TRABAJO FINAL

Visión por Computador

9 DE ENERO DE 2025

REALIZADO POR: YERAY ÁLVAREZ-BUYLLA PARRA Y MARÍA ELENA NAVARRO SANTANA

Índice

Motivación del trabajo 2

Objetivo de la propuesta 2

Descripción técnica del trabajo realizado 3

Desarrollo: YOLO 5

Desarrollo: RetinaNet 11

Conclusiones y propuestas de ampliación 15

Diario de reuniones del grupo 17

Enlace al código fuente 17

Fuentes y tecnologías utilizadas 17

Motivación del trabajo

La clasificación y detección de imágenes de huesos es un área de investigación crucial en el ámbito médico y científico. Con el creciente volumen de datos en radiografías, tomografías y otras imágenes médicas, surge la necesidad de herramientas que puedan analizar eficientemente estos datos para asistir en diagnósticos precisos. Este proyecto nace de la motivación de contribuir al desarrollo de tecnologías que optimicen la interpretación de imágenes médicas, reduciendo la carga de trabajo de los profesionales de la salud y minimizando el riesgo de error humano.

Para el entrenamiento de modelos avanzados se ha decidido usar YOLO y RetinaNet en estos conjuntos de datos específicos que proporciona una oportunidad única para evaluar el rendimiento y la precisión de diferentes algoritmos en esta tarea.

Al comparar varios datasets se nos permite identificar las características que influyen en una clasificación más efectivamente, lo que no solo mejora el entendimiento técnico de estos modelos, sino que también aporta insights valiosos para seleccionar datasets más representativos y robustos.

Se busca en este trabajo establecer una base sólida para futuras investigaciones en el uso de inteligencia artificial en imágenes médicas, promover soluciones accesibles y escalables que puedan ser implementadas en clínicas y hospitales para mejorar la calidad de la atención médica.

Objetivo de la propuesta

El objetivo principal de esta propuesta es analizar y comparar el desempeño de los modelos de detección de imágenes YOLO y RetinaNet en el contexto de imágenes de huesos, utilizando tres datasets diferentes. La evaluación de estos modelos se realizará utilizando métricas estándar en tareas de detección de objetos, específicamente:

1. **Precisión (Precisión):** Proporción de predicciones positivas correctas entre todas las predicciones positivas realizadas por el modelo, reflejando la capacidad de minimizar falsos positivos.
2. **Recall (Sensibilidad):** Proporción de predicciones positivas correctas entre todas las instancias positivas reales, indicando la capacidad del modelo para detectar todos los objetos relevantes.
3. **mAP@50 (Mean Average Precision at IoU=50):** Promedio de la precisión obtenida para todos los objetos en el dataset, calculada a un umbral de

50% de intersección sobre unión (IoU), lo que evalúa la precisión general del modelo.

4. **mAP@50-95:** Métrica más estricta que calcula el promedio de precisiones en umbrales de IoU desde 50% hasta 95%, en incrementos de 5%, proporcionando una visión más completa del rendimiento del modelo.

A través del uso de estas métricas, el trabajo busca:

- **Comparar el rendimiento de YOLO y RetinaNet** en términos de precisión, recall y mAP en los tres datasets seleccionados.
- **Determinar las fortalezas y debilidades de cada modelo** al aplicarlos a conjuntos de datos específicos de imágenes de huesos.
- **Identificar las características críticas de los datasets** que contribuyen a un mejor rendimiento en la tarea de detección y clasificación.
- **Proveer conclusiones y recomendaciones prácticas** para la implementación de estos modelos en aplicaciones médicas.

Descripción técnica del trabajo realizado

Para cumplir con los objetivos propuestos, el trabajo se desarrolló en varias etapas:

1. Selección y preparación de datasets

Se seleccionaron tres conjuntos de datos diferentes, cada uno con imágenes de huesos etiquetadas para tareas de detección de objetos que cumplieran las siguientes características:

- **Conversión de formatos:** Estandarización de los formatos de las etiquetas (JSON y TXT) para garantizar compatibilidad con los modelos YOLO y RetinaNet.
- **División de datos:** Separación de los datasets en subconjuntos de entrenamiento (70%), validación (20%) y prueba (10%).
- Algunas imágenes con **data augmentation**.

2. Paquetes necesarios

- **torch**
- **torchvision**
- **cv2**
- **os**
- **json**
- **tqdm**

- **math**
- **torchmetrics**
- **ultralytics**
- **pandas**

3. Entrenamiento de los modelos

Se entrenaron los modelos en **YOLO** y **RetinaNet**.

YOLO (You Only Look Once): Modelo entrenado en múltiples configuraciones, utilizando una arquitectura ligera para maximizar la velocidad de inferencia.

RetinaNet: Modelo optimizado para maximizar la precisión, utilizando el algoritmo Focal Loss para manejar el desbalance de clases.

Configuraciones de entrenamiento:

- 50 épocas para YOLO.
- 10 épocas y función de pérdida personalizada en el caso de RetinaNet.

4. Evaluación del rendimiento

Se utilizó un conjunto de pruebas común para evaluar ambos modelos, aplicando métricas estándar como:

- Precisión (Precision)
- Recall (Sensibilidad)
- mAP@50
- mAP@50-95

El proceso de evaluación se automatizó mediante un script que ejecuta cada modelo y registra los resultados en un formato estructurado para facilitar el análisis.

5. Análisis comparativo

Los resultados obtenidos fueron organizados en tablas y gráficos para realizar una comparación detallada entre los modelos y los datasets.

Se analizaron las características de los datasets (como tamaño, variabilidad y calidad de las etiquetas) para identificar su impacto en el rendimiento de los modelos.

Se evaluó la eficiencia computacional de cada modelo, considerando factores como el tiempo de inferencia y el consumo de recursos.

Desarrollo: YOLO

Primero se ha creado el entorno, se han instalado los paquetes necesarios y se ha activado:

```
(base) C:\Users\lenin>conda create --name Trabajo_Final python=3.9.5
```

```
(base) C:\Users\lenin>conda activate Trabajo_Final
```

Se ha entrenado el primer modelo de YOLO por CPU:

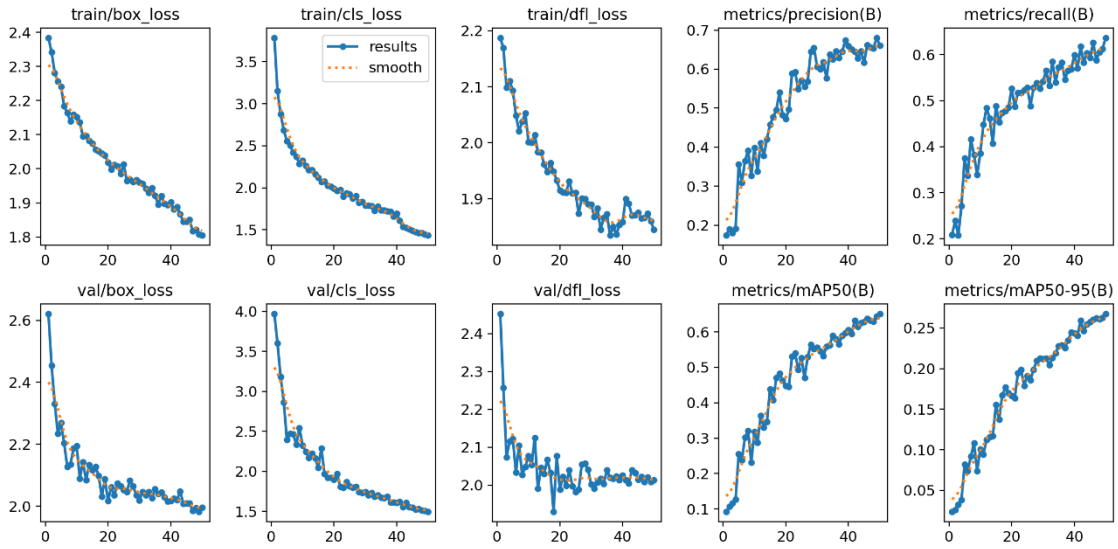
```
(base) C:\Users\lenin>conda activate Trabajo_Final

(Trabajo_Final) C:\Users\lenin>cd C:\Users\lenin\Documents\Universidad_2024-2025\VC\Trabajo_final\Pruebas\Bone-Yolov11

(Trabajo_Final) C:\Users\lenin\Documents\Universidad_2024-2025\VC\Trabajo_final\Pruebas\Bone-Yolov11>yolo detect train m
odel=yolo11n.pt data=data/miarchivo.yml imgsz=640 batch=4 device=CPU epochs=50
```

```
Logging results to runs\detect\train
Starting training for 50 epochs...

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       1/50         0G      2.384      3.787      2.187         14        640: 100%|██████████| 353/353 [06:24<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 51/51 [00:41
                   all        403        661      0.174      0.209     0.0928     0.0239

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       2/50         0G      2.342      3.153      2.169          5        640: 100%|██████████| 353/353 [06:27<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 51/51 [00:36
                   all        403        661       0.19      0.239      0.107     0.0263

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       3/50         0G      2.281      2.875      2.099          2        640: 100%|██████████| 353/353 [06:21<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 51/51 [00:35
                   all        403        661      0.181      0.207      0.116     0.0327

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       4/50         0G      2.256      2.687       2.11          6        640: 100%|██████████| 353/353 [06:26<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 51/51 [00:38
                   all        403        661      0.191      0.271      0.127     0.0385

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       5/50         0G      2.24       2.556      2.094         12        640: 100%|██████████| 353/353 [06:40<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 51/51 [00:39
                   all        403        661      0.356      0.375      0.255     0.0818
```

Ha terminado en 6.6 horas y ha dado la siguiente eficiencia para su conjunto de validación al terminar:

```
50 epochs completed in 6.615 hours.
Optimizer stripped from runs\detect\train\weights\last.pt, 5.5MB
Optimizer stripped from runs\detect\train\weights\best.pt, 5.5MB

Validating runs\detect\train\weights\best.pt...
Ultralytics 8.3.40 🚀 Python-3.9.5 torch-2.5.1+cpu CPU (13th Gen Intel Core(TM) i7-13700HX)
YOLO11n summary (fused): 238 layers, 2,582,347 parameters, 0 gradients, 6.3 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 51/51 [00:30
                   all        403        661      0.659      0.635       0.65      0.267
Speed: 0.8ms preprocess, 66.2ms inference, 0.0ms loss, 0.9ms postprocess per image
Results saved to runs\detect\train
💡 Learn more at https://docs.ultralytics.com/modes/train
```

Ha dado la siguientes gráficas:



- **train/box_loss:** El valor empieza alto (alrededor de 2.4) y disminuye progresivamente, lo que indica que el modelo está aprendiendo a ajustar las cajas correctamente.
- **train/cls_loss:** También disminuye con el tiempo, lo cual es una buena señal, porque significa que el modelo está mejorando su capacidad para identificar correctamente las clases.
- **train/dfl_loss:** Similar a las demás pérdidas, disminuye a lo largo del entrenamiento, indicando progreso.
- **metrics/precision(B):** Se observa una tendencia creciente, lo cual es ideal porque significa que el modelo está cometiendo menos falsos positivos.
- **metrics/recall(B):** Al aumentar, muestra que el modelo está detectando más objetos reales a medida que entrena.
- **val/box_loss:** El descenso constante sugiere que el modelo está aprendiendo a ajustar las cajas para nuevos datos.
- **val/cls_loss:** También disminuye, aunque hay mayor variabilidad que en el conjunto de entrenamiento, lo cual es normal.
- **val/dfl_loss:** Tiene un patrón similar a los anteriores de disminución gradual.
- **metrics/mAP50(B):** Crece a lo largo de las épocas, mostrando que el modelo es cada vez más preciso en datos de validación.
- **metrics/mAP50-95(B):** Aunque es más difícil de mejorar, el aumento indica que el modelo generaliza bien en datos no vistos.

Se ha entrenado el segundo modelo de YOLO por CPU:

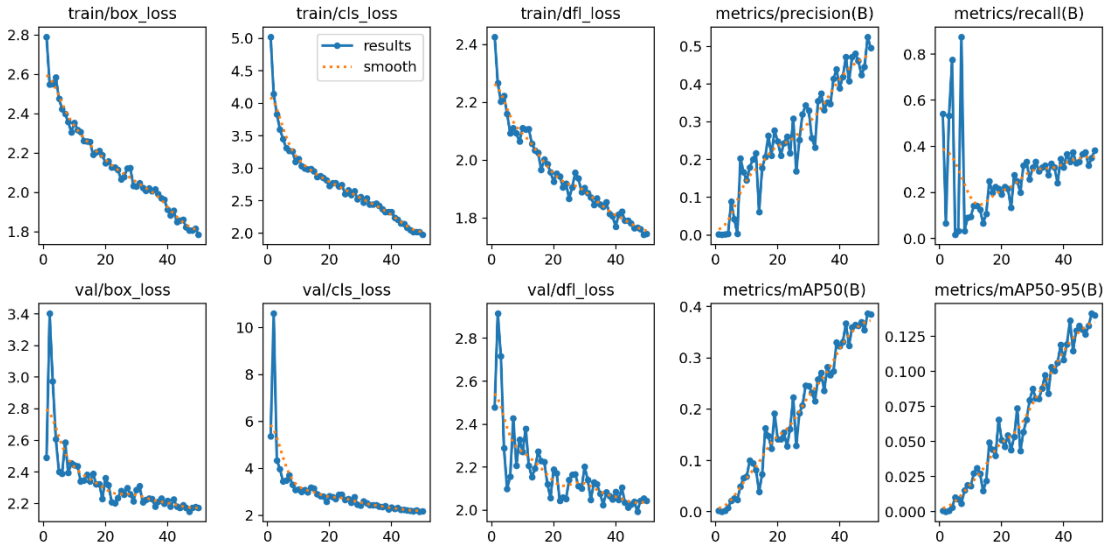
```
(base) C:\Users\lenin>conda activate Trabajo_Final

(Trabajo_Final) C:\Users\lenin>cd C:\Users\lenin\Documents\Universidad_2024-2025\VC\Trabajo_final\Pruebas2\bone2_yolov11

(Trabajo_Final) C:\Users\lenin\Documents\Universidad_2024-2025\VC\Trabajo_final\Pruebas2\bone2_yolov11>yolo detect train
 model=yolo11n.pt data=data/miarchivo.yml imgsz=640 batch=4 device=CPU epochs=50
```

```
Logging results to runs\detect\train
Starting training for 50 epochs...

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       1/50         0G       2.79      5.024      2.426          8        640: 100%|██████████| 208/208 [05:02<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 15/15 [00:17
                   all        114        120     0.0019      0.542    0.00206   0.000729

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       2/50         0G      2.549      4.145      2.267          4        640: 100%|██████████| 208/208 [05:05<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 15/15 [00:26
                   all        114        120   0.000626     0.0667   0.000562   0.000139

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       3/50         0G       2.55      3.834      2.202          4        640: 100%|██████████| 208/208 [05:02<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 15/15 [00:13
                   all        114        120    0.00188      0.533    0.00217   0.000643

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       4/50         0G      2.586      3.598      2.223          4        640: 100%|██████████| 208/208 [05:01<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 15/15 [00:14
                   all        114        120    0.00276      0.775    0.00722    0.00309

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       5/50         0G      2.476      3.455       2.16         10        640: 100%|██████████| 208/208 [04:59<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 15/15 [00:12
                   all        114        120     0.0889     0.0167     0.0219     0.0103
```

Ha terminado en 3.6 horas y ha dado la siguiente eficiencia para su conjunto de validación al terminar:

```
50 epochs completed in 3.653 hours.
Optimizer stripped from runs\detect\train\weights\last.pt, 5.5MB
Optimizer stripped from runs\detect\train\weights\best.pt, 5.5MB

Validating runs\detect\train\weights\best.pt...
Ultralytics 8.3.40 🚀 Python-3.9.5 torch-2.5.1+cpu CPU (13th Gen Intel Core(TM) i7-13700HX)
YOLO11n summary (fused): 238 layers, 2,582,347 parameters, 0 gradients, 6.3 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|██████████| 15/15 [00:07
                   all        114        120      0.528      0.354      0.386      0.141
Speed: 0.7ms preprocess, 59.8ms inference, 0.0ms loss, 0.9ms postprocess per image
Results saved to runs\detect\train
💡 Learn more at https://docs.ultralytics.com/modes/train
```

Ha dado las siguientes gráficas:



- **train/box_loss**: La pérdida asociada a las cajas delimitadoras durante el entrenamiento disminuye de forma constante, comenzando cerca de 2.8 y llegando aproximadamente a 1.8. Esto es una buena señal porque indica que el modelo está aprendiendo a localizar mejor los objetos en las imágenes.

- **train/cls_loss**: La pérdida asociada a la clasificación de los objetos también disminuye de manera consistente, de 5.0 a menos de 2.0. El descenso es más abrupto al inicio, lo cual es normal en los primeros pasos del entrenamiento.
- **train/dfl_loss**: Similar a las otras pérdidas, esta métrica de localización mejora progresivamente, disminuyendo de 2.4 a alrededor de 1.8. El patrón es estable, lo que sugiere que el modelo mejora su precisión en la localización de los objetos.
- **metrics/precision(B)**: La precisión aumenta de manera consistente, alcanzando un valor cercano a 0.5 al final del entrenamiento. Este incremento indica que el modelo está reduciendo los falsos positivos, mejorando su confianza al detectar objetos.
- **metrics/recall(B)**: La recuperación tiene un comportamiento más inestable, comenzando con valores altos, disminuyendo drásticamente, y luego aumentando de nuevo. Esto podría indicar que el modelo al principio detecta muchos objetos (pero no necesariamente de forma correcta), y luego mejora en detectar sólo los relevantes.
- **val/box_loss**: Al igual que en entrenamiento, la pérdida asociada a las cajas delimitadoras en validación disminuye progresivamente, de 3.4 a alrededor de 2.2. Esto indica que el modelo está generalizando bien para datos no vistos.
- **val/cls_loss:** La pérdida de clasificación en validación empieza muy alta (casi 10), pero disminuye rápidamente y se estabiliza alrededor de 2.5. Este descenso brusco es normal al principio y sugiere que el modelo está mejorando en categorizar los objetos correctamente en el conjunto de validación.
- **val/dfl_loss:** La pérdida de localización en validación muestra un comportamiento similar a las otras métricas, comenzando alta (casi 2.8) y estabilizándose alrededor de 2.0. Esto es positivo porque demuestra que el modelo está aprendiendo a localizar los objetos incluso en datos no vistos.
- **metrics/mAP50(B):** El mAP al 50% de IoU mejora constantemente, alcanzando valores cercanos a 0.4. Esto indica que el modelo está logrando un buen equilibrio entre precisión y recuperación en validación.
- **metrics/mAP50-95(B):** El mAP en un rango de IoU (50%-95%) también mejora, aunque los valores son menores (alrededor de 0.125). Esto es normal porque esta métrica es más estricta y mide la capacidad del modelo para ser preciso bajo diferentes tolerancias.

Se entrenado el tercer modelo de YOLO por CPU:

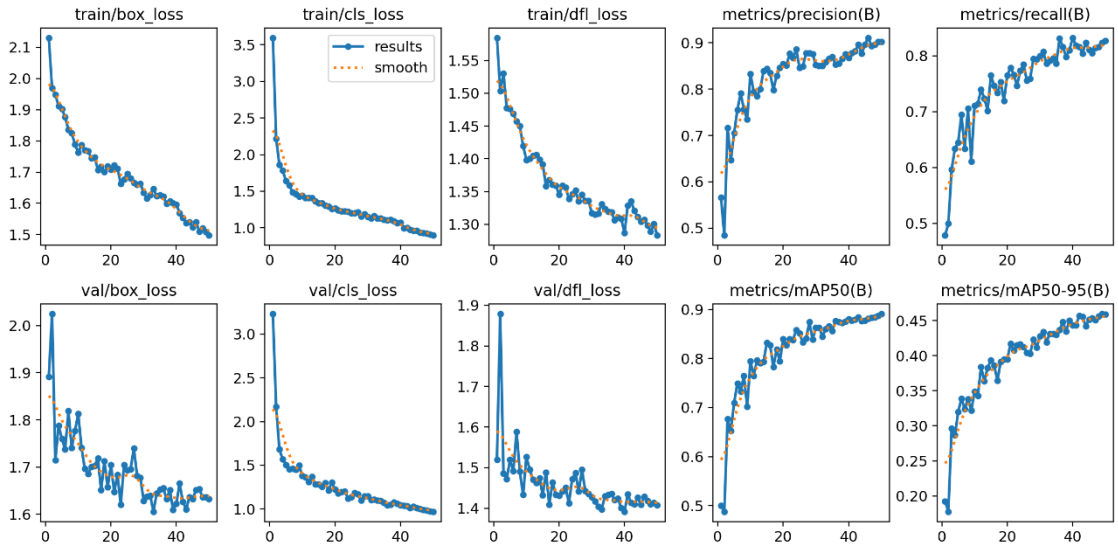
```
(Trabajo_Final) C:\Users\lenin\Documents\Universidad_2024-2025\VC\Trabajo_final\Pruebas3>cd bone3_yolov11

(Trabajo_Final) C:\Users\lenin\Documents\Universidad_2024-2025\VC\Trabajo_final\Pruebas3\bone3_yolov11>yolo detect train
 model=yolo11n.pt data=data/miarchivo.yml imgsz=640 batch=4 device=CPU epochs=50
```

```
Logging results to runs\detect\train
Starting training for 50 epochs...

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       1/50         0G      2.131      3.596      1.585          2        640: 100%|██████████| 436/43
6 [08:14<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|
██| 62/62 [00:46
                   all        494        560      0.566      0.479        0.5      0.193

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       2/50         0G       1.97      2.216      1.504          3        640: 100%|██████████| 436/43
6 [07:53<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|
██| 62/62 [00:43
                   all        494        560      0.484        0.5      0.488      0.178

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       3/50         0G      1.948      1.861       1.53          3        640: 100%|██████████| 436/43
6 [07:53<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|
██| 62/62 [00:39
                   all        494        560      0.716      0.596      0.677      0.296

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       4/50         0G      1.911      1.786      1.477          4        640: 100%|██████████| 436/43
6 [07:31<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|
██| 62/62 [00:40
                   all        494        560      0.647      0.634      0.652      0.287

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       5/50         0G      1.902      1.639      1.476          5        640: 100%|██████████| 436/43
6 [08:13<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|
██| 62/62 [00:43
                   all        494        560      0.705      0.645       0.71       0.32

      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
       6/50         0G      1.876      1.581      1.469          7        640: 100%|██████████| 436/43
6 [08:08<00:00,
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|
██| 62/62 [00:41
                   all        494        560      0.755      0.695      0.749      0.339
```

Ha terminado en 6.1 horas y ha dado la siguiente eficiencia para su conjunto de validación al terminar:

```
50 epochs completed in 6.131 hours.
Optimizer stripped from runs\detect\train\weights\last.pt, 5.5MB
Optimizer stripped from runs\detect\train\weights\best.pt, 5.5MB

Validating runs\detect\train\weights\best.pt...
Ultralytics 8.3.40 🚀 Python-3.9.5 torch-2.5.1+cpu CPU (13th Gen Intel Core(TM) i7-13700HX)
YOLO11n summary (fused): 238 layers, 2,582,347 parameters, 0 gradients, 6.3 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100%|
| 62/62 [00:27
                   all        494        560      0.901      0.821      0.886      0.459
Speed: 0.6ms preprocess, 50.2ms inference, 0.0ms loss, 0.6ms postprocess per image
Results saved to runs\detect\train
💡 Learn more at https://docs.ultralytics.com/modes/train
```

Ha dado las siguientes gráficas:



- **train/box_loss**: La pérdida comienza en un valor alto (2.1) y disminuye progresivamente hasta 1.5. Esto indica que el modelo está mejorando al ajustar las cajas a los objetos en las imágenes.
- **train/cls_loss**:Comienza en 3.5 y disminuye hasta aproximadamente 1.0. Esto significa que el modelo está aprendiendo a clasificar correctamente los objetos.
- **train/dfl_loss**: Comienza alrededor de 1.55 y disminuye a 1.3. Una reducción constante indica que el modelo está ajustando cada vez mejor las predicciones de localización.
- **metrics/precision(B)**: La precisión sube desde 0.5 hasta cerca de 0.9. Esto muestra que el modelo mejora al ser más certero en las detecciones.
- **metrics/recall(B)**: La recuperación sube de 0.5 a 0.8, mostrando que el modelo detecta más objetos correctamente con el tiempo.
- **val/box_loss**: Comienza en 2.0 y disminuye hasta 1.6, mostrando que el modelo también mejora en datos nuevos.
- **val/cls_loss**: Disminuye desde 3.0 hasta 1.0, lo que indica que el modelo no solo aprende a clasificar en entrenamiento, sino que también generaliza bien.
- **val/dfl_loss**: Baja de 1.9 a 1.4, mostrando que el modelo es preciso al localizar objetos incluso en datos que no ha visto antes.
- **metrics/mAP50(B)**: Crece de 0.5 a 0.9, lo cual es excelente. Significa que el modelo detecta y clasifica correctamente la mayoría de los objetos.
- **metrics/mAP50-95(B)**: Mejora de 0.2 a 0.45. Aunque los valores son menores (lo cual es normal), el crecimiento muestra que el modelo está logrando una detección precisa incluso bajo criterios más exigentes.

El modelo que mejor va con su conjunto de datos es el modelo 3 teniendo unas métricas bastante buenas y altas como se puede apreciar en la etapa final del entrenamiento.

Desarrollo: RetinaNet

El desarrollo para el uso del modelo RetinaNet ha requerido bastantes cambios con respecto al desarrollo en YOLO, ya que se nos han presentado bastantes problemas inesperados a la hora de entrenar.

La primera problemática a la que nos enfrentamos fue el cambio de tecnología, ya que para esta parte usamos Pytorch, por lo que no se pudo reutilizar el código de desarrollo de YOLO.

Lo segundo fue, que al hacer fine-tuning al modelo de RetinaNet, se tenía que cambiar solo la cabecera de clasificación, y se nos presentaron errores relacionados al no adaptarla al número de clases con el que queríamos entrenar. Esto se solucionó con el siguiente código:

```
# 4. Carga RetinaNet preentrenado
model = retinanet_resnet50_fpn(pretrained=True)

num_classes = 2

out_channels = model.head.classification_head.conv[0].out_channels
num_anchors = model.head.classification_head.num_anchors
model.head.classification_head.num_classes = num_classes

cls_logits = torch.nn.Conv2d(out_channels, num_anchors * num_classes, kernel_size = 3, stride=1, padding=1)
torch.nn.init.normal_(cls_logits.weight, std=0.01)
torch.nn.init.constant_(cls_logits.bias, -math.log((1 - 0.01) / 0.01))

model.head.classification_head.cls_logits = cls_logits
```

Posteriormente, al entrenar el modelo nos dimos cuenta de que los resultados eran de 0, por lo que se revisó el código exhaustivamente para encontrar posibles fallos a la hora de entrenar. Varios de los fallos consistieron en:

- Falta de uso de la función de pérdida propia de RetinaNet (sigmoid_focal_loss() para Pytorch)
- Uso de una función de optimización subóptima(SGD()).
- Entrenamiento indebido de las capas profundas del modelo de ResNet
- Sobreentrenamiento para fine-tuning(50)

Para cada uno de estos fallos cambiamos el código de las siguientes maneras:

1. Usar la función de pérdida de RetinaNet:

```
# Cálculo de la pérdida con sigmoide focal
classification_loss = sigmoid_focal_loss(
    classification_logits,
    torch.zeros_like(classification_logits),
    reduction="sum"
)
```

2. Usar la función de optimización AdamW, más apropiada para el uso de RetinaNet:

```
# 5. Configura el optimizador
optimizer = torch.optim.AdamW(filter(lambda p: p.requires_grad, model.parameters()), lr=0.001, weight_decay=0.0001)
scaler = GradScaler()
```

3. Congelar las capas necesarias para evitar desconfigurar el modelo:

```
# Congelar todas las capas excepto la última
for param in model.parameters():
    param.requires_grad = False

# Activar gradiente solo para la última capa (cls_logits)
for param in model.head.classification_head.cls_logits.parameters():
    param.requires_grad = True
```

4. Cambiar el número de épocas a 10:

```
num_epochs = 10
```

Incluso con estos cambios, el modelo siguió sin predecir correctamente. En este punto, pedimos una tutoría al profesor José Ignacio, en la que nos advirtió que el tamaño de las imágenes no correspondía al de las esperadas por las anotaciones, debido al uso de la función resize(), en la que no se estaba teniendo en cuenta el cambiar los valores dentro de las anotaciones. Al ver esto, corregimos el valor de entrada de la función resize() al del esperado por las anotaciones(de 300x300 a 640x640).

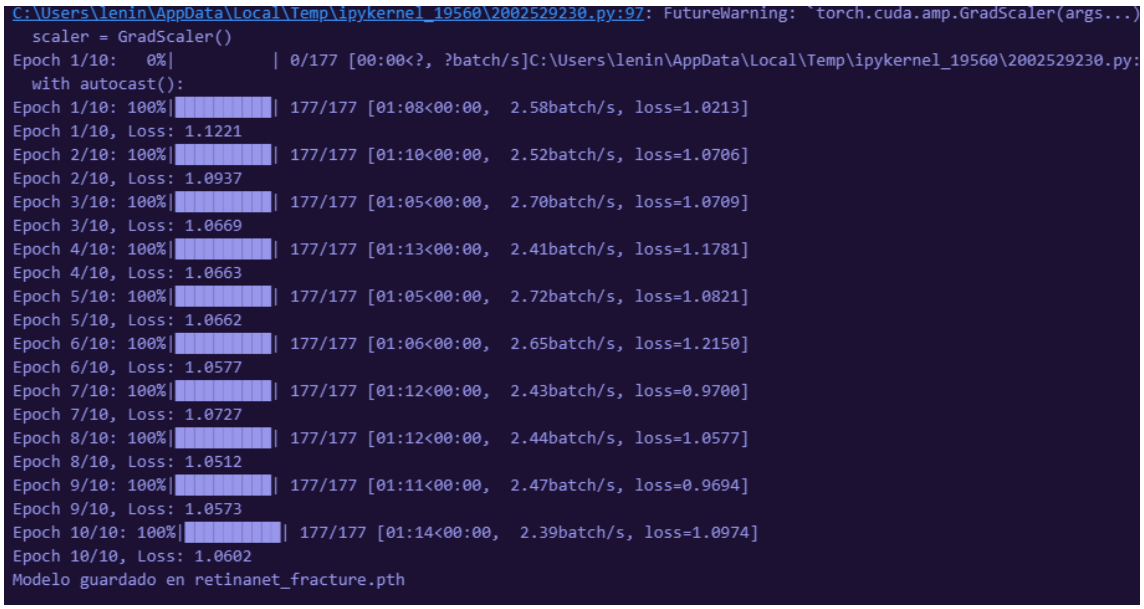
```
# 2. Define transformaciones
class ResizeTransform:
    def __call__(self, image):
        image = cv2.resize(image, (640, 640))
        return F.to_tensor(image)

transforms = ResizeTransform()
```

Por último, el modelo sigue teniendo un fallo que, a pesar de haber realizado todos estos cambios, sigue ocurriendo en gran parte de los entrenamientos:

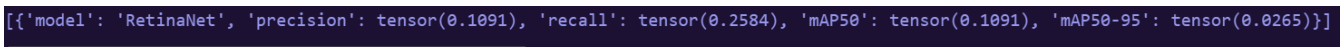
aleatoriamente, tras pasar varias épocas, el valor de pérdida del modelo se convierte en NaN(Not a Number). A pesar de estar más de una semana para corregir ese error, no se ha conseguido solucionar de manera satisfactoria.

A pesar de esto, sí que se ha conseguido entrenar lo suficiente como para que el modelo haga predicciones. Abajo se muestran los modelos con las métricas utilizadas para RetinaNet:

Primer modelo de RetinaNet, entrenado por CUDA:



Estas son sus métricas:



- **metrics/precision(B)**: La precisión del modelo es de 0.1091, lo que indica un gran número de detecciones positivas falsas.
- **metrics/recall(B)**: La recuperación es de 0.2584, bastante mayor que la precisión, pero igualmente muy baja , lo que indica bastantes falsos negativos en comparación a identificaciones correctas.
- **metrics/mAP50(B)**: Valor igual a la precisión, que indica pocas predicciones ”fáciles” realizadas correctamente.
- **metrics/mAP50-95(B)**: Valor de 0.0265, medida extremadamente pobre para casos más complejos.

Segundo modelo de RetinaNet, entrenado por CUDA:

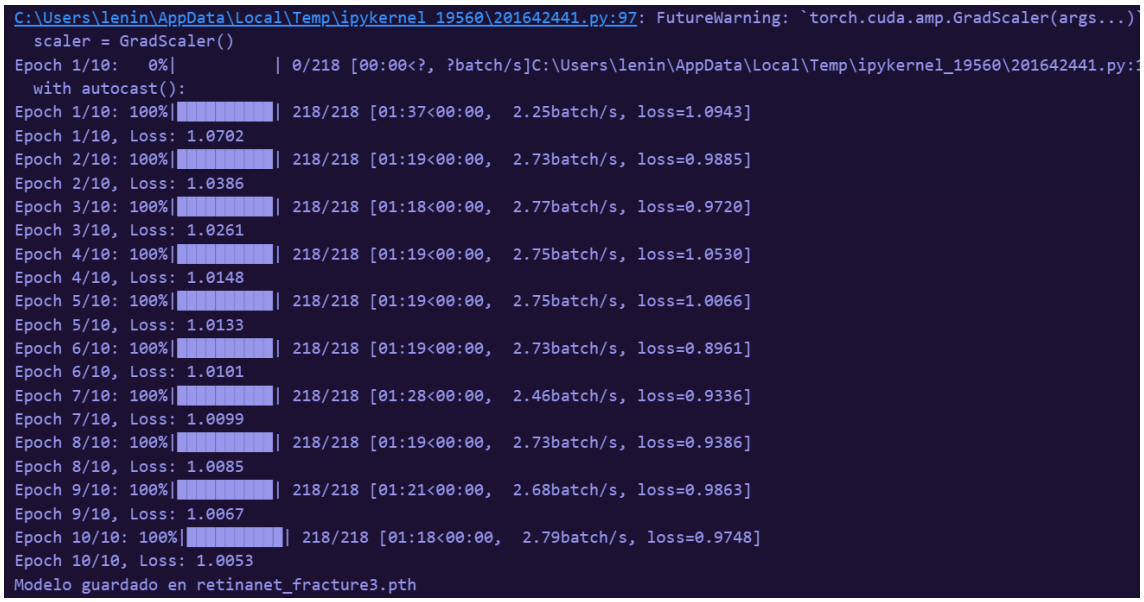
```
C:\Users\lenin\AppData\Local\Temp\ipykernel_19560\3862803722.py:97: FutureWarning: `torch.cuda.amp.GradScaler(args...)`
  scaler = GradScaler()
Epoch 1/10:   0%|          | 0/104 [00:00<?, ?batch/s]C:\Users\lenin\AppData\Local\Temp\ipykernel_19560\3862803722.py:1
  with autocast():
Epoch 1/10: 100%|██████████| 104/104 [00:46<00:00,  2.24batch/s, loss=1.0943]
Epoch 1/10, Loss: 1.1795
Epoch 2/10: 100%|██████████| 104/104 [00:43<00:00,  2.40batch/s, loss=1.0811]
Epoch 2/10, Loss: 1.1386
Epoch 3/10: 100%|██████████| 104/104 [00:43<00:00,  2.37batch/s, loss=1.1351]
Epoch 3/10, Loss: 1.1291
Epoch 4/10: 100%|██████████| 104/104 [00:42<00:00,  2.43batch/s, loss=1.0893]
Epoch 4/10, Loss: 1.1269
Epoch 5/10: 100%|██████████| 104/104 [00:38<00:00,  2.73batch/s, loss=1.1568]
Epoch 5/10, Loss: 1.1357
Epoch 6/10: 100%|██████████| 104/104 [00:37<00:00,  2.76batch/s, loss=1.0586]
Epoch 6/10, Loss: 1.1173
Epoch 7/10: 100%|██████████| 104/104 [00:37<00:00,  2.75batch/s, loss=1.0849]
Epoch 7/10, Loss: 1.1244
Epoch 8/10: 100%|██████████| 104/104 [00:39<00:00,  2.62batch/s, loss=1.0799]
Epoch 8/10, Loss: 1.1144
Epoch 9/10: 100%|██████████| 104/104 [00:38<00:00,  2.72batch/s, loss=1.0704]
Epoch 9/10, Loss: 1.1133
Epoch 10/10: 100%|██████████| 104/104 [00:38<00:00,  2.67batch/s, loss=1.1398]
Epoch 10/10, Loss: 1.1106
Modelo guardado en retinanet_fracture2.pth
```

Estas son sus métricas:

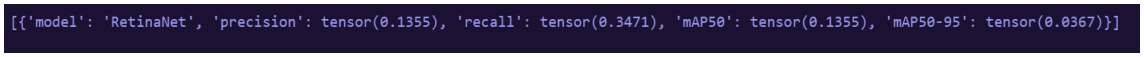
```
[{'model': 'RetinaNet', 'precision': tensor(0.0125), 'recall': tensor(0.1733), 'mAP50': tensor(0.0125), 'mAP50-95': tensor(0.0035)}]
```

- **metrics/precision(B)**: La precisión del modelo es de 0.0125, lo que indica una mayoría absoluta de detecciones positivas falsas. **metrics/recall(B)**: La recuperación es de 0.1733, medida menor que en primer modelo, pero mucho mayor su propia precisión, lo que indica muchos falsos negativos en comparación a identificaciones correctas.
- **metrics/mAP50(B)**: Valor igual a la precisión otra vez, mismas conclusiones.
- **metrics/mAP50-95(B)**: Valor de 0.0035, medida ínfima para los casos más complejos, lo que indica que no sabe identificar en lo absoluto fracturas más sutiles.

Tercer modelo de RetinaNet, entrenado por CUDA:



Estas son sus métricas:



- **metrics/precision(B)**: Precisión de 0.1355, igualmente muy baja pero mejor que en los modelos anteriores.
- **metrics/recall(B)**: La recuperación es de 0.3471,mucho mayor que en los modelos anteriores , lo que indica muchos falsos negativos, pero bastantes menos que en las otras versiones.
- **metrics/mAP50(B)**: Valor igual a la precisión, identifica mejor los casos de detección fáciles, pero sigue siendo muy bajo.
- **metrics/mAP50-95(B)**: Valor de 0.0367, medida muy baja para los casos más complejos, pero mejor que los anteriores.

Cabe destacar que, mientras intentábamos solucionar este código, se estuvo desarrollando una versión en paralelo con la implementación de RetinaNet en Keras2. Por desgracia el modelo que implementamos está incompleto y no pudimos importar el paquete de RetinaNet de Keras que estábamos esperando utilizar.

Conclusiones y propuestas de ampliación

Conclusiones

Tras estudiar ambos modelos y sus diferentes desempeños podemos llegar a las siguientes conclusiones:

YOLO ha tenido un desempeño general mucho mejor que RetinaNet en los 3 datasets estudiados. Aunque esto pudiera significar que YOLO sea mejor para la identificación de fracturas, creemos que esta diferencia en su efectividad a la

hora de detectar fracturas radica más en que el conocimiento de las tecnologías requeridas para usar RetinaNet por parte de ambos miembros del equipo era bajo de entrada, hecho que en YOLO no ocurrió, ya que se ha estado trabajando con este modelo de detección a lo largo de toda la asignatura. Por ello, en RetinaNet hemos tenido muchas dificultades a la hora de entrenar correctamente el modelo, y no se han conseguido corregir todos los fallos, por lo que creemos que su efectividad general es mayor que la demostrada en el transcurso de este trabajo.

En cuanto a los datasets utilizados podemos deducir varias teorías:

En ambos modelos de detección, el dataset con mejores métricas ha sido el tercero En cambio, el segundo es que tiene peores métricas con mucha diferencia, especialmente en RetinaNet. El tercero en cuanto a medidas está en medio de los otros 2. Creemos que esto ocurre por la calidad de las imágenes, ya que el tercer dataset tiene imágenes con mayor contraste, que permite a los modelos detectar con mayor facilidad las fracturas. En cambio en los otros 2 datasets el contraste es menor, con lo que sería más difícil detectar según qué fracturas. En cuanto a las diferencias entre el primero y el segundo, hemos observado que las imágenes del segundo dataset están cortadas en algunos puntos, con un tratamiento más pobre, por lo que si tuviéramos que recomendar un dataset sería el tercero sin lugar a duda.

Propuestas de ampliación

1. **Evaluación en datos clínicos reales:**
   Ampliar el análisis utilizando imágenes de huesos provenientes de casos clínicos reales, considerando variables como ruido en las imágenes y variabilidad en las condiciones de captura.

2. **Unificar los dataset para tener imágenes generalizadas:**
   Unificando los dataset con los que se han trabajado, puede generar un dataset más general y diversos y al entrenar un modelo con él puede generar mejores precisiones.

3. **Exploración de otros modelos:**
   Incluir en el análisis otros modelos avanzados, como Faster R-CNN o DETR, para comparar su desempeño con YOLO y RetinaNet.

4. **Ampliación del conjunto de métricas:**
   Incluir métricas adicionales, como tiempo de inferencia por imagen y consumo de recursos, para una evaluación más integral que contemple también la viabilidad computacional.

5. **Incluir la evolución de gráficas cuando se entrena en RetinaNet**: Incluir estas métricas pueden hacer ver al usuario visualmente que modelo es mejor y hacer entender el por qué funciona un modelo mejor que otro con más detalle.

Diario de reuniones del grupo

Fecha	Horas de reunión
28 de noviembre	15:00 - 17:00
5 de diciembre	15:00 - 17:00
12 de diciembre	15:00 - 17:00
19 de diciembre	15:00 - 17:00
30 de diciembre	16:00 - 20:00
2 de enero	16:00 - 20:00
3 de enero	16:00 - 20:00
4 de enero	16:00 - 21:00
7 de enero	16:00 - 21:00
8 de enero	15:00 - 17:00

Enlace al código fuente

https://github.com/ElenaaNavarroo/Vision-por-Computador/tree/main/Trabajo_final

Fuentes y tecnologías utilizadas

[1] Roboflow, "Dataset de Anotaciones YOLO11 y COCO," [En línea]. Disponible en: https://universe.roboflow.com/uet-taxila-2fk6f/bone-fracture-28rd8/dataset/1. [Accedido: 19/12/2024].

[2] Roboflow, "Dataset 2," [En línea]. Disponible en: https://universe.roboflow.com/inteligencia-computacional-e1utb/bone-only-fracture-detection. [Accedido: 19/12/2024].

[3] Roboflow, "Dataset 3," [En línea]. Disponible en: https://universe.roboflow.com/bonez/wrist-junk. [Accedido: 19/12/2024].

[4] Otsedom, "Visión por Computador - Trabajo," [En línea]. Disponible en: https://otsedom.github.io/VC/Trabajo/. [Accedido: 24/11/2024].
[5] PyTorch, “RetinaNet” [En línea]. Disponible en: https://pytorch.org/vision/main/models/retinanet.html. [Accedido: 03/01/2025].
[6] PyTorch, “TorchVision Object Detection Finetuning Tutorial” [En línea]. Disponible en: https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html. [Accedido: 03/01/2025].

[7] Keras, “Object Detection with RetinaNet“ [En línea]. Disponible en: https://keras.io/examples/vision/retinanet/. [Accedido: 04/01/2025].

[8] Evan Febrianto, “How to Train Custom Object Detection Models using RetinaNet” [En línea]. Disponible en: https://medium.com/@van.evanfebrianto/how-to-train-custom-object-detection-models-using-retinanet-aeed72f5d701. [Accedido: 04/01/2025].
[9] Sovit Ranjan Rath, “Train PyTorch RetinaNet on Custom Dataset” [En línea]. Disponible en: https://debuggercafe.com/train-pytorch-retinanet-on-custom-dataset/. [Accedido: 03/01/2025].