprocessing, por eso usamos 0

---

5. Modelo de Extracción de Características

```
model = models.resnet50(weights=models.ResNet50_Weights.IMAGENET1K_V1)
model = nn.Sequential(*list(model.children())[:-1])
model = model.to(device)
model.eval()
```

¿Por qué ResNet50?

- Es un modelo pre-entrenado en ImageNet (1.4M imágenes, 1000 clases)
- Ya aprendió a reconocer patrones visuales complejos
- Removemos la última capa (clasificación) porque solo queremos los embeddings (representación vectorial)
- El resultado son vectores de 2048 dimensiones que capturan las características visuales de cada imagen
- `model.eval()` : Desactiva dropout y batch normalization para inferencia

---

6. Extracción de Embeddings

```
embeddings = []
labels = []
image_paths = []

with torch.no_grad():
    for images, lbls, paths in dataloader:
        images = images.to(device)
        features = model(images)
        features = features.squeeze().cpu().numpy()

        if len(features.shape) == 1:
            features = features.reshape(1, -1)

        embeddings.append(features)
        labels.extend(lbls)
        image_paths.extend(paths)

embeddings = np.vstack(embeddings)
```

¿Por qué este proceso?

- `torch.no_grad()` : Desactiva el cálculo de gradientes (no estamos entrenando), ahorra memoria
- Procesamos las ~9300 imágenes en lotes de 32
- `squeeze()` : Elimina dimensiones innecesarias (de [32,2048,1,1] a [32,2048])
- `cpu().numpy()` : Convierte de tensor GPU a array NumPy en CPU
- Guardamos los embeddings, las razas y las rutas para poder recuperar las imágenes después
- `np.vstack()` : Apila todos los batches en una matriz (9300, 2048)

---

7. Creación del Índice FAISS

```
faiss.normalize_L2(embeddings)

dimension = embeddings.shape[1]
index = faiss.IndexFlatIP(dimension)
index.add(embeddings)
```

¿Por qué FAISS?

- FAISS (Facebook AI Similarity Search) es extremadamente eficiente para búsquedas en millones de vectores
- `normalize_L2()` : Normaliza los vectores a longitud 1, convirtiendo producto interno en similitud coseno
- `IndexFlatIP` : Índice que usa producto interno (Inner Product) para medir similitud
- Con vectores normalizados, producto interno = similitud coseno (más cercano a 1 = más similar)
- `index.add()` : Agrega todos los vectores al índice

¿Por qué similitud coseno? Mide el ángulo entre vectores, ignorando su magnitud. Dos imágenes similares tendrán embeddings con direcciones similares.

---

8. Función de Búsqueda

```
def search_similar_images(query_image, k=10):
    img = Image.open(query_image).convert('RGB')
    img_tensor = transform(img).unsqueeze(0).to(device)

    with torch.no_grad():
        query_embedding = model(img_tensor)
        query_embedding = query_embedding.squeeze().cpu().numpy().reshape(1, -1)

    faiss.normalize_L2(query_embedding)

    distances, indices = index.search(query_embedding, k)

    similar_images = []
    similar_labels = []

    for idx in indices[0]:
        similar_images.append(image_paths[idx])
        similar_labels.append(labels[idx])

    breed_counts = Counter(similar_labels)
    predicted_breed = breed_counts.most_common(1)[0][0]

    return similar_images, similar_labels, predicted_breed
```

¿Cómo funciona?

1. **Cargar y procesar imagen query**: Aplica las mismas transformaciones que al dataset
2. **Extraer embedding**: Usa ResNet50 para obtener el vector de 2048 dimensiones
3. **Normalizar**: Mismo proceso que con el dataset
4. **Buscar en FAISS**: `index.search()` encuentra los k=10 vectores más cercanos en milisegundos
5. **Recuperar imágenes**: Usa los índices devueltos para obtener las rutas y razas
6. **Voto mayoritario**: `Counter` cuenta cuántas veces aparece cada raza y elige la más común

¿Por qué voto mayoritario? Si 7 de las 10 imágenes más similares son "Beagle", es muy probable que la imagen query también sea un Beagle.

---

9. Interfaz de Gradio

```
def gradio_search(image):
    similar_images, similar_labels, predicted_breed = search_similar_images(image, k=10)

    result_text = f"Raza predicha: {predicted_breed}"

    gallery_images = []
    for img_path, label in zip(similar_images, similar_labels):
        gallery_images.append((img_path, label))

    return result_text, gallery_images
```

¿Por qué esta función wrapper? Adapta la salida de `search_similar_images` al formato que espera Gradio (texto + galería de imágenes con labels).

```
with gr.Blocks() as demo:
    gr.Markdown("# Buscador de Razas de Perros por Similitud")

    with gr.Row():
        with gr.Column():
            input_image = gr.Image(type="filepath", label="Subir imagen")
            search_btn = gr.Button("Buscar")

        with gr.Column():
            result_text = gr.Textbox(label="Resultado")

    gallery = gr.Gallery(label="Imágenes similares", columns=5, height="auto")

    search_btn.click(gradio_search, inputs=input_image, outputs=[result_text, gallery])

demo.launch(share=True)
```

¿Por qué Gradio?

- Crea una interfaz web interactiva con pocas líneas de código
- `share=True` : Genera un link público temporal para compartir la app
- La UI tiene: input para subir imagen, botón de búsqueda, texto con la predicción y galería con las 10 imágenes similares

---

10. Evaluación: NDCG@10

¿Qué es NDCG@10?

NDCG (Normalized Discounted Cumulative Gain) mide la calidad de un sistema de ranking.

¿Por qué usar NDCG en vez de accuracy?

- Nos importa el **orden** de los resultados, no solo si hay resultados correctos
- Resultados relevantes en las primeras posiciones son más valiosos
- Un resultado correcto en posición 1 vale más que uno en posición 10

¿Cómo funciona?

```
def dcg_at_k(relevances, k):
    relevances = np.array(relevances)[:k]
    return np.sum(relevances / np.log2(np.arange(2, len(relevances) + 2)))
```

- **DCG**: Suma las relevancias divididas por $\log_2$(posición+1)
- Las posiciones tempranas tienen más peso ($\log_2(2)=1$, $\log_2(3)\approx1.58$, $\log_2(11)\approx3.46$)
- Ejemplo: [1,1,0,0,0,0,0,0,0,0] → DCG = $1/\log_2(2) + 1/\log_2(3) \approx 1.63$

```
def ndcg_at_k(relevances, k):
    dcg = dcg_at_k(relevances, k)
    ideal_relevances = sorted(relevances, reverse=True)
    idcg = dcg_at_k(ideal_relevances, k)
    return dcg / idcg if idcg > 0 else 0.0
```

- **NDCG**: Normaliza DCG dividiéndolo por el DCG ideal (IDCG)
- IDCG = DCG si los resultados estuvieran perfectamente ordenados
- NDCG $\in [0,1]$, donde 1 = ranking perfecto

Ejemplo:

- Resultados: [Beagle, Beagle, Husky, Husky, ...] (query = Beagle)
- Relevancia: [1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
- DCG ≈ 1.63
- IDCG (ideal: todas las relevantes primero) ≈ max posible para esa cantidad
- NDCG = DCG/IDCG

Preparación del conjunto de prueba

```
df = pd.read_csv('dataset/dogs.csv')
test_df = df[df['data set'] == 'test']

breed_groups = test_df.groupby('labels')
test_samples = []

for breed, group in breed_groups:
    samples = group.sample(min(5, len(group)))
    test_samples.extend(samples['filepaths'].tolist())
```

¿Por qué?

- Seleccionamos 5 imágenes aleatorias por raza del conjunto "test"
- Estas imágenes NO se usaron para crear el índice (están separadas)
- Total: ~350 imágenes de prueba (70 razas × 5 imágenes)

Cálculo de NDCG@10

```
ndcg_scores = []

for test_path in test_samples:
    full_path = Path('dataset') / test_path
    true_label = df[df['filepaths'] == test_path]['labels'].values[0]

    similar_images, similar_labels, predicted_breed = search_similar_images(str(full_path), k=10)

    relevances = [1 if label == true_label else 0 for label in similar_labels]
    ndcg = ndcg_at_k(relevances, 10)
    ndcg_scores.append(ndcg)

avg_ndcg = np.mean(ndcg_scores)
```

¿Cómo se interpreta?

- Para cada imagen de prueba, buscamos las 10 más similares
- Marcamos con 1 si la raza coincide, 0 si no
- Calculamos NDCG@10 para esa búsqueda
- Promediamos NDCG sobre todas las imágenes de prueba

Valores esperados:

- NDCG@10 > 0.8: Excelente sistema de búsqueda
- NDCG@10 > 0.6: Bueno
- NDCG@10 < 0.4: Necesita mejoras

Resumen del Pipeline Completo

1. **Carga del dataset** (9300 imágenes, 70 razas)
2. **Extracción de embeddings** con ResNet50 pre-entrenado → vectores de 2048 dims
3. **Indexación FAISS** para búsquedas rápidas por similitud coseno
4. **Búsqueda**: Query → Embedding → FAISS → Top 10 imágenes similares
5. **Clasificación**: Voto mayoritario sobre las 10 imágenes recuperadas
6. **Interfaz**: Gradio para interacción web
7. **Evaluación**: NDCG@10 sobre conjunto de prueba

Ventajas de este Approach

- **No requiere entrenamiento**: Usa modelo pre-entrenado
- **Rápido**: FAISS busca en milisegundos
- **Interpretable**: Ves las imágenes similares que justifican la predicción
- **Escalable**: FAISS maneja millones de vectores
- **Flexible**: Fácil agregar nuevas razas sin reentrenar

Capítulo II: Entrenamiento y Comparación de Modelos (Etapa 2)

Objetivo

Entrenar modelos propios para mejorar la clasificación de razas de perros y compararlos con el modelo pre-entrenado de la Etapa 1.

Modelos a Entrenar

Modelo A: Transfer Learning con ResNet18

Fine-tuning de ResNet18 pre-entrenado en ImageNet sobre las 70 razas de perros.

Modelo B (Opcional): CNN Custom

Red convolucional diseñada desde cero para clasificación de perros.

---

1. Dataset y Transformaciones

Dataset Personalizado

```
class DogDataset(Dataset):
    def __init__(self, csv_file, root_dir, dataset_type, transform=None):
        df = pd.read_csv(csv_file)
        self.data = df[df['data set'] == dataset_type].reset_index(drop=True)
        self.root_dir = Path(root_dir)
        self.transform = transform

        self.label_to_idx = {label: idx for idx, label in enumerate(sorted(self.data['labels'].unique()))}
        self.idx_to_label = {idx: label for label, idx in self.label_to_idx.items()}
        self.num_classes = len(self.label_to_idx)
```

¿Por qué separar por dataset_type?

- Filtramos train/valid/test del CSV
- Cada conjunto tiene diferentes transformaciones
- `label_to_idx` : Convierte nombres de razas a índices numéricos (0-69)
- `idx_to_label` : Mapeo inverso para interpretación de resultados

Transformaciones

Para Entrenamiento (Data Augmentation):

```
train_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.RandomCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

¿Por qué estas transformaciones?

- `RandomCrop` : Variabilidad en posición del perro
- `RandomHorizontalFlip` : Aumenta dataset (perro mirando izquierda/derecha)
- `ColorJitter` : Robustez a diferentes condiciones de iluminación
- **Data Augmentation previene overfitting** al crear variaciones realistas

Para Validación/Test:

```
val_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

¿Por qué sin augmentation?

- Queremos evaluar el modelo en condiciones consistentes

- Solo `CenterCrop` (no random) para reproducibilidad

---

2. Modelo ResNet18 con Transfer Learning

Carga del Modelo Pre-entrenado

```
model = models.resnet18(weights=models.ResNet18_Weights.IMAGENET1K_V1)
num_features = model.fc.in_features
model.fc = nn.Linear(num_features, num_classes)
```

¿Qué es Transfer Learning?

- ResNet18 ya sabe reconocer patrones visuales básicos de ImageNet (1000 clases)
- Reemplazamos la última capa (FC) para clasificar 70 razas de perros
- Las capas convolucionales iniciales ya saben detectar bordes, texturas, formas

¿Por qué ResNet18 en vez de ResNet50?

- Más rápido de entrenar
- Menos parámetros = menos riesgo de overfitting
- Suficiente capacidad para 70 clases

Reemplazo de la Última Capa

- `model.fc.in_features` : 512 (salida de la última capa convolucional)
- Nueva FC: `512 → 70` (70 razas)
- **Solo entrenamos esta capa + fine-tuning de las anteriores**

---

3. Loss Function y Optimizer

CrossEntropyLoss

```
criterion = nn.CrossEntropyLoss()
```

¿Por qué CrossEntropyLoss?

- Estándar para clasificación multi-clase
- Combina `LogSoftmax + NLLLoss`
- Penaliza predicciones incorrectas proporcionalmente
- Entrada: logits (sin softmax), salida: scalar loss

Optimizer: Adam

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

¿Por qué Adam?

- Adapta el learning rate para cada parámetro
- Funciona bien sin mucho tuning
- Momentum + RMSprop combinados
- `lr=0.001` : Learning rate conservador para fine-tuning

Learning Rate Scheduler

```
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
```

¿Por qué usar scheduler?

- Reduce el learning rate cada 7 epochs
- `gamma=0.1` : LR se multiplica por 0.1 (ej: 0.001 → 0.0001)
- Al principio aprende rápido, luego refina con pasos más pequeños
- Previene oscilaciones en la convergencia

---

4. Funciones de Entrenamiento y Validación

train_epoch()

```
def train_epoch(model, loader, criterion, optimizer):
    model.train()  # Activa dropout y batch normalization
    running_loss = 0.0
    correct = 0
    total = 0

    for images, labels in tqdm(loader, desc="Entrenando"):
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()  # Limpia gradientes anteriores
        outputs = model(images)  # Forward pass
        loss = criterion(outputs, labels)  # Calcula loss
        loss.backward()  # Backward pass (calcula gradientes)
        optimizer.step()  # Actualiza pesos

        running_loss += loss.item()
        _, predicted = outputs.max(1)  # Clase con mayor probabilidad
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()

    epoch_loss = running_loss / len(loader)
    epoch_acc = 100. * correct / total
    return epoch_loss, epoch_acc
```

Flujo de entrenamiento:

1. `model.train()` : Activa modo entrenamiento (dropout, etc.)
2. `optimizer.zero_grad()` : Limpia gradientes del batch anterior
3. Forward pass: calcula predicciones
4. Calcula loss
5. Backward pass: calcula gradientes
6. `optimizer.step()` : actualiza pesos
7. Acumula métricas (loss, accuracy)

validate()

```
def validate(model, loader, criterion):
    model.eval()  # Desactiva dropout y batch normalization
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():  # No calcula gradientes (más rápido)
        for images, labels in tqdm(loader, desc="Validando"):
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)

            running_loss += loss.item()
            _, predicted = outputs.max(1)
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

    epoch_loss = running_loss / len(loader)
    epoch_acc = 100. * correct / total
    return epoch_loss, epoch_acc
```

Diferencias con train_epoch:

- `model.eval()` : Desactiva dropout/batch norm
- `torch.no_grad()` : No calcula gradientes (ahorra memoria)
- No hay `optimizer.step()` ni `backward()`
- Solo evalúa el modelo sin modificarlo

---

5. Loop de Entrenamiento

```
num_epochs = 15
best_val_acc = 0.0

for epoch in range(num_epochs):
    train_loss, train_acc = train_epoch(model, train_loader, criterion, optimizer)
    val_loss, val_acc = validate(model, val_loader, criterion)

    train_losses.append(train_loss)
    val_losses.append(val_loss)
    train_accs.append(train_acc)
    val_accs.append(val_acc)

    if val_acc > best_val_acc:
        best_val_acc = val_acc
        torch.save(model.state_dict(), 'resnet18_best.pth')

    scheduler.step()
```

¿Por qué guardar solo el mejor modelo?

- El modelo puede hacer overfitting en epochs finales
- Guardamos el checkpoint con mejor accuracy en validación
- Evita guardar un modelo que memorizó el training set

¿Por qué 15 epochs?

- Generalmente suficiente para converger en transfer learning
- Más epochs = riesgo de overfitting
- El scheduler reduce LR en epoch 7 y 14

---

6. Visualización de Resultados

Gráficas de Entrenamiento

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

ax1.plot(train_losses, label='Train Loss')
ax1.plot(val_losses, label='Val Loss')

ax2.plot(train_accs, label='Train Acc')
ax2.plot(val_accs, label='Val Acc')
```

¿Qué buscar en las gráficas?

- **Entrenamiento exitoso:** La pérdida (loss) disminuye consistentemente.
- **Overfitting:** La loss de validación empieza a subir mientras la de entrenamiento baja.
- **Underfitting:** Ambas loss son altas y no bajan.l acc (gap grande)

- Val loss aumenta mientras train loss baja
- Solución: más data augmentation, regularización

- **Underfitting:**

- Train acc y val acc bajas
- Loss no baja lo suficiente
- Solución: modelo más complejo, más epochs

---

7. Evaluación en Test Set

```
model.load_state_dict(torch.load('resnet18_best.pth'))
test_loss, test_acc = validate(model, test_loader, criterion)
```

¿Por qué evaluar en test?

- Validación se usó para seleccionar hiperparámetros
- Test set es la métrica final e imparcial
- **Nunca entrenar ni ajustar basándose en test set**

Métricas de Evaluación (Próximo Script)

Para cumplir con los requisitos de la etapa 2, calcularemos:

1. Confusion Matrix

Matriz que muestra predicciones correctas vs incorrectas para cada clase.

2. Métricas por Clase

- **Sensibilidad (Recall)**: De todos los perros de raza X, ¿cuántos identificamos correctamente?
  - `TP / (TP + FN)`
  - Importante cuando no queremos perder casos (ej: diagnósticos médicos)
- **Especificidad**: De todos los perros que NO son raza X, ¿cuántos clasificamos correctamente como NO-X?
  - `TN / (TN + FP)`
  - Importante para evitar falsos positivos
- **Precisión (Precision)**: De todos los que predijimos como raza X, ¿cuántos son realmente X?
  - `TP / (TP + FP)`
  - Importante cuando falsos positivos son costosos
- **F1-Score**: Media armónica de Precision y Recall
  - `2 * (Precision * Recall) / (Precision + Recall)`
  - Balancea precision y recall
- **Exactitud (Accuracy)**: Porcentaje total de predicciones correctas
  - `(TP + TN) / Total`
  - Puede ser engañosa con clases desbalanceadas

Ejemplo de Interpretación

Imagina clasificar "Golden Retriever":

- **TP (True Positive)**: 45 Golden correctamente identificados
- **FN (False Negative)**: 5 Golden clasificados como otra raza
- **FP (False Positive)**: 10 otras razas clasificadas como Golden
- **TN (True Negative)**: Todas las otras razas correctamente NO clasificadas como Golden

```
Sensibilidad = 45/(45+5) = 0.90 → Identificamos 90% de los Golden
Precisión = 45/(45+10) = 0.82 → 82% de nuestras predicciones "Golden" son correctas
F1-Score = 2*(0.90*0.82)/(0.90+0.82) = 0.86
```

Ventajas del Transfer Learning

- **Converge más rápido**: ResNet18 ya sabe reconocer patrones
- **Menos datos necesarios**: No necesitamos millones de imágenes
- **Mejor generalización**: Conocimiento de ImageNet ayuda
- **Menos overfitting**: Capas iniciales están pre-entrenadas

---

Comparación con Etapa 1

Aspecto	Etapa 1	Etapa 2
Modelo	ResNet50 pre-entrenado	ResNet18 fine-tuned
Tarea	Búsqueda por similitud	Clasificación directa
Entrenamiento	No	Sí (15 epochs)
Predicción	Voto mayoritario (k=10)	Softmax directa
Embeddings	2048 dims	512 dims
Velocidad	Rápida	Más rápida

¿Cuándo usar cada uno?

- **Etapa 1**: Exploración, interpretabilidad (ves imágenes similares)
- **Etapa 2**: Producción, mayor accuracy, predicción directa

---

Capítulo III: Pipeline de Detección y Clasificación (Etapa 3)

Objetivo

Crear un sistema completo que:

1. Detecte perros en imágenes con YOLO preentrenado
2. Clasifique la raza de cada perro detectado con ResNet18

---

1. Aplicación Principal (`app_deteccion.py`)

Aplicación Gradio con pipeline completo YOLO + ResNet18.

Pipeline:

1. YOLO preentrenado detecta bounding boxes de perros (clase 16 en COCO)
2. Recorta cada detección
3. ResNet18 clasifica la raza de cada recorte
4. Dibuja bounding boxes con etiquetas

Uso:

```
cd etapa-3
python app_deteccion.py
```

Abre http://127.0.0.1:7861

Features:

- Usa YOLOv8n preentrenado (yolov8n.pt) - NO requiere entrenamiento
- Detección múltiple (varios perros en una imagen)
- Confianza de detección + clasificación
- Visualización con bounding boxes

2. Evaluación del Pipeline (`evaluar_pipeline.py`)

Evalúa el rendimiento del pipeline completo en el conjunto de test.

Métricas:

- Tasa de detección
- Accuracy de clasificación
- Confianza promedio de detección
- Confianza promedio de clasificación
- F1-Score, Precision, Recall por clase

Uso:

```
python evaluar_pipeline.py
```

Genera `metricas_pipeline.csv` con métricas detalladas.

Flujo de Trabajo

1. Ejecutar aplicación

```
cd etapa-3
python app_deteccion.py
```

El modelo YOLOv8n preentrenado detecta perros automáticamente (clase 16 en COCO dataset). No es necesario entrenar el detector.

2. Evaluar pipeline

```
python evaluar_pipeline.py
```

Integración con Etapas Anteriores

Reutiliza:

- `utils/models.py` - Carga ResNet18
- `utils/transforms.py` - Transformaciones
- `utils/dataset.py` - Dataset y mapeo de clases
- `etapa-2/resnet18_best.pth` - Clasificador entrenado

Notas Importantes

- **YOLO preentrenado**: Según consigna del TP, NO es necesario entrenar el detector
- **Clase 16**: En COCO dataset, la clase 16 corresponde a "dog"
- **Multiple detecciones**: El pipeline soporta múltiples perros en una imagen
- **Optimización**: La Etapa 4 cubre cuantización y optimización de modelos

---

Capítulo IV: Evaluación, Optimización y MLOps (Etapa 4)

Objetivo

1. Evaluar el pipeline completo con imágenes anotadas manualmente
2. Optimizar modelos mediante cuantización
3. Generar anotaciones automáticas en formatos YOLOv5 y COCO

Archivos y Funcionalidades

1. evaluar_pipeline.py

Evalúa el desempeño del pipeline completo (YOLO + ResNet18) usando imágenes con anotaciones manuales.

Métricas calculadas:

- mAP (Mean Average Precision)
- IoU (Intersection over Union)
- Precision, Recall, F1-Score
- Accuracy de clasificación

Requisito previo: Crear archivo `anotaciones_manuales.json` con 10 imágenes complejas anotadas manualmente.

Formato del JSON:

```
{
  "images": [
    {
      "file": "ruta/a/imagen1.jpg",
      "annotations": [
        {
          "bbox": [x1, y1, x2, y2],
          "breed": "nombre_raza"
        }
      ]
    }
  ]
}
```

Ver `anotaciones_manuales_ejemplo.json` para referencia.

Uso:

```
cd etapa-4
python evaluar_pipeline.py
```

Genera `resultados_evaluacion.json` con métricas detalladas.

2. optimizar_modelos.py

Aplica cuantización al modelo ResNet18 para acelerar inferencia.

Proceso:

1. Carga modelo original (FP32)
2. Aplica cuantización dinámica (INT8)
3. Calibra con datos de validación
4. Guarda modelo cuantizado
5. Compara velocidad y precisión

Métricas comparadas:

- Accuracy
- Tiempo de inferencia
- Tamaño del modelo
- Speedup

Uso:

```
python optimizar_modelos.py
```

Genera:

- `resnet18_quantized.pth` - Modelo cuantizado
- `resultados_optimizacion.json` - Comparación de métricas

Alternativa: Exportar a ONNX El script incluye código comentado para exportar a ONNX si se prefiere.

3. anotar_automatico.py

Genera anotaciones automáticas usando el pipeline completo.

Formatos de salida:

- **YOLOv5 (.txt)**: Un archivo .txt por imagen con formato:

  ```
  class_id x_center y_center width height
  ```

  (coordenadas normalizadas 0-1)

- **COCO (.json)**: Archivo JSON con estructura COCO standard:
  - info
  - images
  - annotations
  - categories

Uso:

```
python anotar_automatico.py
```

El script pedirá la ruta de la carpeta con imágenes.

Genera:

- `anotaciones_yolo/` - Carpeta con archivos .txt
- `anotaciones_coco.json` - Archivo COCO

Flujo de Trabajo

Paso 1: Preparar Anotaciones Manuales

1. Recopilar 10 imágenes complejas (con perros y otros objetos)
2. Anotar manualmente cada imagen:
   - Bounding boxes de perros
   - Raza correcta
3. Crear `anotaciones_manuales.json`

Puedes usar herramientas como:

- LabelImg
- CVAT
- Roboflow

Paso 2: Evaluar Pipeline

```
python evaluar_pipeline.py
```

Revisa métricas en `resultados_evaluacion.json`

Paso 3: Optimizar Modelos

```
python optimizar_modelos.py
```

Analiza el trade-off entre velocidad y precisión.

Paso 4: Generar Anotaciones Automáticas

```
python anotar_automatico.py
```

Usa estas anotaciones para:

- Entrenar detectores custom
- Aumentar dataset
- Validación rápida

Dependencias

Ya incluidas en `requirements.txt`:

- torch (cuantización)
- ultralytics (YOLO)
- pycocotools (formato COCO)

Integración con Etapas Anteriores

Reutiliza:

- `etapa-2/resnet18_best.pth` - Modelo clasificador
- `etapa-3/` - Pipeline YOLO + clasificador
- `utils/` - Todos los componentes compartidos

Notas Importantes

- **10 imágenes**: El TP pide exactamente 10 imágenes complejas anotadas
- **Cuantización vs ONNX**: Elige una técnica de optimización
- **Formatos standard**: YOLOv5 y COCO son formatos ampliamente usados
- **IoU threshold**: 0.5 es el standard para calcular mAP@0.5

Capítulo V: Análisis de Resultados Experimentales

9.1 Resultados de Búsqueda (Etapa 1)

Evaluación sobre dataset de test (350 imágenes).

Métrica	Valor	Interpretación
NDCG@10	**0.9874**	El ranking por similitud es casi perfecto. ResNet50 agrupa razas visualmente idénticas de forma muy efectiva.
Accuracy (Voto)	**97.43%**	Usar los 10 vecinos para votar la clase es una estrategia viable sin entrenamiento.

9.2 Resultados de Clasificación (Etapa 2)

Entrenamiento de ResNet18 (15 épocas).

Métrica	ResNet50 (KNN - Etapa 1)	ResNet18 (Fine-Tuned - Etapa 2)

Métrica	ResNet50 (kNN - Etapa 1)	ResNet18 (Fine-Tuned - Etapa 2)
Accuracy	97.43%	94.86%
Precision	N/A	0.9531
Recall	N/A	0.9486
F1-Score	N/A	0.9483
Peso (MB)	~100 MB	~45 MB

Discusión: Aunque el accuracy puro bajó levemente, ResNet18 es 3 veces más rápida y liviana. El F1-Score de 0.95 indica un clasificador extremadamente robusto y balanceado. Además, ResNet18 resuelve mejor pares de razas difíciles (*Hairless Dogs*) que ResNet50 pre-entrenada.

Nota sobre la Matriz de Confusión: Los valores observados en la matriz de confusión (rango 0-10 aproximadamente) corresponden a **conteos absolutos** de imágenes en el conjunto de test, no a porcentajes ni correlaciones. Dado que el subset de evaluación contiene un número limitado de ejemplos por clase (~10 imágenes), estos valores reflejan directamente la cantidad de aciertos y errores en unidades de casos de prueba.

Interpretación de Resultados: La matriz exhibe una **diagonal dominante** clara, donde la gran mayoría de los valores (cercanos a 10) se concentran en la intersección Clase Real vs. Clase Predicha. Esto valida visualmente el Accuracy global del ~95%. Los pocos valores fuera de la diagonal (errores) no son aleatorios, sino que tienden a agruparse entre razas fenotípicamente similares (ej. *American Staffordshire Terrier* y *Staffordshire Bull Terrier*), lo cual es un comportamiento esperado y deseable en modelos de visión que capturan características semánticas robustas.

Análisis de Curvas de Entrenamiento: Al observar las gráficas de *Loss* y *Accuracy*, se confirma un proceso de aprendizaje saludable:

- **Loss:** La curva de validación (naranja) desciende consistentemente junto con la de entrenamiento (azul) hasta la época 7-8, indicando que el modelo generaliza bien y no memoriza datos (overfitting temprano).
- **Accuracy:** Se alcanza una meseta (plateau) de estabilidad alrededor del 92-93% en validación a partir de la mitad del entrenamiento. La pequeña brecha entre la curva de Train (~98%) y Test valida que el modelo mantiene su capacidad predictiva ante datos no vistos.

9.3 Desempeño del Detector YOLO (Etapa 3) (Aislado)

Métrica	Valor	Observación
Tasa de Detección	60.14%	Baja debido a fallos en primeros planos (caras).
Accuracy Clasificación	92.16%	Cuando detecta, clasifica bien.

Este 60% justifica completamente la implementación del **Fallback**. Sin él, el sistema ignoraría el 40% de las imágenes de usuario.

Análisis Detallado de Métricas (Archivo `metricas_pipeline.csv`): El pipeline alcanzó un **Accuracy Global del 92.16%** y un **F1-Score Ponderado de 0.9220** sobre 421 imágenes de prueba.

- **Rendimiento Perfecto (F1=1.0):** Se logró clasificación perfecta en razas distintivas como *African Wild Dog*, *Basset*, *Beagle*, *Bloodhound*, *Rottweiler* y *Yorkie*.
- **Casos Difíciles:** El modelo presentó dificultades con razas de alta variabilidad o cruces visuales, específicamente:
  - *Labradoodle* (F1=0.40): Confusión frecuente con Caniches y Labradores puro.
  - *Schnauzer* (F1=0.50): Dificultad para distinguir entre tamaños (Giant/Miniature) o mezclas.
  - *Dingo* (F1=0.67): Confusión con otros perros salvajes o mestizos genéricos.

9.4 Métricas del Pipeline Final (Etapa 4)

Evaluado sobre 10 imágenes *Ground Truth* complejas.

Métrica	Valor	Significado
mAP@0.5	**0.9417**	Alta precisión en localización + clasificación combinadas.
Precision	1.0000	0 Falsos Positivos.
Recall	**0.9333**	Recuperó casi todos los perros.

Optimización - Análisis Comparativo de Estrategias:

Escenario A: Cuantización Dinámica Completa (Standard/x86) Se intentó una conversión completa de pesos (Linear + Conv2d) a INT8, tal como se haría en un entorno de servidor estándar.

- **Tamaño Original:** 42.85 MB
- **Tamaño Cuantizado:** 10.93 MB
- **Reducción: ~75%** (Exitosa en términos de almacenamiento).
- **Limitación:** No se pudo evaluar en tiempo de ejecución (inferencia) en la arquitectura Apple Silicon debido a `RuntimeError` por falta de soporte de operadores convolucionales cuantizados en el backend `qnnpack`.

Escenario B: Cuantización Dinámica Selectiva (Apple Silicon M1) Se adaptó la estrategia mediante el script `optimizar_dinamico.py`, limitando la cuantización únicamente a las capas soportadas (`nn.Linear`) para lograr una ejecución exitosa.

- **Tiempo de Inferencia:** 1466ms $\rightarrow$ 1254ms por batch (**1.17x Speedup**)
- **Tamaño:** 42.85 MB $\rightarrow$ 42.75 MB (Reducción marginal del 0.2%).
- **Conclusión:** En arquitectura ARM, se logró priorizar la latencia (mejora del 17%) aunque no fue posible comprimir el tamaño del modelo significativamente con esta técnica.

Escenario C: Interoperabilidad con ONNX Runtime Se exploró el uso del formato abierto ONNX para desacoplar el modelo del framework PyTorch y evaluar su stack de optimización.

- **Modelo FP32 (Original):** Mostró un desempeño excepcional con ~20ms de latencia (Batch=1), posicionándose como **la opción más rápida** para inferencia en tiempo real en esta plataforma.
- **Modelo INT8 (Cuantizado):** Se logró una compresión exitosa de 42.76 MB a **10.74 MB (Reducción 4x)**, validando que el modelo es comprimible. Sin embargo, su ejecución falló con error `NotImplemented: ConvInteger`, confirmando que la barrera es la falta de implementación de kernels de convolución entera en los backends de CPU para Apple Silicon (tanto en PyTorch como en ONNX Runtime).

Ventajas Adicionales Verificadas de ONNX:

- **Portabilidad:** El formato funciona independientemente del framework original (PyTorch), facilitando el despliegue en C++, JavaScript o móviles.
- **Aceleración de Hardware:** Permite conectar con backends nativos como CoreML (macOS) o TensorRT (NVIDIA) sin cambiar el código del modelo.
- **Producción:** ONNX Runtime está altamente optimizado para entornos de producción, reduciendo el overhead de Python.

---

10. Bitácora de Desarrollo: Dificultades y Soluciones

Documentación de los obstáculos técnicos superados durante el ciclo de vida del proyecto.

A. El Problema de la Ventana Congelada (OpenCV en macOS)

- **Síntoma:** Al ejecutar el `etiquetador_manual.py`, tras dibujar una caja, la terminal pedía el ID de la raza, pero la ventana de la imagen dejaba de responder y el SO la marcaba como "No responde".
- **Causa:** El hilo principal de la UI de OpenCV necesita procesar eventos periódicamente. La función bloqueante `input()` detenía este procesamiento.
- **Solución:** Implementar un bucle no bloqueante o insertar `cv2.waitKey(1)` forzado antes y después de las operaciones de E/S de consola.

B. Incompatibilidad de Backend de Cuantización (ARM M1/M2)

- **Síntoma:** Error `RuntimeError: NoQEngine: No QEngine is set` o `NotImplementedError` al intentar optimizar el modelo.
- **Causa:** La distribución estándar de PyTorch predetermina el backend `fbgemm` (optimizado para Intel x86). Los procesadores Apple Silicon requieren el backend `qnnpack`.
- **Solución:** Detección de arquitectura en tiempo de ejecución:

  ```
  import platform
  if 'arm' in platform.machine():
      torch.backends.quantized.engine = 'qnnpack'
  ```

  Además, se descubrió que incluso con `qnnpack`, ciertas operaciones convolucionales cuantizadas (`quantized::conv2d.new`) no están completamente implementadas en la versión actual de PyTorch para CPU ARM. Esta limitación **también afecta a ONNX Runtime**, que reporta `[ONNXRuntimeError] : 9 : NOT_IMPLEMENTED : Could not find an implementation for ConvInteger`.

  Esto confirma que la barrera es a nivel de drivers/implementación de bajo nivel de operaciones enteras en la plataforma Apple Silicon para inferencia en CPU. Se manejó esto con bloques `try-except` en ambos scripts (`optimizar_dinamico.py` y `optimizar_onnx.py`) para permitir que la ejecución continúe y reporte los resultados parciales (tamaño de modelo o velocidad FP32) sin detener el flujo.

C. Seguridad de Rutas en Gradio

- **Síntoma:** `InvalidPathError` al intentar mostrar imágenes.
- **Causa:** Por seguridad, las versiones recientes de Gradio impiden acceso a archivos fuera del directorio de trabajo del script.
- **Solución:** Configurar el parámetro `allowed_paths` al lanzar la aplicación: `demo.launch(allowed_paths=["ruta/al/repo"])`.

---

11. Conclusión

Este trabajo práctico ha permitido simular un ciclo completo de desarrollo de IA. Se ha transitado desde la teoría matemática de las CNNs hasta la "fontanería" del código necesaria para hacer que los modelos funcionen en el mundo real.

Puntos clave de éxito:

1. **Arquitectura Robusta:** La decisión de separar la lógica en `utils` pagó dividendos, permitiendo iterar rápido en las etapas 3 y 4 sin romper el código base.
2. **Pragmatismo sobre Pureza:** La implementación del *Fallback* en detección demuestra que, en ingeniería aplicada, una heurística simple que funciona es superior a un modelo complejo que falla en casos de borde.
3. **Optimización (M1/M2):** Se evaluaron tres estrategias. La cuantización completa (PyTorch/ONNX) logró reducir el modelo a 10MB (4x) pero no es ejecutable en la CPU M1 actual. La cuantización selectiva (PyTorch) ofreció un speedup del 17%. Finalmente, **ONNX Runtime en FP32** emergió como la mejor opción para despliegue, ofreciendo la menor latencia (~20ms) y máxima portabilidad, aunque sin reducción de disco.

---

12. Anexo A: Stack Tecnológico

Lista detallada de las tecnologías y versiones utilizadas.

- **Lenguaje de Programación:** Python 3.10
- **Deep Learning Framework:** PyTorch 2.1.0 + Torchvision 0.16.0
- **Computer Vision:**
  - Ultralytics YOLOv8 (v8.0.196) - Detección
  - OpenCV (headless) - Procesamiento de imagen
  - Pillow (PIL) - Manipulación de imagen
- **Data Science:**
  - Pandas - Manejo de CSV
  - NumPy - Álgebra lineal
  - Matplotlib - Visualización
- **Búsqueda Vectorial:** FAISS-CPU (Facebook AI Similarity Search)
- **Interfaz de Usuario:** Gradio (v3.50.2)
- **Hardware de Desarrollo:**
  - Procesador: Apple M1/M2 (ARM64)
  - Aceleración: MPS (Metal Performance Shaders) para entrenamiento