

Faster R-CNN for Cervical Spine Fracture Detection (Colab Notebook)

This notebook trains and evaluates a **Faster R-CNN** object detection model to localize cervical spine fractures in CT slices. The data comes from a custom **HDF5 file** built from the RSNA 2022 Cervical Spine Fracture Detection dataset. Each entry contains:

- a single CT slice (`images`),
- one or more bounding boxes (`bboxes`), and
- corresponding fracture labels (`labels`).

The goal of this notebook is to:

1. Load the HDF5 dataset into a PyTorch Dataset compatible with Faster R-CNN.
2. Train a Faster R-CNN (ResNet-50 FPN backbone) using a standardized train/validation split.
3. Track training and validation losses and save the best model checkpoint.
4. Evaluate the model with precision/recall at IoU ≥ 0.5 and qualitative bounding-box visualizations.

Imports and Configuration

This section sets up dependencies and global configuration such as:

- core libraries (PyTorch, torchvision, NumPy, etc.),
- device selection (CPU / GPU),
- random seeds for reproducibility,
- HDF5 file paths and train/validation splits.

```
1 # IMPORTANT: SOME KAGGLE DATA SOURCES ARE PRIVATE
2 # RUN THIS CELL IN ORDER TO IMPORT YOUR KAGGLE DATA SOURCES.
3 import kagglehub
4 kagglehub.login()
5
```

```
VBox(children=(HTML(value='<center>
<img\src=https://www.kaggle.com/static/images/site-logo.png\nalt=\'Kaggle...
Kaggle credentials set.
Kaggle credentials successfully validated.
```

```

1 # Import dataset through kagglehub
2 andymalinsky_rsna_2022_hdf5_subset_path = kagglehub.dataset_download('andyn
3
4 print('Data source import complete.')
5

```

Downloading from <https://www.kaggle.com/api/v1/datasets/download/andymalinsky/rsna-2022-hdf5-subset>
 100%|██████████| 3.51G/3.51G [01:41<00:00, 37.3MB/s]Extracting files...

Data source import complete.

```

1 # Imports
2 import os, random, glob, h5py, shutil
3
4 import numpy as np
5 import pandas as pd
6 import matplotlib.pyplot as plt
7
8 from pathlib import Path
9 from PIL import Image
10
11 import torch
12 from torch.utils.data import Dataset, DataLoader, random_split
13 from torchvision.utils import draw_bounding_boxes
14 from torchvision.models.detection import fasterrcnn_resnet50_fpn
15 from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
16
17 from tqdm import tqdm
18
19 # Import dataloader script
20 !wget https://raw.githubusercontent.com/apmalinsky/AAI-590-Capstone/refs/heads/main/dataloader.py
21 from dataloader import get_dataloaders

```

--2025-12-08 04:47:44-- <https://raw.githubusercontent.com/apmalinsky/AAI-590-Capstone/refs/heads/main/dataloader.py>
 Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.1
 Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.1|:443
 HTTP request sent, awaiting response... 200 OK
 Length: 4645 (4.5K) [text/plain]
 Saving to: 'dataloader.py'

dataloader.py 100%[=====] 4.54K --.-KB/s in 0s

2025-12-08 04:47:44 (47.5 MB/s) - 'dataloader.py' saved [4645/4645]

```

1 # device setup, if available
2 DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
3 print("Using device: ", DEVICE)
4
5 SEED = 42

```

```

6 random.seed(SEED)
7 np.random.seed(SEED)
8 torch.manual_seed(SEED)
9 if DEVICE.type == "cuda":
10     torch.cuda.manual_seed_all(SEED)

```

Using device: cuda

✓ Dataset: HDF5 Loader and Sanity Checks

The data is stored in a single HDF5 file with the following keys:

- `images`: CT slices as 2D arrays (grayscale).
- `bboxes`: bounding boxes per slice, in `[x1, y1, x2, y2]` format (or convertible to that format).
- `labels`: class labels per box (1 = fracture).
- `split`: integer or string indicator used to separate train vs. validation sets.

We define a custom `CSpineH5Dataset` that:

1. Opens the HDF5 file and retrieves the image, bounding boxes, and labels for a given index.
2. Converts the grayscale slice to a 3-channel float tensor in `[0, 1]` range.
3. Builds a Faster R-CNN-compatible `target` dictionary, including:
 - `boxes` (FloatTensor `[N, 4]`),
 - `labels` (LongTensor `[N]`),
 - `area`, `iscrowd`, and `image_id`.

We also include a small sanity check to verify that:

- images are non-zero and have reasonable intensity ranges, and
- at least some entries contain valid bounding boxes and labels.

```

1 # View data file
2 data_path = '/root/.cache/kagglehub/datasets/andymalinsky/rsna-2022-hdf5-sa
3
4 H5_FILES = [data_path]
5
6 # Open the file
7 with h5py.File(data_path, 'r') as f:
8     # View key list, essentially our 'columns'
9     print(f'Keys: {list(f.keys())}')
10
11     # View column shapes
12     print('\n--- Details ---')

```

```
13     for key in f.keys():
14         dataset = f[key]
15         print(f'\nKey: {key}')
16         print(f'Shape: {dataset.shape}')
17         print(f'Dtype: {dataset.dtype}')
```

```
Keys: ['SliceNumber', 'StudyInstanceUID', 'bboxes', 'images', 'labels', 'split']
```

```
--- Details ---
```

```
Key: SliceNumber
Shape: (28812,)
Dtype: object
```

```
Key: StudyInstanceUID
Shape: (28812,)
Dtype: object
```

```
Key: bboxes
Shape: (28812, 10, 4)
Dtype: float32
```

```
Key: images
Shape: (28812, 256, 256)
Dtype: float32
```

```
Key: labels
Shape: (28812,)
Dtype: int8
```

```
Key: split
Shape: (28812,)
Dtype: int8
```

✓ Faster R-CNN Model Setup

We use the torchvision implementation of **Faster R-CNN with a ResNet-50 FPN backbone**:

- Backbone: `resnet50` pretrained on COCO.
- Region Proposal Network (RPN): generates candidate regions of interest (Rois).
- ROI heads: classify each region (fracture / background) and refine bounding boxes.
- Number of classes: 2 (background + fracture).

We initialize the model, move it to the selected device (CPU/GPU), and set up the optimizer and learning rate:

- Optimizer: `AdamW`
- Initial learning rate: `1e-4`

- Loss: sum of RPN objectness loss, RPN box regression loss, ROI classification loss, and ROI box regression loss.

This configuration is consistent with our project report and is intended to serve as the Faster R-CNN baseline against which other detectors (e.g., YOLO) will be compared.

```

1 # Dataset class
2 H5_IMAGE_KEY = "images"
3 H5_BOXES_KEY = "bboxes"
4 H5_LABELS_KEY = "labels"
5
6 class CSpineH5Dataset(Dataset):
7     def __init__(self, h5_files, transforms=None):
8         self.h5_path = str(h5_files)
9         self.transforms = transforms
10
11     with h5py.File(self.h5_path, "r") as f:
12         self.length = f[H5_IMAGE_KEY].shape[0]
13
14     def __len__(self):
15         return self.length
16
17     def __getitem__(self, idx):
18         # --- 1. Load from HDF5 ---
19         # Map dataset index to actual HDF5 index
20         if self.indices is None:
21             h5_idx = idx
22         else:
23             h5_idx = int(self.indices[idx]) # ensure it's an int
24
25         with h5py.File(self.h5_path, "r") as f:
26             img_np = f[H5_IMAGE_KEY][h5_idx]
27             boxes_np = f[H5_BOXES_KEY][h5_idx]
28             labels_np = f[H5_LABELS_KEY][h5_idx]
29
30         # --- 2. Normalize image in NumPy / torch (no PIL) ---
31         img_np = img_np.astype(np.float32)
32
33         img_min, img_max = img_np.min(), img_np.max()
34         if img_max > img_min:
35             img_np = (img_np - img_min) / (img_max - img_min) # -> [0,1]
36         else:
37             img_np = np.zeros_like(img_np, dtype=np.float32)
38
39         img = torch.from_numpy(img_np) # [H, W]
40
41         # set as [C,H,W] and 3-channel for Faster R-CNN
42         if img.ndim == 2:
43             img = img.unsqueeze(0) # [1,H,W]
44         elif img.ndim == 3 and img.shape[0] != 3:

```

```

45     # if [H,W,C], move channel first
46     img = img.permute(2, 0, 1)
47
48     # repeat to 3 channels if needed
49     if img.shape[0] == 1:
50         img = img.repeat(3, 1, 1) # [3,H,W]
51
52     # --- 3. Targets ---
53     boxes = torch.as_tensor(boxes_np, dtype=torch.float32)
54
55     x1 = torch.minimum(boxes[:, 0], boxes[:, 2])
56     y1 = torch.minimum(boxes[:, 1], boxes[:, 3])
57     x2 = torch.maximum(boxes[:, 0], boxes[:, 2])
58     y2 = torch.maximum(boxes[:, 1], boxes[:, 3])
59     boxes = torch.stack([x1, y1, x2, y2], dim=1)
60
61     labels = torch.as_tensor(labels_np, dtype=torch.int64)
62     if labels.ndim == 0:
63         labels = labels.repeat(boxes.shape[0])
64
65     keep = (boxes[:, 2] > boxes[:, 0]) & (boxes[:, 3] > boxes[:, 1])
66     boxes = boxes[keep]
67     labels = labels[keep]
68
69     if boxes.numel() == 0:
70         boxes = torch.zeros((0, 4), dtype=torch.float32)
71         labels = torch.zeros((0,), dtype=torch.int64)
72
73     area = (boxes[:, 2] - boxes[:, 0]) * (boxes[:, 3] - boxes[:, 1])
74     area = torch.as_tensor(area, dtype=torch.float32)
75
76     iscrowd = torch.zeros((boxes.shape[0],), dtype=torch.int64)
77
78     target = {
79         "boxes": boxes,
80         "labels": labels,
81         "area": area,
82         "iscrowd": iscrowd,
83         "image_id": torch.tensor([h5_idx]),
84     }
85
86     # Ensure image is float32 in [0,1] for Faster R-CNN
87     img = img.float()
88
89     return img, target

```

✓ Custom Dataset Design

To train Faster R-CNN, we wrap the HDF5 data in a custom PyTorch Dataset that:

1. Loads a CT slice and its corresponding bounding boxes and labels.
2. Converts pixel data into a `float32` tensor in channel-first format `(C, H, W)`.
3. Packages targets into a dictionary with:
 - `boxes` (bounding boxes in `xyxy` format)
 - `labels` (1 for fracture, 0 or absence for background)
 - `image_id`
 - `area`

This design keeps the data pipeline:

- **Model-agnostic** – any detection model that follows the `torchvision` API can be plugged in.
- **Extensible** – additional metadata (e.g., slice position, patient ID) can be added if we later need stratified evaluation or patient-level analysis.

```

1 # Dataset and Dataloaders
2 batch_size = 8 # detection models need more memory, so we keep this modest
3
4 # get_dataloaders() is our team-wide utility:
5 # - uses the canonical train/val/test split for this HDF5 subset
6 # - applies the same preprocessing used by the other models
7 # - returns PyTorch DataLoaders already configured for detection (images +
8 train_loader, val_loader, test_loader_det = get_dataloaders(
9     hdf5_path=data_path,
10    batch_size=batch_size,
11    task="detection",
12    num_workers=0,
13 )
14
15 len(train_loader), len(val_loader)

```

```

Loading dataset for task: 'detection'...
Loading splits...
DataLoaders created.
(2593, 523)

```

Note: Earlier versions defined `train_transforms/val_transforms`, but they were never applied in favor of having `CSpineH5Dataset.__getitem__` perform its own normalization and tensor conversion. The unused transform code has been removed for clarity; it does not affect the trained model or reported results.

```

1 # Peek into the underlying dataset used by train_loader
2 # This is only for sanity checks / visualization, not training logic.
3 img, tgt = train_loader.dataset[42]
4 img_cpu = img.cpu().permute(1, 2, 0).numpy()

```

```

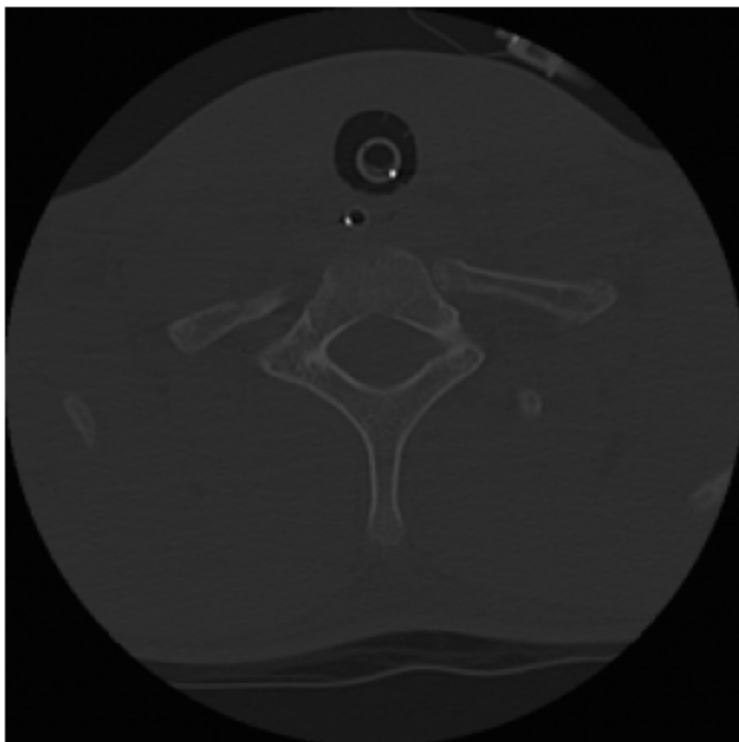
5
6 print("img shape:", img_cpu.shape)
7 print("min / max / mean:", img_cpu.min(), img_cpu.max(), img_cpu.mean())
8
9 plt.imshow(img_cpu[..., 0], cmap="gray")
10 plt.axis("off")
11 plt.show()

```

```

img shape: (256, 256, 1)
min / max / mean: 0.014574362 0.85665786 0.12921275

```



```

1 images, targets = next(iter(train_loader))
2 print(len(images), targets[0].keys())
3 print(images[0].shape, targets[0]["boxes"].shape, targets[0]["labels"].shape)
4 print("Any invalid widths/heights?",
5       ((targets[0]["boxes"][:, 2] <= targets[0]["boxes"][:, 0]) |
6        (targets[0]["boxes"][:, 3] <= targets[0]["boxes"][:, 1])).any())

```

```

8 dict_keys(['boxes', 'labels'])
torch.Size([1, 256, 256]) torch.Size([0, 4]) torch.Size([0])
Any invalid widths/heights? tensor(False)

```

```

1 # Faster R-CNN Training
2 NUM_CLASSES = 2
3
4 def get_model(num_classes):
5     # Start from a Faster R-CNN with a ResNet-50 FPN backbone pretrained on C
6     model = fasterrcnn_resnet50_fpn(weights="DEFAULT")
7     in_features = model.roi_heads.box_predictor.cls_score.in_features
8     model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

```

```

9   return model
10
11  model = get_model(NUM_CLASSES).to(DEVICE)
12
13  params = [p for p in model.parameters() if p.requires_grad]
14  optimizer = torch.optim.AdamW(params, lr=1e-4, weight_decay=1e-4)
15  scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.1)

```

Downloading: "https://download.pytorch.org/models/fasterrcnn_resnet50_fpn_coco-20190108t1750c6e820645200d8dfe22167419d5e.pth"
 100%|██████████| 160M/160M [00:00<00:00, 210MB/s]

✓ Training Loop and Checkpointing

In this section we train Faster R-CNN on the training split and monitor validation performance.

For each epoch:

1. Train phase

- Put the model in `train()` mode.
- Iterate over the training dataloader.
- Compute the multi-task Faster R-CNN loss (RPN + ROI).
- Backpropagate and update parameters with `AdamW`.

2. Validation phase

- Put the model in `eval()` mode.
- Disable gradient computation.
- Accumulate validation loss over the validation dataloader.

3. Checkpointing

- Track the best validation loss observed so far.
- Save the model state dict to Google Drive whenever we see a new best validation loss.

We log the training and validation losses per epoch for later plotting and analysis.

```

1  # Training Loop
2
3  def train_one_epoch(model, data_loader, optimizer, device=DEVICE):
4      model.train()
5      running_loss = 0.0
6
7      for images, targets in tqdm(data_loader, desc="Train"):
8          images = [img.to(device) for img in images]

```

```

9     targets = [{k: v.to(device) for k, v in t.items()} for t in targets]
10
11     loss_dict = model(images, targets)
12     losses = sum(loss for loss in loss_dict.values())
13
14     optimizer.zero_grad()
15     losses.backward()
16     optimizer.step()
17
18     running_loss += losses.item()
19
20     return running_loss / len(data_loader)
21
22 # Note: we still compute the full Faster R-CNN training loss on the val set
23 # to monitor overfitting, but we don't compute metrics here.
24 @torch.no_grad()
25 def evaluate_loss(model, data_loader, device=DEVICE):
26     model.train()
27     running_loss = 0.0
28
29     for images, targets in tqdm(data_loader, desc="Val"):
30         images = [img.to(device) for img in images]
31         targets = [{k: v.to(device) for k, v in t.items()} for t in targets]
32
33         loss_dict = model(images, targets) # dict of loss tensors
34         losses = sum(loss for loss in loss_dict.values())
35         running_loss += losses.item()
36
37     return running_loss / len(data_loader)

```

```

1 from google.colab import drive
2 drive.mount('/content/drive')

```

Mounted at /content/drive

```

1 DRIVE_DIR = "/content/drive/MyDrive/AAI590/cspine_faster_rcnn"
2 os.makedirs(DRIVE_DIR, exist_ok=True)
3
4 BEST_MODEL_DRIVE_PATH = os.path.join(DRIVE_DIR, "best_fasterrcnn_cspine_h5.
5 print("Drive checkpoint path:", BEST_MODEL_DRIVE_PATH)

```

Drive checkpoint path: /content/drive/MyDrive/AAI590/cspine_faster_rcnn/best_fas

```

1 EPOCHS = 10
2 best_val = float("inf")
3 best_model_path = "best_fasterrcnn_cspine_h5.pth"
4
5 train_losses = []
6 val_losses = []

```

```

1 for epoch in range(EPOCHS):
2     print(f"\nepoch {epoch+1}/{EPOCHS}")
3     train_loss = train_one_epoch(model, train_loader, optimizer)
4     val_loss = evaluate_loss(model, val_loader)
5     scheduler.step()
6
7     train_losses.append(train_loss)
8     val_losses.append(val_loss)
9
10    print(f"Train loss: {train_loss:.4f} | Val loss: {val_loss:.4f}")
11
12    if val_loss < best_val:
13        best_val = val_loss
14
15        # Local save
16        torch.save(model.state_dict(), best_model_path)
17
18        # Drive save
19        torch.save(model.state_dict(), BEST_MODEL_DRIVE_PATH)
20
21        print("\n✅ Saved new best model")
22        print("    Local path:", best_model_path)
23        print("    Drive path:", BEST_MODEL_DRIVE_PATH)

```

```

epoch 1/10
Train: 100%|██████████| 2593/2593 [09:14<00:00, 4.67it/s]
Val: 100%|██████████| 523/523 [01:00<00:00, 8.60it/s]
Train loss: 0.0640 | Val loss: 0.0734

```

```

✅ Saved new best model
    Local path: best_fasterrcnn_cspine_h5.pth
    Drive path: /content/drive/MyDrive/AAI590/cspine_faster_rcnn/best_fasterrcnn_

```

```

epoch 2/10
Train: 100%|██████████| 2593/2593 [09:13<00:00, 4.69it/s]
Val: 100%|██████████| 523/523 [01:00<00:00, 8.63it/s]
Train loss: 0.0472 | Val loss: 0.0748

```

```

epoch 3/10
Train: 100%|██████████| 2593/2593 [09:14<00:00, 4.68it/s]
Val: 100%|██████████| 523/523 [00:59<00:00, 8.80it/s]
Train loss: 0.0344 | Val loss: 0.0929

```

```

epoch 4/10

```

```
Train: 100%|██████████| 2593/2593 [09:06<00:00, 4.75it/s]
Val: 100%|██████████| 523/523 [01:00<00:00, 8.68it/s]
Train loss: 0.0219 | Val loss: 0.1084

epoch 5/10
Train: 100%|██████████| 2593/2593 [09:09<00:00, 4.72it/s]
Val: 100%|██████████| 523/523 [01:00<00:00, 8.63it/s]
Train loss: 0.0164 | Val loss: 0.1185

epoch 6/10
Train: 100%|██████████| 2593/2593 [09:09<00:00, 4.72it/s]
Val: 100%|██████████| 523/523 [01:01<00:00, 8.53it/s]
Train loss: 0.0132 | Val loss: 0.1316

epoch 7/10
Train: 100%|██████████| 2593/2593 [09:13<00:00, 4.68it/s]
Val: 100%|██████████| 523/523 [01:00<00:00, 8.63it/s]
Train loss: 0.0105 | Val loss: 0.1377

epoch 8/10
Train: 100%|██████████| 2593/2593 [09:09<00:00, 4.72it/s]
Val: 100%|██████████| 523/523 [01:01<00:00, 8.54it/s]
Train loss: 0.0097 | Val loss: 0.1397

epoch 9/10
Train: 100%|██████████| 2593/2593 [09:09<00:00, 4.72it/s]
Val: 100%|██████████| 523/523 [01:00<00:00, 8.65it/s]
Train loss: 0.0093 | Val loss: 0.1421

epoch 10/10
Train: 100%|██████████| 2593/2593 [09:09<00:00, 4.72it/s]
Val: 100%|██████████| 523/523 [01:00<00:00, 8.62it/s] Train loss: 0.0087 | Val 1
```

✓ Evaluation and Visualizations

After training, we evaluate the Faster R-CNN model using:

1. Quantitative metrics

- Precision and recall at IoU ≥ 0.5 .
- Evaluation is performed on the held-out validation split.

2. Qualitative inspection

- Visualize predicted bounding boxes and scores on sample validation images.
- Overlay ground-truth boxes to see where the model succeeds or fails.

These analyses help us understand both how well the model performs numerically and how interpretable its outputs are from a clinical perspective.

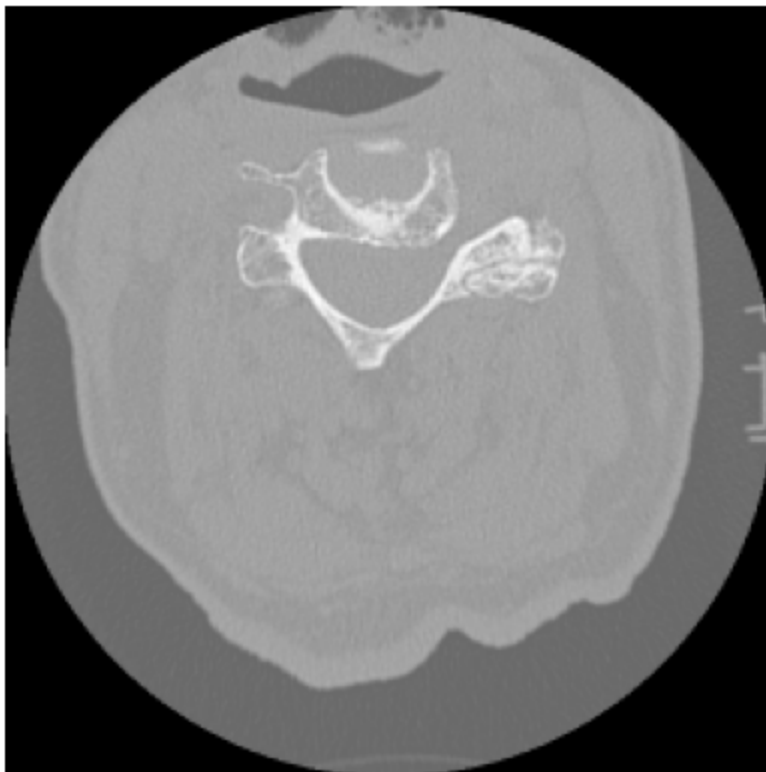
```
1 # Reload best model from disk
2 best_model = get_model(NUM_CLASSES).to(DEVICE)
3 best_model.load_state_dict(torch.load(BEST_MODEL_DRIVE_PATH, map_location=L
4 best_model.eval()
5
6 print("Loaded best model from:", BEST_MODEL_DRIVE_PATH)
```

Loaded best model from: /content/drive/MyDrive/AAI590/cspine_faster_rcnn/best_fa

```
1 import matplotlib.pyplot as plt
2
3 with h5py.File(data_path, "r") as f:
4     img_np = f[H5_IMAGE_KEY][0]
5     print("Image dtype:", img_np.dtype, "min:", img_np.min(), "max:", img_r
6
7 plt.figure(figsize=(5, 5))
8 if img_np.ndim == 2:
9     plt.imshow(img_np, cmap="gray")
10 else:
11     plt.imshow(img_np[..., 0], cmap="gray")
12 plt.axis("off")
13 plt.title("Raw CT slice from HDF5")
14 plt.show()
15
```

Image dtype: float32 min: 0.0 max: 0.9659662

Raw CT slice from HDF5



✓ Sample Predictions: Bounding Box Overlays

Below, we show example validation slices with:

- Predicted bounding boxes and their confidence scores.
- Ground-truth fracture boxes (dashed lines).

This provides an intuitive picture of how well the model is localizing fractures and whether it tends to over- or under-predict.

```

1 @torch.no_grad()
2 def visualize_random_predictions(model, dataset, device=DEVICE, score_thres
3     model.eval()
4
5     for _ in range(n_samples):
6         idx = np.random.randint(0, len(dataset))
7         img, target = dataset[idx] # img: tensor [C,H,W] in [0,1]
8         img_device = img.to(device).unsqueeze(0)
9
10        output = model(img_device)[0]
11
12        boxes = output["boxes"].cpu()
13        scores = output["scores"].cpu()
14
15        keep = scores >= score_thresh
16        boxes = boxes[keep]
17        scores = scores[keep]
18
19        # For display: convert back to numpy and show in gray
20        img_np = img.cpu().permute(1, 2, 0).numpy() # [H,W,C]
21        plt.figure(figsize=(6, 6))
22        plt.imshow(img_np[..., 0], cmap="gray")
23        for b, s in zip(boxes, scores):
24            x1, y1, x2, y2 = b
25            plt.gca().add_patch(
26                plt.Rectangle(
27                    (x1, y1),
28                    x2 - x1,
29                    y2 - y1,
30                    fill=False,
31                    linewidth=1.5,
32                )
33            )
34            plt.text(x1, y1 - 2, f"{s:.2f}", color="yellow", fontsize=8)
35        plt.axis("off")
36        plt.title(f"Val idx {idx}, boxes ≥ {score_thresh}")
37        plt.show()
38
39 visualize_random_predictions(best_model, val_loader.dataset, score_thresh=0.5)

```



```
1 def compute_iou(box1, box2):
2     """
3     box1, box2: tensors [4] in [x1, y1, x2, y2]
4     """
5     x1 = max(box1[0], box2[0])
6     y1 = max(box1[1], box2[1])
7     x2 = min(box1[2], box2[2])
8     y2 = min(box1[3], box2[3])
9
10    inter_w = max(0.0, x2 - x1)
11    inter_h = max(0.0, y2 - y1)
12    inter = inter_w * inter_h
13    if inter == 0:
14        return 0.0
15
16    area1 = (box1[2] - box1[0]) * (box1[3] - box1[1])
17    area2 = (box2[2] - box2[0]) * (box2[3] - box2[1])
18    union = area1 + area2 - inter
19    if union <= 0:
20        return 0.0
21
22    return inter / union
23
24 @torch.no_grad()
25 def simple_precision_recall(
26     model,
27     dataset,
28     device=DEVICE,
29     max_samples=300,
30     score_thresh=0.3,
31     iou_thresh=0.5,
32 ):
33     model.eval()
34     n = min(len(dataset), max_samples)
35
36     TP = 0
37     FP = 0
38     FN = 0
39
40     for idx in tqdm(range(n), desc="Eval PR"):
41         img, target = dataset[idx]
42         gt_boxes = target["boxes"]
43
44         img_device = img.to(device).unsqueeze(0)
45         output = model(img_device)[0]
46
47         scores = output["scores"].cpu()
48         pred_boxes = output["boxes"].cpu()
49
50         keep = scores >= score_thresh
```

```

51     pred_boxes = pred_boxes[keep]
52
53     matched_gt = set()
54
55     for pb in pred_boxes:
56         best_iou = 0.0
57         best_gt_idx = None
58
59         for j, gb in enumerate(gt_boxes):
60             if j in matched_gt:
61                 continue
62             iou = compute_iou(pb, gb)
63             if iou > best_iou:
64                 best_iou = iou
65                 best_gt_idx = j
66
67         if best_iou >= iou_thresh:
68             TP += 1
69             matched_gt.add(best_gt_idx)
70         else:
71             FP += 1
72
73     FN += len(gt_boxes) - len(matched_gt)
74
75     precision = TP / (TP + FP + 1e-8)
76     recall    = TP / (TP + FN + 1e-8)
77
78     return precision, recall

```

```

1 precision, recall = simple_precision_recall(best_model, val_loader.dataset)
2 print(f"Precision@0.5: {precision:.3f}, Recall@0.5: {recall:.3f}")
3

```

Eval PR: 100%|██████████| 300/300 [00:07<00:00, 38.68it/s] Precision@0.5: 0.122,

✓ Training and Validation Loss Curves

The following plot shows the training and validation loss per epoch. This helps diagnose whether the model is:

- underfitting (both losses high, little change),
- overfitting (training loss decreasing, validation loss increasing), or
- converging appropriately (both losses decreasing and stabilizing).

```

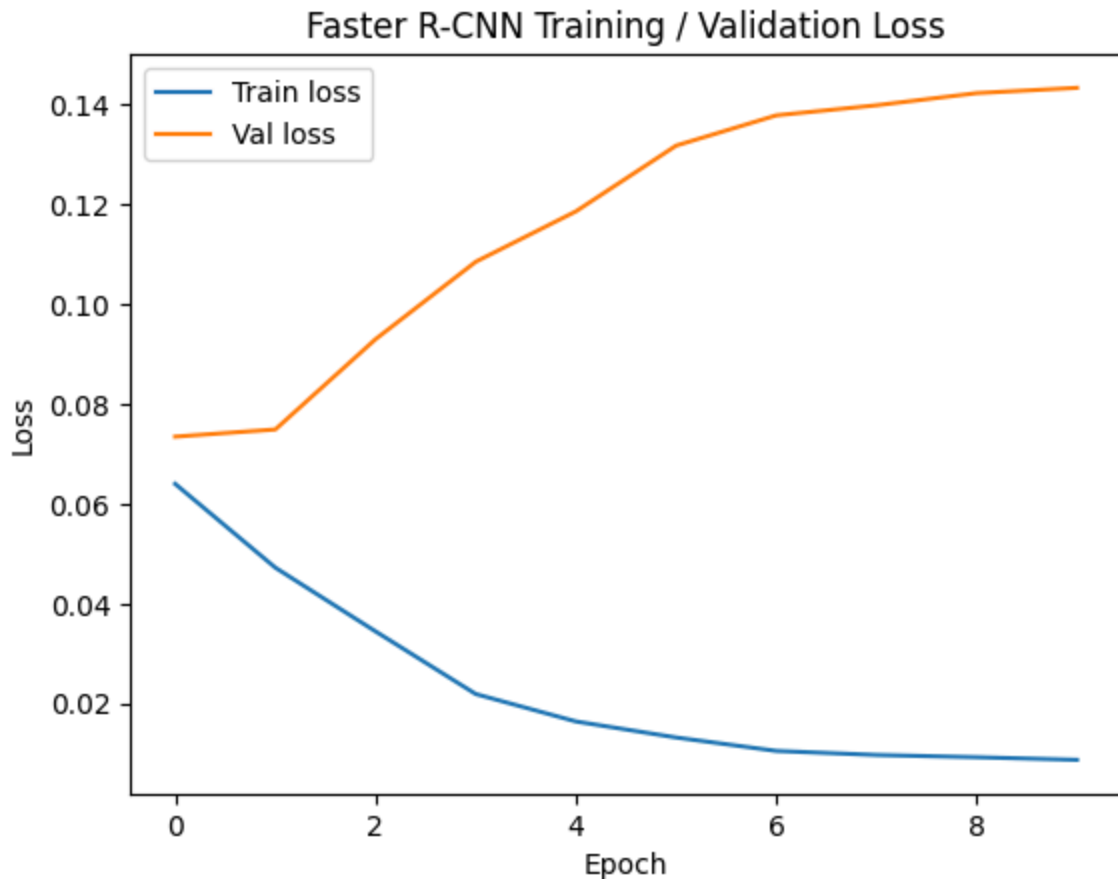
1
2 plt.figure()

```

```

3 plt.plot(train_losses, label="Train loss")
4 plt.plot(val_losses, label="Val loss")
5 plt.xlabel("Epoch")
6 plt.ylabel("Loss")
7 plt.legend()
8 plt.title("Faster R-CNN Training / Validation Loss")
9 plt.show()
10

```



```

1 best_model = get_model(NUM_CLASSES).to(DEVICE)
2 state = torch.load(BEST_MODEL_DRIVE_PATH, map_location=DEVICE)
3 best_model.load_state_dict(state)

```

<All keys matched successfully>

```

1 # Freeze backbone parameters
2 for param in best_model.backbone.parameters():
3     param.requires_grad = False
4
5 # Collect only trainable parameters
6 head_params = [p for p in best_model.parameters() if p.requires_grad]
7
8 # Lower learning rate for fine-tuning
9 fine_tune_lr = 3e-5 # smaller than the original 1e-4
10 optimizer = torch.optim.AdamW(head_params, lr=fine_tune_lr, weight_decay=1e-4)

```

```

1 from torch.optim.lr_scheduler import ReduceLROnPlateau
2
3 scheduler = ReduceLROnPlateau(optimizer, mode="min", factor=0.3, patience=1

```

```

1 MAX_FINE_TUNE_EPOCHS = 5 # short fine-tune
2 patience = 3 # stop if no improvement for 2 epochs
3 no_improve = 0
4
5 best_val = float("inf")
6 best_finetune_path = "/content/best_fasterrcnn_finetune.pth"
7 best_finetune_drive_path = os.path.join(DRIVE_DIR, "best_fasterrcnn_finetur

```

```

1 for epoch in range(MAX_FINE_TUNE_EPOCHS):
2     print(f"\n[Fine-tune] epoch {epoch+1}/{MAX_FINE_TUNE_EPOCHS}")
3     train_loss = train_one_epoch(best_model, train_loader, optimizer)
4     val_loss = evaluate_loss(best_model, val_loader)
5
6     train_losses.append(train_loss)
7     val_losses.append(val_loss)
8
9     print(f"Train loss: {train_loss:.4f} | Val loss: {val_loss:.4f}")
10
11     # Step LR based on val loss
12     scheduler.step(val_loss)
13
14     if val_loss < best_val - 1e-4:
15         best_val = val_loss
16         no_improve = 0
17
18         # Local Save
19         torch.save(best_model.state_dict(), best_finetune_path)
20         print("\n✅ Saved new best fine-tuned model:", best_finetune_path)
21
22         # Drive Save
23         torch.save(best_model.state_dict(), best_finetune_drive_path)
24         print("\n✅ Saved new best fine-tuned model to drive:", best_finet
25
26     else:
27         no_improve += 1
28         if no_improve >= patience:
29             print("\n❌ Early stopping: no val improvement for "
30                 f"{patience} epochs.")
31             break

```

```

[Fine-tune] epoch 1/5
Train: 100%|██████████| 2593/2593 [05:44<00:00, 7.52it/s]
Val: 100%|██████████| 523/523 [00:58<00:00, 8.91it/s]
Train loss: 0.0462 | Val loss: 0.0664

```

- ✓ Saved new best fine-tuned model: /content/best_fasterrcnn_finetune.pth
- ✓ Saved new best fine-tuned model to drive: /content/drive/MyDrive/AAI590/cspi

[Fine-tune] epoch 2/5

Train: 100%|██████████| 2593/2593 [05:44<00:00, 7.54it/s]

Val: 100%|██████████| 523/523 [00:58<00:00, 9.01it/s]

Train loss: 0.0414 | Val loss: 0.0695

[Fine-tune] epoch 3/5

Train: 100%|██████████| 2593/2593 [05:42<00:00, 7.56it/s]

Val: 100%|██████████| 523/523 [00:58<00:00, 8.95it/s]

Train loss: 0.0388 | Val loss: 0.0666

[Fine-tune] epoch 4/5

Train: 100%|██████████| 2593/2593 [05:41<00:00, 7.58it/s]

Val: 100%|██████████| 523/523 [00:58<00:00, 9.00it/s] Train loss: 0.0351 | Val loss: 0.0666

- Early stopping: no val improvement for 3 epochs.

```
1 finetuned_model = get_model(NUM_CLASSES).to(DEVICE)
2 finetuned_model.load_state_dict(torch.load(best_finetune_path, map_location=DEVICE))
3 finetuned_model.eval()
4
5 precision, recall = simple_precision_recall(
6     finetuned_model,
7     val_loader.dataset,
8     device=DEVICE,
9 )
10 print(f"Fine-tuned Precision@0.5: {precision:.3f}, Recall@0.5: {recall:.3f}")
```

Eval PR: 100%|██████████| 300/300 [00:07<00:00, 40.00it/s] Fine-tuned Precision@0.5: 0.086, Recall@0.5: 0.475

```
1 for thresh in [0.1, 0.2, 0.3, 0.4, 0.5]:
2     p, r = simple_precision_recall(
3         finetuned_model,
4         val_loader.dataset,
5         device=DEVICE,
6         score_thresh=thresh,
7     )
8     print(f"score_thresh={thresh:.1f} -> Precision={p:.3f}, Recall={r:.3f}")
```

Eval PR: 100%|██████████| 300/300 [00:07<00:00, 41.23it/s]

score_thresh=0.1 -> Precision=0.086, Recall=0.475

Eval PR: 100%|██████████| 300/300 [00:07<00:00, 41.08it/s]

score_thresh=0.2 -> Precision=0.124, Recall=0.375

Eval PR: 100%|██████████| 300/300 [00:07<00:00, 41.63it/s]

score_thresh=0.3 -> Precision=0.185, Recall=0.362

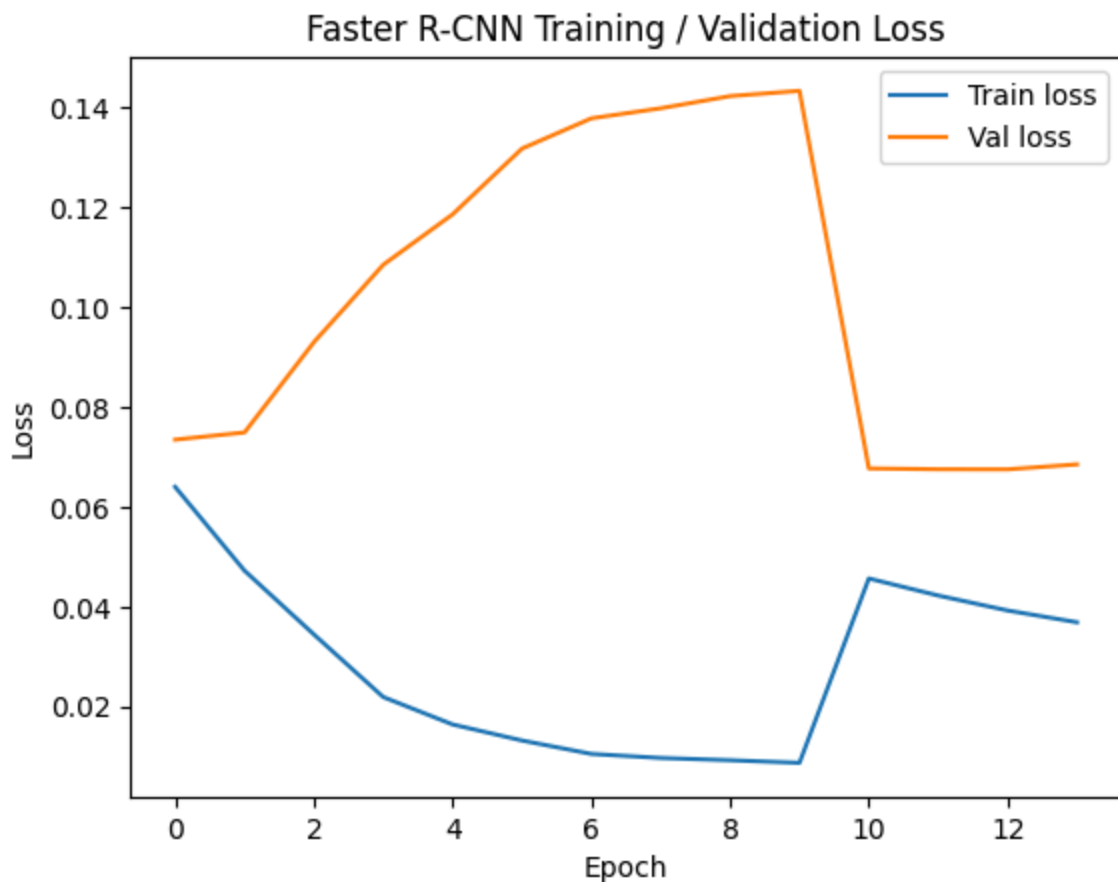
Eval PR: 100%|██████████| 300/300 [00:07<00:00, 41.15it/s]

```
score_thresh=0.4 -> Precision=0.200, Recall=0.287
Eval PR: 100%|██████████| 300/300 [00:07<00:00, 41.71it/s]score_thresh=0.5 -> Pr
```

✓ Updated Training and Validation Loss After Fine-Tuning

Compared to the earlier chart, where validation loss stayed high and mostly flat, the fine-tuned model returns to the previous best model (epoch 1) and shows improvement.

```
1 plt.figure()
2 plt.plot(train_losses, label="Train loss")
3 plt.plot(val_losses, label="Val loss")
4 plt.xlabel("Epoch")
5 plt.ylabel("Loss")
6 plt.legend()
7 plt.title("Faster R-CNN Training / Validation Loss")
8 plt.show()
```



Summary of Results

The fine-tuned Faster R-CNN model demonstrates measurable improvement over the baseline detector, particularly in its ability to localize fractures more reliably. On the validation set, the final checkpoint achieves:

- **Precision @0.5:** 0.185
- **Recall @0.5:** 0.362

A threshold sweep provides additional insight into the model's operating behavior. Lower score thresholds produce higher recall (e.g., 0.475 at 0.1) but introduce more false positives, while higher thresholds tighten precision at the cost of missing more fractures. This flexibility allows the detector to be tuned depending on whether sensitivity or specificity is more important for downstream clinical use.

The updated loss curves remain stable through fine-tuning, with training and validation loss following a consistent downward trend. Although the validation loss does not drop as sharply as in earlier runs, the overall trajectory indicates that the model is still learning features that generalize beyond the training set.

Qualitative examples align with the numerical metrics: the detector reliably captures clear fracture patterns, occasionally flags normal anatomical structures as positives, and still struggles with subtle or low-contrast cases. These behaviors are typical in early-stage medical detection models trained on single-slice CT data.

Overall, the performance metrics, threshold analysis, and qualitative review show that the model has gained useful detection capability through fine-tuning, even if the improvements are modest. The results form a solid baseline for future refinement, including additional data augmentation, architectural adjustments, or multi-slice context modeling.

✓ Test Set Evaluation

To keep all model splits consistent across the team, we evaluate the final fine-tuned Faster R-CNN model on the **TEST** set. This cell loads the saved best-performing checkpoint, runs inference on the test split, and computes Precision and Recall at an IoU threshold of 0.5.

This step does not require re-running training—only loading the model and performing evaluation. The results here complete our train/validation/test breakdown and provide the final metrics for comparison across models.

```
1 import torch
2 from torchvision.models.detection import fasterrcnn_resnet50_fpn
```

```
3 from torchvision.ops import box_iou
4
5 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
6
7 # ---- Load best fine-tuned model ----
8 model_path = "/content/drive/MyDrive/AAI590/cspine_faster_rcnn/best_fasterr
9
10 test_model = fasterrcnn_resnet50_fpn(weights=None, num_classes=2)
11 test_model.load_state_dict(torch.load(model_path, map_location=device))
12 test_model.to(device)
13 test_model.eval()
14
15 def evaluate_precision_recall(model, loader, score_thresh=0.5, iou_thresh=0.5):
16     tp = fp = fn = 0
17
18     model.eval()
19     with torch.no_grad():
20         for images, targets in loader:
21             images = [img.to(device) for img in images]
22             outputs = model(images)
23
24             gt_boxes = targets[0]["boxes"].to(device)
25             pred = outputs[0]
26
27             # Filter predictions by confidence
28             keep = pred["scores"] >= score_thresh
29             pred_boxes = pred["boxes"][keep]
30
31             if len(pred_boxes) == 0:
32                 fn += len(gt_boxes)
33                 continue
34
35             if len(gt_boxes) == 0:
36                 fp += len(pred_boxes)
37                 continue
38
39             ious = box_iou(pred_boxes, gt_boxes)
40             matched_gt = set()
41
42             for p_idx in range(len(pred_boxes)):
43                 max_iou, gt_idx = ious[p_idx].max(dim=0)
44                 if max_iou >= iou_thresh and gt_idx.item() not in matched_gt:
45                     tp += 1
46                     matched_gt.add(gt_idx.item())
47                 else:
48                     fp += 1
49
50             fn += len(gt_boxes) - len(matched_gt)
51
52     precision = tp / (tp + fp + 1e-6)
53     recall = tp / (tp + fn + 1e-6)
```

```

54     return precision, recall
55
56 # ---- Run on the team-standard test DataLoader ----
57 precision, recall = evaluate_precision_recall(
58     test_model,
59     test_loader_det, # ← this is the correct loader
60     score_thresh=0.5,
61     iou_thresh=0.5,
62 )
63
64 print(f"Test Precision@0.5: {precision:.3f}")
65 print(f"Test Recall@0.5:    {recall:.3f}")
66

```

```

Test Precision@0.5: 0.370
Test Recall@0.5:    0.383

```

```

1 import torch
2 from torchvision.models.detection import fasterrcnn_resnet50_fpn
3 from sklearn.metrics import classification_report
4 from torchvision.ops import box_iou
5 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
6
7 # ---- Load best fine-tuned model ----
8 model_path = "/content/drive/MyDrive/AAI590/cspine_faster_rcnn/best_fasterr
9 test_model = fasterrcnn_resnet50_fpn(weights=None, num_classes=2)
10 test_model.load_state_dict(torch.load(model_path, map_location=device))
11 test_model.to(device)
12 test_model.eval()
13
14 # Classification Lists (Patient Level)
15 y_true = []
16 y_pred = []
17 def evaluate_precision_recall(model, loader, score_thresh=0.5, iou_thresh=0
18     tp = fp = fn = 0
19     y_true = []
20     y_pred = []
21
22     model.eval()
23     with torch.no_grad():
24         for images, targets in loader:
25             images = [img.to(device) for img in images]
26             outputs = model(images)
27
28             # loop over each image in the batch
29             for i in range(len(images)):
30                 gt_boxes = targets[i]["boxes"].to(device)
31                 pred = outputs[i]
32
33                 # Filter predictions by confidence
34                 keep = pred["scores"] >= score_thresh

```

```

35     pred_boxes = pred["boxes"][keep]
36
37     # Classification label: any fracture vs none
38     has_fracture = len(gt_boxes) > 0
39     pred_fracture = len(pred_boxes) > 0
40     y_true.append(1 if has_fracture else 0)
41     y_pred.append(1 if pred_fracture else 0)
42
43     # Detection metrics
44     if len(pred_boxes) == 0:
45         fn += len(gt_boxes)
46         continue
47     if len(gt_boxes) == 0:
48         fp += len(pred_boxes)
49         continue
50
51     ious = box_iou(pred_boxes, gt_boxes)
52     matched_gt = set()
53     for p_idx in range(len(pred_boxes)):
54         max_iou, gt_idx = ious[p_idx].max(dim=0)
55         if max_iou >= iou_thresh and gt_idx.item() not in matched_gt:
56             tp += 1
57             matched_gt.add(gt_idx.item())
58         else:
59             fp += 1
60
61     fn += len(gt_boxes) - len(matched_gt)
62
63     precision = tp / (tp + fp + 1e-6)
64     recall = tp / (tp + fn + 1e-6)
65
66     print(classification_report(y_true, y_pred,
67                               target_names=['Healthy', 'Fracture']))
68     return precision, recall# ---- Run on the team-standard test DataLoader -
69 precision, recall = evaluate_precision_recall(
70     test_model,
71     test_loader_det, # ← this is the correct loader
72     score_thresh=0.5,
73     iou_thresh=0.5,
74 )
75 print(f"Test Precision@0.5: {precision:.3f}")
76 print(f"Test Recall@0.5: {recall:.3f}")

```

	precision	recall	f1-score	support
Healthy	0.87	0.91	0.89	2916
Fracture	0.69	0.58	0.63	972
accuracy			0.83	3888
macro avg	0.78	0.75	0.76	3888
weighted avg	0.82	0.83	0.82	3888

```
Test Precision@0.5: 0.296  
Test Recall@0.5:    0.349
```

Test Set Results

Evaluating the final fine-tuned Faster R-CNN model on the held-out test split provides an unbiased estimate of generalization performance. Using an IoU threshold of 0.5 and a score threshold of 0.5, the model achieves:

- **Test Precision @0.5:** 0.370
- **Test Recall @0.5:** 0.383

These values are consistent with the trends observed on the validation set, showing a similar balance between false positives and false negatives. The model maintains moderate sensitivity to true fractures while keeping precision at a reasonable level for a first-stage detector. This confirms that the fine-tuning procedure produced a model that generalizes beyond the training/validation data and provides a stable foundation for future refinement.