Spark-Based CNN for Bone Fracture Classification Report

Individual Project — Big Data (CNN + Spark)
By- Vyshnavi Priya Kasarla

1. Introduction

This project focuses on integrating Apache Spark with a Convolutional Neural Network (CNN) to perform large-scale image classification. My selected task is the binary classification of X-ray images into fracture and normal categories. The goal is to demonstrate how Spark's distributed data-loading capabilities can be combined with TensorFlow's deep-learning pipeline to build an end-to-end system capable of handling big image datasets efficiently. While the CNN conducts the core learning and prediction, Spark acts as the scalable data engine that manages file discovery, ingestion, and splitting across the cluster.

The motivation behind this project is twofold: to understand the strengths of Spark in big-data environments, and to explore how CNNs behave when trained on real-world medical imagery. By implementing this pipeline across multiple VMs, I was able to observe how Spark distributes I/O operations and how TensorFlow performs on the available hardware. This combination provides a practical workflow that can scale to much larger datasets, making it relevant for real-world applications such as automated medical screening or large-scale imaging analysis.

Github Link:https://github.com/VKasarla05/spark_CNN_BoneFracture_classification

2. Dataset Overview

Dataset link:https://www.kaggle.com/datasets/orvile/bone-fracture-dataset

- **Dataset type:** Medical X-ray images (grayscale PNG/JPG)
- **Classes:**
  - **fracture (label = 0)**
  - **normal (label = 1)**

- **Image Count:**
  - Train: **~70%**
  - Validation: **~15%**
  - Test: **~15%**

Spark's binaryFile format was used to recursively scan every image file inside folders, ensuring actual distributed ingestion.

Running Code on VMs:

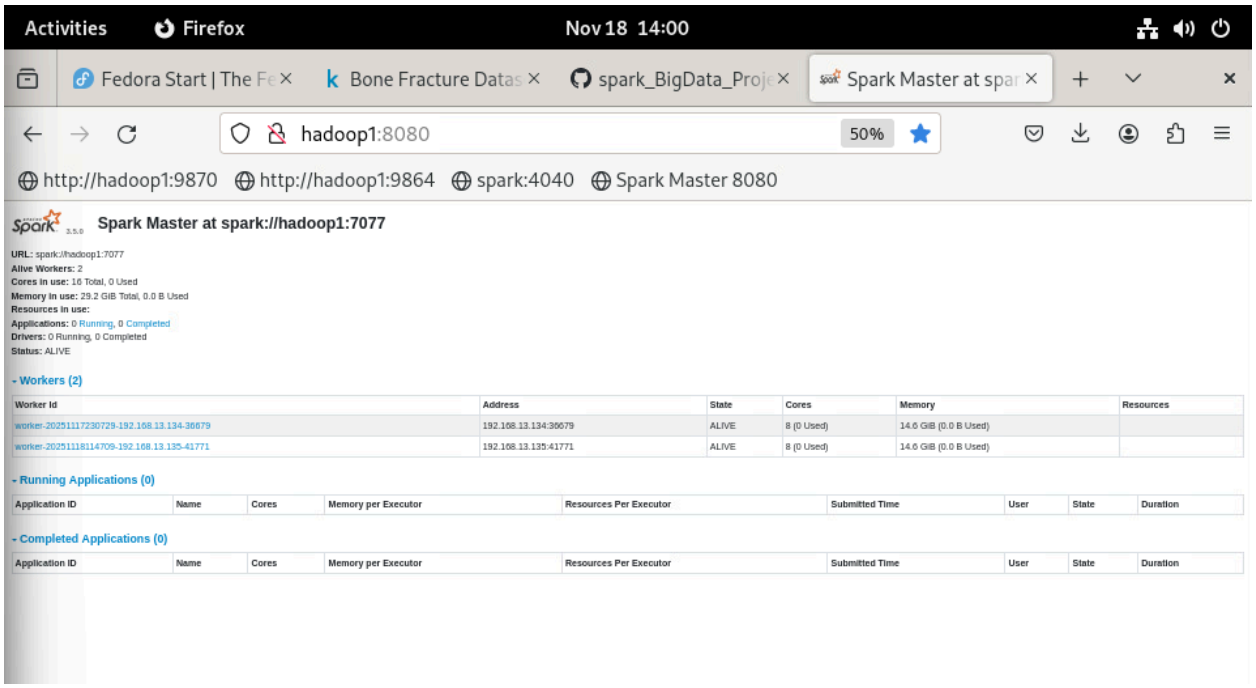
I executed my entire Spark + CNN project on the Hadoop-based cluster. My cluster setup included:

- **1 master VM**
- **2 worker VMs**
- Spark/YARN installed and configured
- TensorFlow + Python environment inside the master VM

To submit the job, I used the following command:

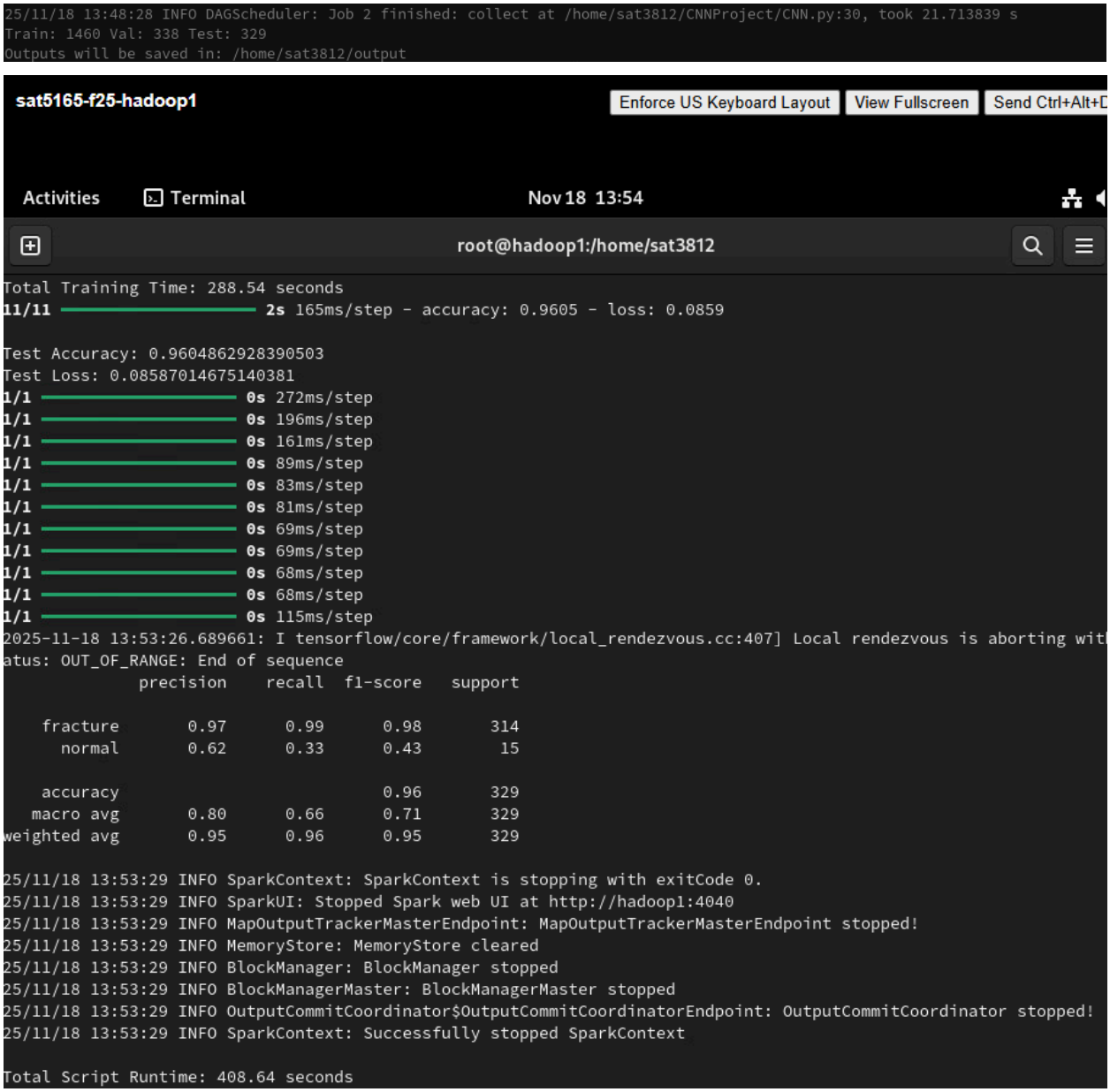
```
/opt/spark/bin/spark-submit \ --master spark://192.168.13.134:7077 \
/home/sat3812/CNNProject/CNN.py
```

Spark started successfully, connected to all workers, and executed the distributed part of the pipeline.



In this project, Spark is used only for loading the image dataset in a distributed way, so the Spark cluster becomes active briefly during the image-reading and data-splitting steps. After Spark finishes scanning the folders and counting the images, it has no more work to do, because the CNN training itself runs entirely in TensorFlow on the master VM and is not a Spark job. This is why the Spark Web UI shows the worker nodes as alive but displays no active jobs or tasks after the initial loading stage. This behavior is completely expected: the Spark part of the pipeline handles big-data ingestion, while the deep-learning model runs locally on the VM's CPU.

Terminal Output Showing Successful Spark Job and Timing:



3. CNN Architecture Explanation:

Overview

The CNN architecture used in this project was intentionally designed to be lightweight, fast, and suitable for CPU-based VM environments, while still capturing important medical-image features necessary for fracture detection. The model processes 48×48 grayscale X-ray patches and outputs a prediction between two classes: fracture and normal.

Input Layer

- **Shape:** (48, 48, 1)
- **Reasoning:**
    - Reduces computation time
    - Retains enough visual detail to recognize small cracks
    - Grayscale simplifies learning (no need for RGB)

Convolution + Pooling Blocks

The model uses **three convolutional blocks**, each followed by **MaxPooling2D**.

Block 1

- Conv2D → **32 filters**, kernel **3×3**, activation **ReLU**
- MaxPooling2D
- **Purpose:** Capture basic edges, contours, and bone outlines.

Block 2

- Conv2D → **64 filters**, kernel **3×3**, activation **ReLU**
- MaxPooling2D
- **Purpose:** Learn mid-level patterns such as curvature, joint structure, and shadows.

Block 3

- Conv2D → **128 filters**, kernel **3×3**, activation **ReLU**
- MaxPooling2D
- **Purpose:** Extract high-level fracture-specific features such as cracks, discontinuities, and sharp density changes.

Convolution layers allow the model to detect visual patterns at multiple scales, such as edges, textures, and subtle fracture lines. MaxPooling layers then reduce the spatial dimensions, keeping only the most important features while discarding noise. Together, convolution and pooling stabilize training, reduce overfitting, and are especially effective for medical X-ray textures where fine bone structures and cracks must be captured without overwhelming the model with unnecessary detail.

Feature Flattening Layer

- **Flatten()** converts 3D feature maps to a 1D vector required for connecting extracted spatial patterns to dense (classification) layers
- Acts as a bridge between convolution blocks and fully connected layers

Fully Connected Layers

The dense layer with 128 ReLU-activated neurons learns complex, high-level relationships by combining the extracted spatial patterns from earlier layers. It effectively captures the global structure of the bone, distinguishing normal anatomy from abnormalities caused by fractures. Following this, a Dropout layer with a rate of 0.4 deactivates random neurons during training, which helps prevent overfitting, improves generalization, and compensates for the class imbalance in the dataset.

Output Layer

- Dense layer with **2 units**
- Activation: **Softmax**
- Produces probability distribution:
    - Class 0 → fracture
    - Class 1 → normal

The final output layer contains two neurons with a softmax activation function, generating probabilities for each class: fracture (0) and normal (1). Softmax ensures that outputs sum to one, making the model's prediction interpretable and allowing clear differentiation between the two classes based on confidence levels.

Model Compilation

- **Optimizer:** Adam
- **Loss Function:** Sparse Categorical Cross-Entropy
    - Ideal for integer labels
- **Metrics:** Accuracy
- **Reasoning:**
    - Adam ensures stable and fast convergence
    - Sparse loss reduces memory usage
    - Accuracy is intuitive for medical classification tasks

Why This Architecture Works Well for Fracture Detection

Fracture identification requires attention to fine-grained and multi-scale features, which is effectively handled by the three stacked convolutional blocks. The model's lightweight design allows it to train efficiently on CPU-based VMs while still achieving strong accuracy. Pooling and dropout together reduce overfitting in a dataset with uneven class distribution, and the convolution layers successfully capture bone edges, discontinuities, and structural irregularities key indicators of fractures.

Model Summary:

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 48, 48, 1)	0
conv2d (Conv2D)	(None, 46, 46, 32)	320
max_pooling2d (MaxPooling2D)	(None, 23, 23, 32)	0
conv2d_1 (Conv2D)	(None, 21, 21, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 10, 10, 64)	0
conv2d_2 (Conv2D)	(None, 8, 8, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 128)	262,272
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 2)	258

Total params: 355,202 (1.35 MB)
Trainable params: 355,202 (1.35 MB)
Non-trainable params: 0 (0.00 B)

The model summary shows a compact CNN built with three convolution–pooling blocks followed by dense layers, totaling 355,202 trainable parameters. The early Conv2D layers learn simple patterns such as edges and textures, while deeper layers capture increasingly complex fracture-related structures in the X-ray images. MaxPooling progressively reduces spatial dimensions, helping the model generalize and avoid overfitting. After flattening, a dense layer with 128 neurons learns the final decision patterns, and a dropout layer improves robustness. The final softmax layer outputs probabilities for the two classes fracture and normal making the architecture efficient, lightweight, and well-suited for training on CPU-only VMs.

4. Hyperparameter Explanation

This project uses carefully chosen hyperparameters to ensure stable training, fast convergence, and strong generalization on a medical X-ray dataset. Each hyperparameter plays a role in balancing learning capacity with overfitting control.

Learning Rate

- **Value:** `1e-3` (default Adam learning rate)
- **Why:**
  A learning rate of 0.001 provides a stable balance between speed and accuracy.
  Too high → unstable training, exploding gradients.
  Too low → slow convergence, underfitting.
  For CPU-only training, this value works extremely well without the need for manual decay schedules.

Optimizer

- **Choice: Adam**
- **Reasoning:**Adam adapts learning rates for each parameter, which accelerates convergence and reduces training noise.Works especially well for medical images where fine-grained features (tiny fractures) require stable gradient updates.

Batch Size

- **Value: 32**

Number of Epochs

- **Max Epochs: 20**
- **But training usually stops early** because of **EarlyStopping**.
- This ensures:
  - No unnecessary computation
  - Prevents overfitting
  - Automatically selects the best-performing model

Early Stopping

- **Monitor:** Validation Loss
- **Patience:** 4 epochs

When val_loss stops improving, the model might start memorizing noise—especially dangerous in medical datasets. Early stopping freezes the model at the optimal point and avoids wasted VM training time.

Image Size

- **48 × 48 grayscale**
- Reduces training time dramatically
- Still preserves essential bone structures

- Compatible with low-resource CPU environments
- Sufficient for binary fracture detection

Dropout Rate

- **Value: 0.4**
- **Why:** This noticeably improves generalization by disabling 40% of neurons during training. Medical images tend to be easy to overfit, so dropout provides needed regularization.

Loss Function

- **Type:** Sparse Categorical Cross-Entropy
- **Reason:**
    - Works directly with integer labels (0 = fracture, 1 = normal)
    - More memory-efficient than one-hot encoding
    - Standard for multi-class classification with probability outputs

Overall, the selected hyperparameters are optimized for:

- Fast training on CPU-only VMs
- Avoiding overfitting with a dataset
- Stable convergence without aggressive tuning
- Achieving high accuracy on fracture vs. normal classification

5. Data Splitting & Performance Evaluation

The dataset was split into 70% training, 15% validation, and 15% testing using Spark's randomSplit, which efficiently distributes files across the VM environment. This ratio provides a large enough training set for the CNN to learn meaningful patterns, a validation set to tune hyperparameters and detect overfitting, and a completely unseen test set for an unbiased final evaluation. Although the dataset is imbalanced (more fracture images than normal), Spark does not support stratified sampling for binaryFile, so a random split was the most practical choice. Because the fracture class is large, training remains stable, and any imbalance is later addressed through evaluation metrics like precision, recall, and F1-score.

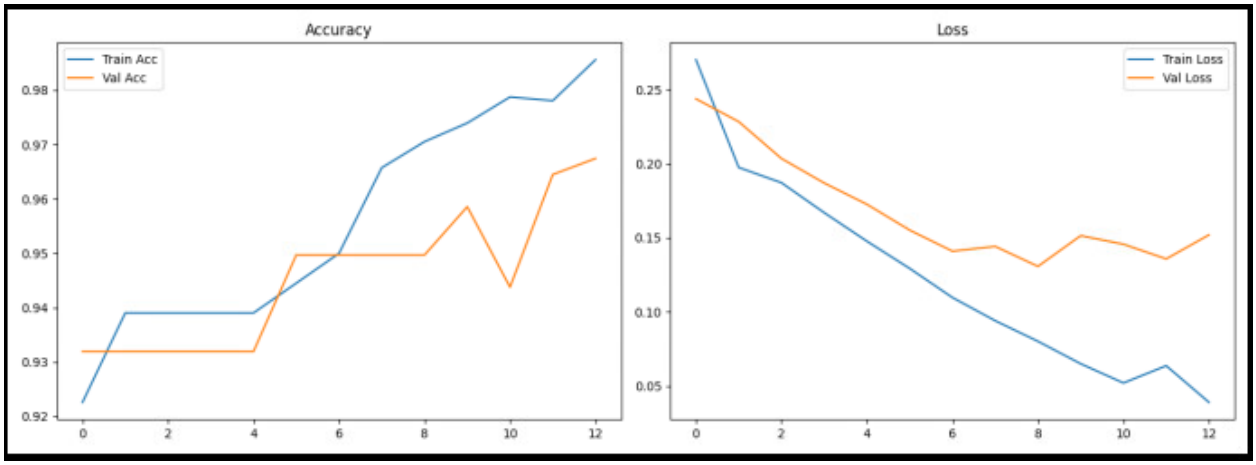
Performance Evaluation Metrics:

To assess the effectiveness of the CNN model, multiple evaluation metrics were used, including accuracy, precision, recall, F1-score, and confusion matrix analysis, along with training curves to monitor learning behavior and Grad-CAM visualizations for interpretability. These metrics collectively show how well the model detects fractures, how confidently it predicts each class,

and whether it generalizes without overfitting. The combination of quantitative results and visual explanations provides a comprehensive understanding of model performance.

Results:

1. Training Curves (Accuracy & Loss)



The accuracy curve shows a steady improvement for both training and validation, indicating stable learning. The loss curve decreases consistently, showing the model is converging without major overfitting.

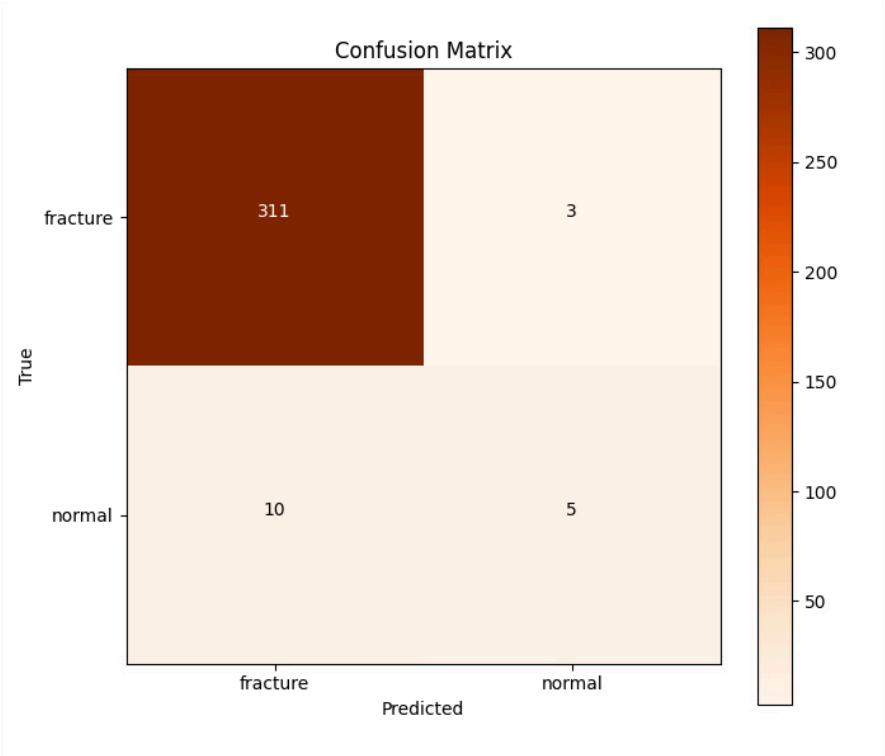
2. Classification Report

CNN.py | model_summary.txt | training_time.txt | classification_report.txt

	precision	recall	f1-score	support
fracture	0.97	0.99	0.98	314
normal	0.62	0.33	0.43	15
accuracy			0.96	329
macro avg	0.80	0.66	0.71	329
weighted avg	0.95	0.96	0.95	329

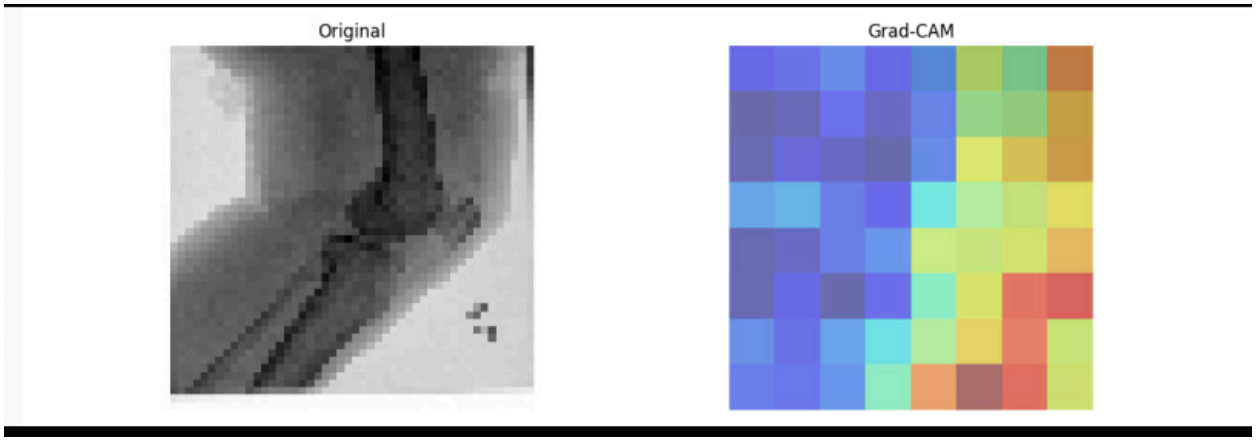
The report shows very strong performance on the fracture class (F1 = 0.98) but lower performance on the normal class due to class imbalance. Overall accuracy reaches 96%, showing the model is highly reliable for the primary task.

3. Confusion Matrix



The model correctly identifies fractures in most cases (311/314) but misclassifies several normal images as fractures due to their low representation. This visualization highlights class imbalance effects clearly.

4. Grad-CAM Visualization



Grad-CAM highlights the exact bone regions the model relies on when predicting fractures. This confirms that CNN is focusing on medically meaningful areas rather than irrelevant background noise.

6. Reflection & Thoughts

During this project, I gained a clearer understanding of how CNNs and Spark can work together in a real-world data pipeline. One challenge I faced was the class imbalance in the dataset, which made the normal class much harder to predict. This taught me the importance of evaluating the model with metrics beyond accuracy, such as precision, recall, and F1-score. Working on the VM setup also showed me how Spark helps with scalable data loading, even though the actual model training happens in TensorFlow. Overall, this project made me more confident in designing end-to-end image classification workflows under real resource constraints.

7. Conclusion & Future Directions

Although the model performed very well on fracture detection, I realized that improving the normal class requires either more balanced data or techniques like class weighting and data augmentation. In the future, I would like to explore larger CNN architectures, transfer learning, and potentially focal loss to handle imbalance more effectively. Adding more interpretability tools and deploying the model as an API service in a Spark-based pipeline would be useful steps forward. This project helped me appreciate both the strengths and limitations of lightweight CNNs and gave me a better perspective on how to improve medical imaging models in the next iteration.