Bone Break Classifier Dataset

(https://www.kaggle.com/datasets/amohankumar/bone-break-classifier-dataset/data)

About The Dataset

This dataset comprises a comprehensive collection of 1750 radiographic images, categorized into 12 distinct types of bone fractures. Each type is segregated into its specific directory under the main folder titled 'Bone_Break_Classifier'. The inclusion of various fracture types makes this dataset an invaluable asset for medical image analysis, particularly in the development of diagnostic tools and machine learning algorithms focused on bone injuries.

Fracture Types

The dataset encapsulates a wide range of fracture types, each with its unique characteristics:

- **Spiral Fracture**
  Images in this category show fractures caused by twisting forces resulting in helical breaks.
- **Pathological Fracture**
  This subset includes fractures occurring in bones weakened by diseases such as osteoporosis or cancer.
- **Oblique Fracture**
  These images depict fractures with an angled pattern across the bone.
- **Longitudinal Fracture**
  This category includes fractures running along the length of the bone.
- **Intra-Articular Fracture**

Images here show fractures that extend into the surfaces of joints.

- **Impacted Fracture**
  This type involves images of bone fragments that are driven into each other.
- **Hairline Fracture**
  These are images of small, thin breaks in the bone, often challenging to detect.
- **Greenstick Fracture**
  Common in children, these fractures involve bending and cracking of the bone without complete breakage.
- **Dislocation Fracture**
  This category combines joint dislocations with bone fractures.
- **Compression (Crush) Fracture**
  These images typically show fractures occurring in the spongy bone of the spine.
- **Comminuted Fracture**
  This subset includes complex fractures where the bone is broken into several pieces.
- **Avulsion Fracture**
  Here, the images show scenarios where a fragment of bone is torn away from the main mass.

Image Count per Class and Image Formats

```
Image Count per Class:
{'spiralFracture': 87,
 'PathologicalFracture': 110,
 'ObliqueFracture': 80,
 'LongitudinalFracture': 73,
 'IntraArticularFracture': 87,
 'ImpactedFracture': 89,
 'HairlineFracture': 125,
 'GreenstickFracture': 114,
 'DislocationFracture': 130,
 'CompressionCrushFracture': 120,
 'ComminutedFracture': 147,
 'AvulsionFracture': 122}
```

Fracture Type Distribution

- **Most Common Fracture:** The dataset contains the highest number of images for Comminuted Fractures (147 images), indicating a focus or greater interest in this fracture type, possibly due to its complexity.
- **Least Common Fracture:** Longitudinal Fractures are the least represented with 73 images. This could be due to their rarity or lower clinical significance in comparison to others.
- **Average Images per Fracture Type:** On average, each fracture type is represented by about 145-146 images.

```
Image Formats:
Counter({'jpg': 800, 'jpeg': 317, 'png': 140, 'gif': 26, 'webp': 1})
```

Image Format Distribution

- **Total Image Formats:** The images are distributed across five different formats.
- **Most Common Format:** JPEG (including both 'jpg' and 'jpeg' extensions) is the dominant format, with 1117 images (800 'jpg' + 317 'jpeg'), accounting for approximately 64% of the dataset. This popularity could be due to JPEG's wide usage and balance of quality and file size.
- **PNG Format:** With 140 images, PNG is the second most common format. PNG is known for its lossless compression, suggesting that these images might be of higher quality or used where transparency is needed.
- **Other Formats:** GIF and WebP formats are less represented (26 and 1 images, respectively), indicating they are less preferred, possibly due to their specific use-cases or limitations in quality for radiographic images.

Code

(https://drive.google.com/file/d/1evIbxdxBqsxdynECB4PPm-srPR9QGd1a/view?usp=sharing)

Remove Unnecessary Image Extensions————————————————————————————

As mentioned before, in my dataset, there are images with extensions of *.gif* and *.webp* .
I want to remove images with that extension since we can not process images with those extensions.
By doing this code below, we can easily remove images with unnecessary extensions.

```
# Remove files with extension .gif and .webp
def remove_files_with_extension(path, extensions):
    for filename in os.listdir(path):
        if filename.lower().endswith(extensions):
            os.remove(os.path.join(path, filename))
            print(f"Removed: {filename}")
extensions_to_remove = ('.gif', '.webp')
```

```
# Iterating over each class directory and remove files with unnecessary
extensions
for class_name, path in dataset_paths.items():
    print(f"Cleaning up class: {class_name}")
    remove_files_with_extension(path, extensions_to_remove)


>
Cleaning up class: spiralFracture
Cleaning up class: PathologicalFracture
Removed: figure1.gif
Removed: figure2.gif
Cleaning up class: ObliqueFracture
Removed: Foot-Series.gif
Cleaning up class: LongitudinalFracture
Removed: C4-FF2-6.gif
Removed: fracture-complete.gif
Removed: radiol.16142305.fig3.gif
Removed: rg.2020190085.fig9.gif
Cleaning up class: IntraArticularFracture
Removed: 12245_2015_75_Fig5_HTML.gif
Removed: rg.2016150216.fig27b.gif
Removed: Xray20-20Lat20-20Smith20Fx_moved.gif
Cleaning up class: ImpactedFracture
Cleaning up class: HairlineFracture
Removed: 13244_2014_371_Fig1_HTML.gif
Removed: rg.2018180073.fig10a.gif
Removed: rg.2018180073.fig4c.gif
Removed: rg.2018180073.fig4d.gif
Removed: rg.2018180073.fig6a.gif
Removed: rg.2018180073.fig9.gif
Cleaning up class: GreenstickFracture
Removed: A214966_1_En_4_Fig3b_HTML.gif
Removed: buckle-fx-distal-radius.gif
Removed: greenstick__distal__fibula..gif
Removed: rg.342135073.fig4.gif
Cleaning up class: DislocationFracture
Removed: CCASE_44-FF1.gif
Removed: CCASE_44-FF4.gif
Cleaning up class: CompressionCrushFracture
Removed: radiol.2017162100.fig5b.gif
Removed: traum1c.gif
Cleaning up class: ComminutedFracture
Removed: 1773-97598.webp
Cleaning up class: AvulsionFracture
Removed: GII-2-137-g001.gif
Removed: ijfa-1-001-001.gif
```

Extract Feature Embeddings

Now the dataset is cleaned from images with unnecessary extensions. Now I want to process the dataset of images and extract feature embeddings for each image using a pre-trained MobileNetV2 model.

```
# Pre-trained MobileNetV2 model
model = models.mobilenet_v2(pretrained='imagenet')
model.eval()

data_list = []

for class_name, path in tqdm(dataset_paths.items()):
    images = os.listdir(path)

    for img_name in images:
        img_path = os.path.join(path, img_name)

        # Load and preprocess image
        image = Image.open(img_path)
        if image.mode != 'RGB':
            image = image.convert('RGB')
        preprocess = transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225]) # Mean and Std_dev of RGB
        ])
        image = preprocess(image)
        image = image.unsqueeze(0)

        # Extract embeddings
        with torch.no_grad():
            embeddings = model(image)

            # Create a dictionary with embeddings, class, and path
            row_data = {f"embedding_{i+1}": value.item() for i, value in
enumerate(embeddings[0])}
            row_data['class'] = class_name
            row_data['path'] = img_path
            data_list.append(row_data)

# Create a DataFrame from the list of dictionaries
workingDF = pd.DataFrame(data_list)
```

The Embedded data frame will look like this:

	embedding_1	embedding_2	embedding_3	embedding_4	embedding_5	...	embedding_995	embedding_996	embedding_997	embedding_998	embedding_999	embedding_1000	class	path
0	-3.378522	1.284846	0.043607	1.260554	3.479605	...	-4.443556	0.620382	-1.740096	-3.785853	0.998965	2.736010	spiralFracture	Bone_Break_Classifier/SpiralFracture/0_jumbo...
1	-1.848094	2.978379	1.532953	2.569057	5.004469	...	-1.481201	1.368065	0.150323	-3.330071	3.124024	4.390635	spiralFracture	Bone_Break_Classifier/SpiralFracture/110001_ju...
2	-3.809473	-3.904888	-1.765737	-3.538975	0.886887	...	-0.489107	-2.222746	-0.950555	-3.858117	0.652780	1.931750	spiralFracture	Bone_Break_Classifier/SpiralFracture/1286_spir...
3	-3.789466	-3.912750	-3.969476	-3.319884	1.116871	...	0.191601	-2.142022	-0.540787	-7.445231	0.476014	3.149789	spiralFracture	Bone_Break_Classifier/SpiralFracture/1286_spir...
4	-1.938522	-0.330303	3.573163	3.438416	5.139499	...	-1.660001	-4.241234	-4.163421	-5.089740	1.737933	7.669780	spiralFracture	Bone_Break_Classifier/SpiralFracture/192811243...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1252	-0.149328	1.419265	0.645460	0.597279	1.304086	...	-2.498964	-0.481475	0.288701	-3.391221	-1.704050	2.342594	AvulsionFracture	Bone_Break_Classifier/AvulsionFracture/slide_5...
1253	1.276412	1.960133	-0.877962	-1.277206	-2.036461	...	-1.305279	-1.626999	-0.299629	-3.979919	1.550607	2.024171	AvulsionFracture	Bone_Break_Classifier/AvulsionFracture/spine.jpg
1254	-0.899119	1.640448	-0.671237	-0.113375	-0.290816	...	-2.991724	-1.507977	-1.074937	-1.981432	1.929319	3.820926	AvulsionFracture	Bone_Break_Classifier/AvulsionFracture/tcr-5-0...
1255	-3.132690	2.122205	-0.890480	1.972687	4.160993	...	-6.398017	-3.273866	-1.869247	-4.927500	1.090722	2.114085	AvulsionFracture	Bone_Break_Classifier/AvulsionFracture/tibial...
1256	0.607556	3.220061	-0.110223	2.680245	2.715802	...	0.842477	0.130532	1.110834	-4.682260	1.055729	2.867413	AvulsionFracture	Bone_Break_Classifier/AvulsionFracture/Xray-pi...

Image Processing Loop

The outer for loop iterates over each class of images in the dataset, with class_name and path representing the class and the directory of images, respectively.

- os.listdir(path): Lists all the image files in the current class directory.

The inner for loop processes each image individually:

- Image.open(img_path): Opens the image file.
- image.convert('RGB'): Converts images to RGB format if they are not already, ensuring consistency.
- preprocess: A series of transformations (resizing to 224x224 pixels, converting to a tensor, and normalizing using the specified mean and standard deviation) are applied to prepare the image for processing by MobileNetV2.

Feature Extraction

For each image, the code extracts feature embeddings using the MobileNetV2 model:

- with torch.no_grad(): Disables gradient calculations, which are unnecessary during inference.
- embeddings = model(image): Passes the preprocessed image through the model to obtain embeddings.
- A dictionary row_data is created for each image, containing the embedding values, the class of the fracture, and the image path.

DataFrame Creation

- The extracted information for each image is added to data_list.
- pd.DataFrame(data_list): Finally, a Pandas DataFrame is created from this list, where each row represents an image with its corresponding embeddings, class, and file path.

Resulting DataFrame

The DataFrame contains columns for each embedding dimension (embedding_1, embedding_2, ..., embedding_1000), the class of the image (class), and the file path (path). Each row corresponds to an image in the dataset.

Insights

- The extracted embeddings represent high-level features of the images, learned by the MobileNetV2 model on the ImageNet dataset, and are useful for various downstream tasks like similarity search which I will implement later on this code.

Identify and Removing Duplicate Images

After extracting feature embedding of each image in the dataset, we can easily identify duplicated images by calculating a cosine similarity matrix of the embedded value of each image.

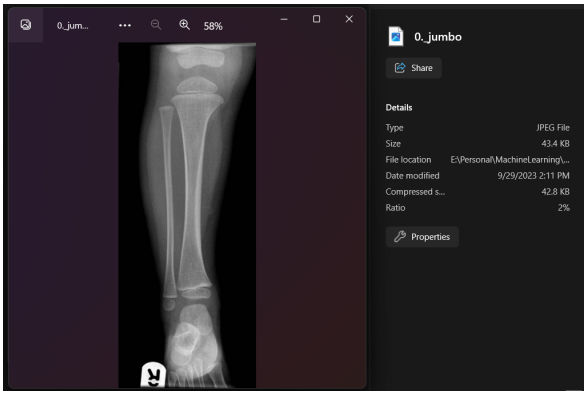
```
# Calculate cosine similarity matrix
```

```
similarity_matrix = cosine_similarity(workingDF.iloc[:, :-2])

# List to store duplicate pairs
duplicates = []

# Set cosine similarity threshold
cosine_threshold = 0.85

# Find duplicate pairs
for i in range(len(workingDF)):
    for j in range(i + 1, len(workingDF)):
        if 0.95 > similarity_matrix[i, j] > cosine_threshold:
            duplicates.append((i, j))

# Display duplicate pairs
for i, j in duplicates:
    if i != j:
        print(workingDF.loc[i, 'path'].split(os.path.sep)[-2:], ' similar to ',
workingDF.loc[j, 'path'].split(os.path.sep)[-2:])
```

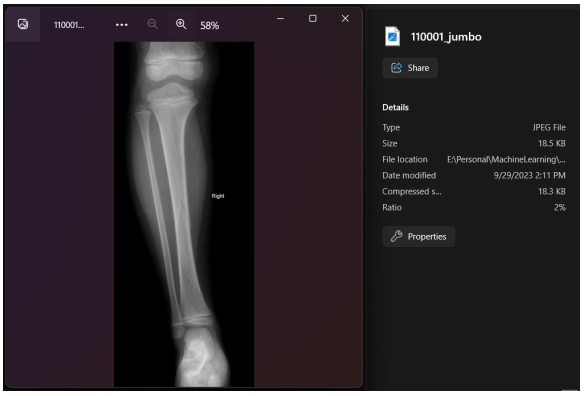
By doing this code, we will get the result of duplicated images, here's several rows of the result:

```
['Bone_Break_Classifier/SpiralFracture/0._jumbo.jpeg']  similar to
['Bone_Break_Classifier/SpiralFracture/110001_jumbo.jpeg']

['Bone_Break_Classifier/SpiralFracture/0._jumbo.jpeg']  similar to
['Bone_Break_Classifier/LongitudinalFracture/110002_gallery.jpeg']

['Bone_Break_Classifier/SpiralFracture/0._jumbo.jpeg']  similar to
['Bone_Break_Classifier/GreenstickFracture/3115574c112e21d961cf0f63929a8786.jpg'
]

['Bone_Break_Classifier/SpiralFracture/0._jumbo.jpeg']  similar to
['Bone_Break_Classifier/GreenstickFracture/Greenstickfx.jpg']
```

To make sure this task is really capture the duplicate image, here's the sample image of the result given above:



Bone_Break_Classifier/SpiralFracture/0._jumbo.jpeg



Bone_Break_Classifier/SpiralFracture/110001_jumbo.jpeg

Both images look similar but the image path is different. To handle this, I just have to remove the right-side image [j] and keep the left-side image [i]. We can do this by running this code:

```
# Iterate over duplicate pairs
for i, j in duplicates:
    try:
        path_i = workingDF.loc[i, 'path']
        path_j = workingDF.loc[j, 'path']

        # Check if the files exist before processing
        if os.path.exists(path_i) and os.path.exists(path_j):
            embedding_i = workingDF.loc[i, workingDF.columns[:-2]].values
            class_i = workingDF.loc[i, 'class']
            class_j = workingDF.loc[j, 'class']

            # Check if classes are the same
            if class_i == class_j:
                size_i = os.path.getsize(path_i)
                size_j = os.path.getsize(path_j)
                if size_j < size_i:
                    removed_paths.append(path_j)
                    os.remove(path_j)
                else:
                    removed_paths.append(path_i)
                    os.remove(path_i)
            else:
                # Check if mean embeddings exist for both classes
                if class_i in class_mean_embeddings and class_j in class_mean_embeddings:
                    distance_ii = cosine(embedding_i, class_mean_embeddings[class_i])
                distance_ij = cosine(embedding_i, class_mean_embeddings[class_j])

              # Compare distances and remove the image with the larger distance
                    if distance_ii < distance_ij:
                        # Print file paths before attempting to remove
                        if os.path.exists(path_j):
                            removed_paths.append(path_j)
                            os.remove(path_j)
                    else:
                        removed_paths.append(path_i)
                        os.remove(path_i)
    except Exception as e:
        print(f"An error occurred: {e}")

# List to store duplicate pairs
duplicate_confirm = []

# Set cosine similarity threshold
cosine_threshold = 0.85

# Find duplicate pairs
for i in range(len(workingDF)):
    for j in range(i + 1, len(workingDF)):
        if 0.95 > similarity_matrix[i, j] > cosine_threshold:
            duplicate_confirm.append((i, j))
```

```
# Check if there are no duplicates
if not duplicate_confirm:
    print("Duplication handled.")
else:
    print("Duplication handled. No specific pairs shown.")
```

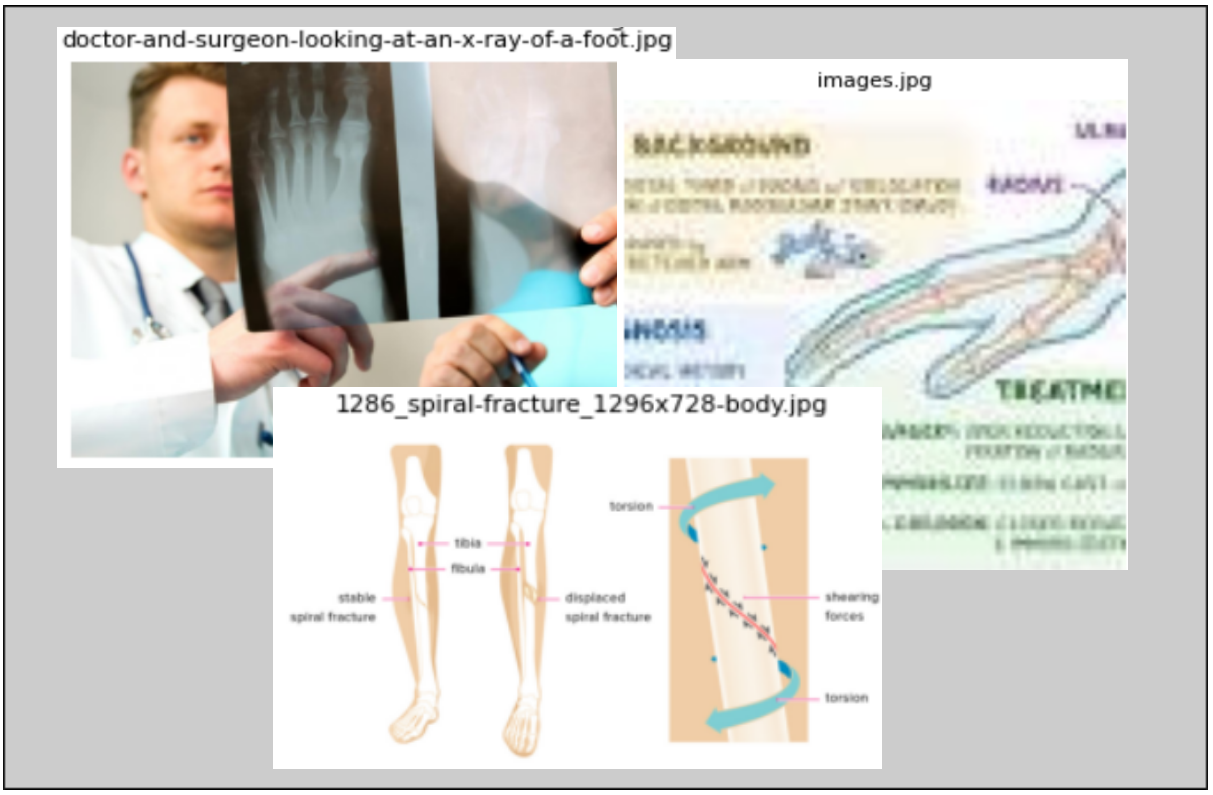
Identify and Removing Outlier Images

Now I want to see if there are any outliers on this dataset, the outlier of images could be images that contain anything that is not related to bone fracture, it could be there's a human inside the image, there's text inside the image, and so on. Since the 'workingDF' data frame contain embedded value for each image, we also can easily find the outliers by running this code:

```
# PCA for dimensionality reduction
pca = PCA(n_components=50)
reduced_embeddings = pca.fit_transform(workingDF.iloc[:, :-2])

# Using Isolation Forest for outlier detection
from sklearn.ensemble import IsolationForest

# Using Isolation Forest for outlier detection
isolation_forest = IsolationForest(contamination=0.05)
outliers = isolation_forest.fit_predict(reduced_embeddings)

# Add outlier information to the DataFrame
workingDF['outlier'] = outliers

# Filter out the outlier images
outlier_images = workingDF[workingDF['outlier'] == -1]['path'].tolist()
def display_images_in_grid_with_paths(image_paths, columns=5):
    if not image_paths:
        print("No images to display.")
        return

    rows = len(image_paths) // columns
    rows += 0 if len(image_paths) % columns == 0 else 1

    fig, axs = plt.subplots(rows, columns, figsize=(15, rows * 3))
    axs = axs.ravel()

    for idx, img_path in enumerate(image_paths):
        img = Image.open(img_path)
        axs[idx].imshow(img)
        axs[idx].set_axis_off()
        axs[idx].set_title(os.path.basename(img_path), fontsize=8)  # Display
image path

    for ax in axs[len(image_paths):]:
        ax.set_axis_off()
```

```
    plt.tight_layout()
    plt.show()

# Display the outlier images
display_images_in_grid_with_paths(outlier_images)
```

The output will give us list of images that identified as outliers, here's the sample result:



These images are well-identified as outliers because they contains humans, text, and anything that is not related to an X Ray bone fracture. All i have to do now is remove the outliers.

```
# Flag the outliers in the DataFrame
workingDF['is_outlier'] = workingDF['outlier'] == -1

# Create a new DataFrame without the outliers
workingDF_clean = workingDF[~workingDF['is_outlier']].drop(columns=['outlier',
'is_outlier'])

# reset the index of the new DataFrame
workingDF_clean.reset_index(drop=True, inplace=True)
workingDF_clean
```

By running this code, the outlier images is removed from the dataset. But notice here that i assign a new dataframe named 'workingDF_clean' to store the cleaned data.

Identify and Removing Irrelevant images

The image is now cleaned, but I wonder, is there any irrelevant image in each class? If so, I want to remove them so that the classification task will giving me the best accuracy.
Searching irrelevant images is quite easy since the data frame is already embedded into list of numerical value, by calculating the distance between mean of each class and mean of image.

```
unique_classes = workingDF_clean['class'].unique()
# Iterate over unique classes
for class_ in unique_classes:
    class_df = workingDF_clean[workingDF_clean['class'] == class_]

    # Calculate mean embedding for the class
    mean_embedding = np.mean(class_df.iloc[:, :-3].values, axis=0)
    class_mean_embeddings[class_] = mean_embedding

# Set a threshold for considering an image as irrelevant
distance_threshold = 0.5  # I run this code several time to use threshold
distance of 0.2, 0.3, 0.4, and 0.5

# List to store potentially irrelevant images
potential_irrelevant_images = []

# Iterate over all images
for i in range(len(workingDF_clean)):
    try:
        embedding_i = workingDF_clean.loc[i,
workingDF_clean.columns[:-3]].values
        class_i = workingDF_clean.loc[i, 'class']

        # Calculate the distance between the image's mean embedding and the
class mean embedding
        distance_to_class_mean = cosine(embedding_i,
class_mean_embeddings[class_i])

        # Compare with the threshold
        if distance_to_class_mean > distance_threshold:
            # Store the potentially irrelevant image
            potential_irrelevant_images.append(workingDF_clean.loc[i, 'path'])
    except Exception as e:
         print(f"An error occurred at index {i}: {type(e).__name__}, {str(e)}")

# Write potentially irrelevant images to a CSV file
csv_file_path = 'potential_irrelevant.csv'
with open(csv_file_path, 'a', newline='') as csvfile:
    csv_writer = csv.writer(csvfile)
    if csvfile.tell() == 0:
        csv_writer.writerow(['Path'])

    # Write potentially irrelevant images
    csv_writer.writerows([[image_path] for image_path in
```

```
potential_irrelevant_images])

print(f"Potentially irrelevant images appended to {csv_file_path}")
```

Identify Unique Classes

- unique_classes = workingDF_clean['class'].unique(): Extracts the unique classes from the 'class' column of the workingDF_clean DataFrame

  .

Calculate Mean Embeddings for Each Class

- This code section will iterate over each unique class, creating a subset DataFrame class_df for each class.
- It calculates the mean embedding for each class by averaging the values of all embeddings in that class

Setting a Threshold

- distance_threshold: Defines a threshold for determining whether an image is considered potentially irrelevant. The threshold represents a limit on the acceptable distance between an image's embedding and the class mean embedding.
- Notice that i Run the code with different thresholds (0.2, 0.3, 0.4, 0.5) allow me to get all possibility of irrelevant images.

Identify Potentially Irrelevant Images

- The code iterates over each image in the workingDF_clean DataFrame.
- For each image, it calculates the cosine distance between its embedding and the mean embedding of its class.
- If this distance is greater than the specified threshold (distance_threshold), the image is flagged as potentially irrelevant.

Recording Potentially Irrelevant Images

- The paths of the potentially irrelevant images are stored in a list potential_irrelevant_images.
- These paths are then written to a CSV file named 'potential_irrelevant.csv'. The code appends to the file if it exists.
- The CSV file starts with a header row 'Path' and then lists the paths of the potentially irrelevant images.

The main reason why I create a new CSV file to store all possibility of irrelevant images is, will explained below.

The reason is, I will find the mode (most frequently shown data) on the ‘potential_irrelevant.csv’, by doing this, I will get real irrelevant images.

```
# Load the CSV file into a DataFrame
irrelevants = pd.read_csv(csv_file_path)

# Find the mode of the 'path' column
```

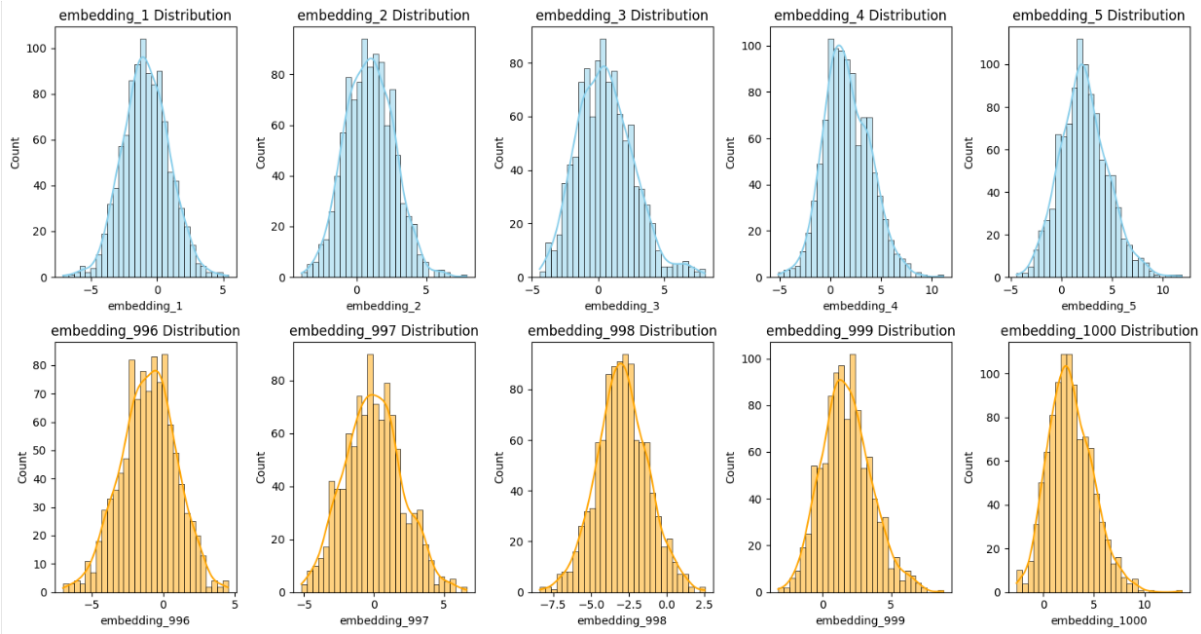
```
mode_paths = irrelevants['Path'].mode()

# Print the frequently occurring paths
print("Frequently occurring paths (mode):")
for path in mode_paths:
    print(path)
>
Frequently occurring paths (mode):
Bone_Break_Classifier/CompressionCrushFracture/images24.jpg
Bone_Break_Classifier/CompressionCrushFracture/images29.jpg
Bone_Break_Classifier/DislocationFracture/a-Radiograph-showing-the-sacroiliac-fr
acture-dislocation-with-fractures-of-bilateral.png
Bone_Break_Classifier/DislocationFracture/image4.jpeg
Bone_Break_Classifier/ImpactedFracture/images10.jpg
Bone_Break_Classifier/LongitudinalFracture/19261tn.jpg
Bone_Break_Classifier/ObliqueFracture/oblique-view-of-adult-fractured-elbow-ARR2
YE.jpg
Bone_Break_Classifier/PathologicalFracture/image4.jpeg
Bone_Break_Classifier/PathologicalFracture/medical-case-reports-cortical-destruc
tion-5-1-108-g003.png
```

Now all the the irrelevant images already removed, i will save the 'workingDF_clean' data frame into a CSV file so i can proceed to image classification task.

```
# Remove rows with paths found in mode_paths from the original DataFrame
for path in mode_paths:
    index_to_remove = workingDF_clean[workingDF_clean['path'] == path].index
    workingDF_clean = workingDF_clean.drop(index_to_remove)

# Verify the DataFrame after removal
workingDF_clean.head()

# save the cleaned DataFrame to a new CSV file
workingDF_path = 'BONEBREAK_cleaned_data.csv'
workingDF_clean.to_csv(workingDF_path, index=False)
```

Histogram Distribution of each feature embeddings————————————————————————

Before jumping into the classification model, I want to show the histogram distribution of values for different feature embeddings extracted from the cleaned data.



Observations

- **Normal Distributions**
  Embeddings 1 through 5 (blue histograms) show distributions that resemble a normal distribution, with a single prominent peak and values tapering off symmetrically on both sides. This suggests that for these embeddings, most of the data points are clustered around the mean.

- **Scale of Values**
  The x-axes of the histograms indicate the range of values that the embeddings take. For instance, embeddings 1 through 5 range from approximately -5 to +10, while embeddings 996 through 1000 range from about -7.5 to +10. This shows that the features have been well-scaled similarly, which is common practice before feeding data into machine learning models.

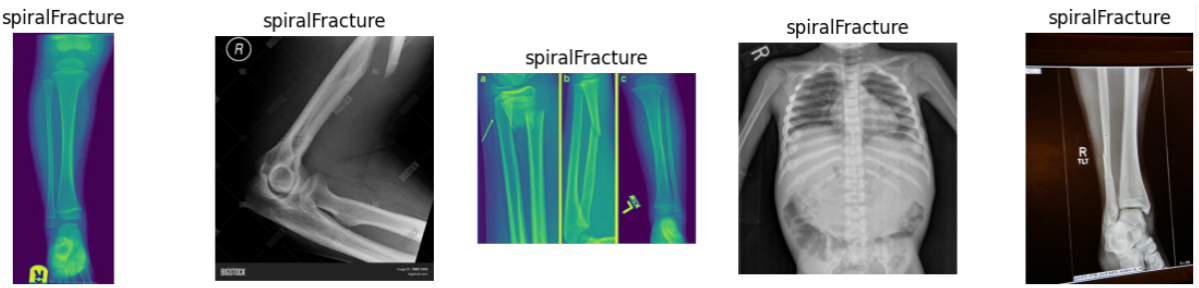
- **Outliers**
  The histograms do not indicate the presence of significant outliers, as the tails of the distributions taper off smoothly, which is preferable for many statistical models.

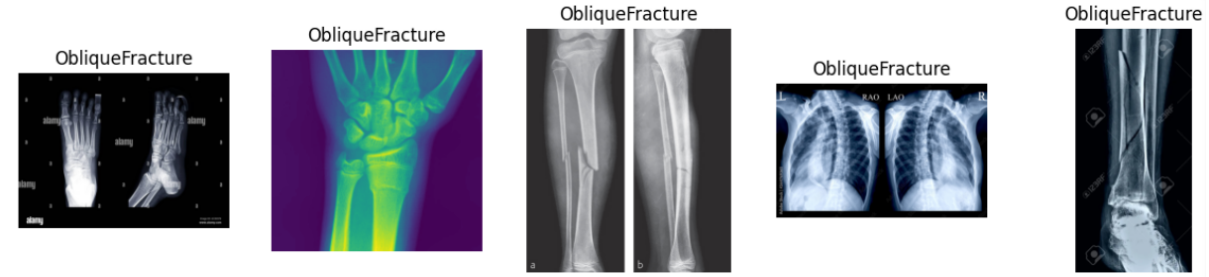
Before proceeding to the classification task, firstly I want to create a new directory folder that contain images that already pre-processed.

```
# Root dir for the cleaned images
base_dir = 'BoneBreak_Clean'

# Clear out the existing directory to start fresh
if os.path.exists(base_dir):
    shutil.rmtree(base_dir)
os.makedirs(base_dir)
```

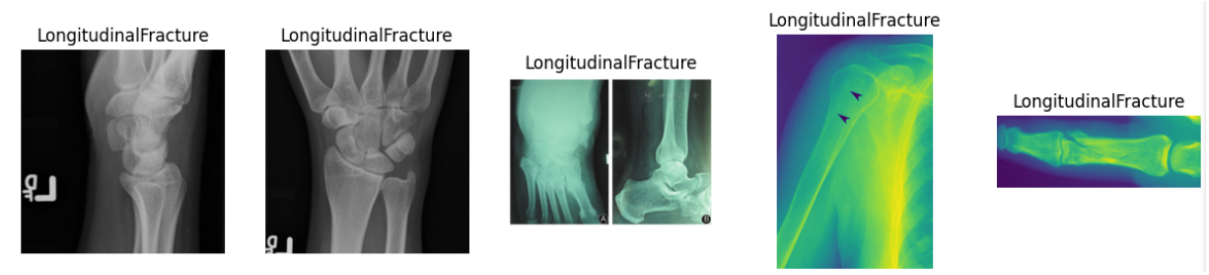
```
# Copying images from workingDF_clean
for class_name in workingDF_clean['class'].unique():
    class_dir = os.path.join(base_dir, class_name)
    os.makedirs(class_dir, exist_ok=True)

    for _, row in workingDF_clean[workingDF_clean['class'] ==
class_name].iterrows():
        src_path = row['path']
        dst_path = os.path.join(class_dir, os.path.basename(src_path))
        shutil.copy(src_path, dst_path)

print("Images have been copied to 'BoneBreak_Clean/'.")

# Paths to the cleaned dataset folders
cleaned_dataset_paths = {
    "spiralFracture": "BoneBreak_Clean/SpiralFracture/",
    "PathologicalFracture": "BoneBreak_Clean/PathologicalFracture/",
    "ObliqueFracture": "BoneBreak_Clean/ObliqueFracture/",
    "LongitudinalFracture": "BoneBreak_Clean/LongitudinalFracture/",
    "IntraArticularFracture": "BoneBreak_Clean/IntraArticularFracture/",
    "ImpactedFracture": "BoneBreak_Clean/ImpactedFracture/",
    "HairlineFracture": "BoneBreak_Clean/HairlineFracture/",
    "GreenstickFracture": "BoneBreak_Clean/GreenstickFracture/",
    "DislocationFracture": "BoneBreak_Clean/DislocationFracture/",
    "CompressionCrushFracture": "BoneBreak_Clean/CompressionCrushFracture/",
    "ComminutedFracture": "BoneBreak_Clean/ComminutedFracture/",
    "AvulsionFracture": "BoneBreak_Clean/AvulsionFracture/"
}
```

Now I want to show you a few images for each class!

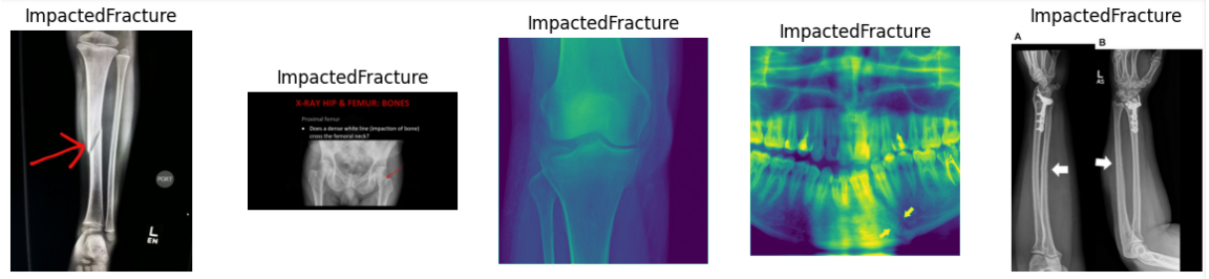



ObliqueFracture
ObliqueFracture
ObliqueFracture
ObliqueFracture
ObliqueFracture

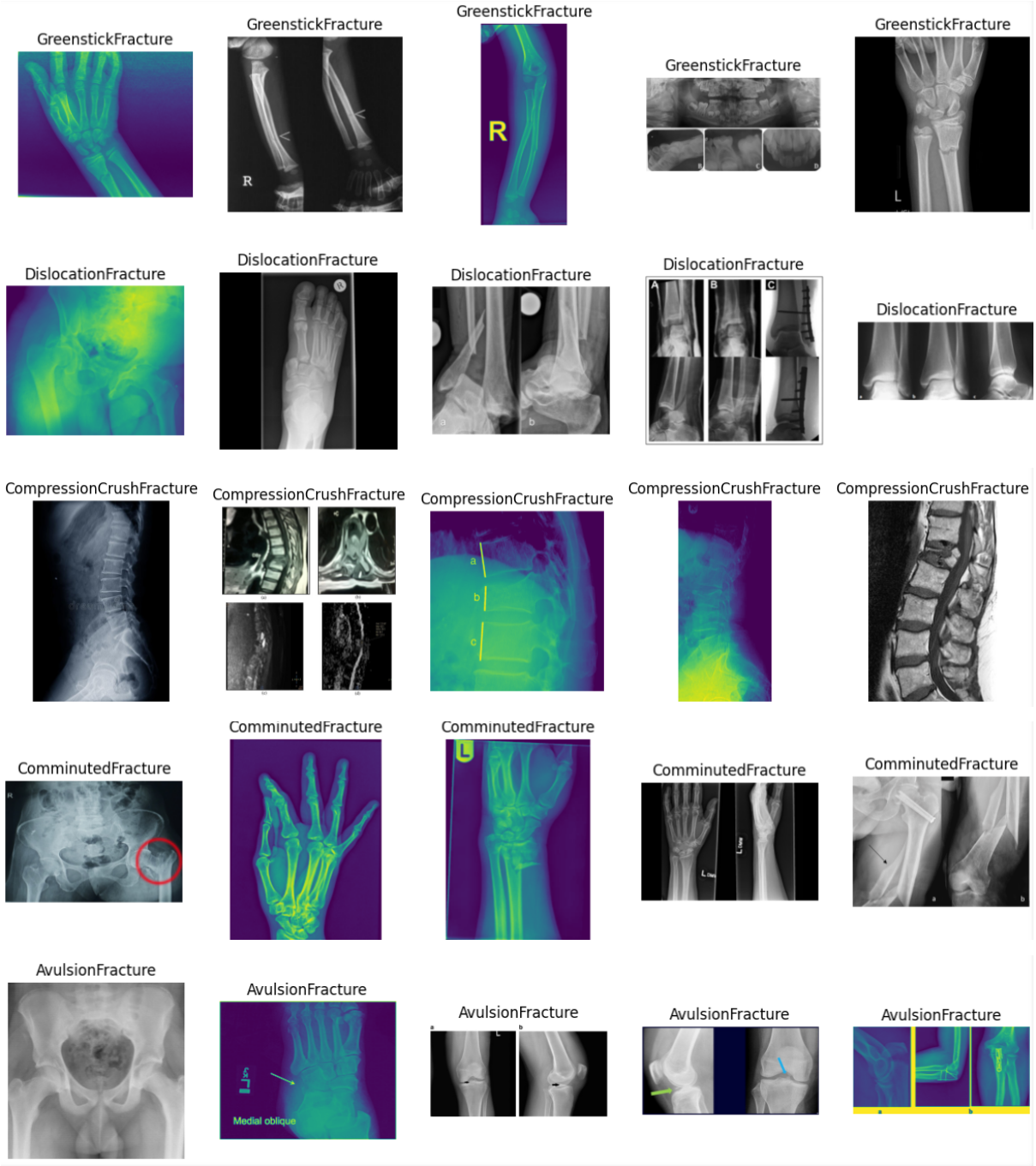

PathologicalFracture
PathologicalFracture
PathologicalFracture
PathologicalFracture
PathologicalFracture


LongitudinalFracture
LongitudinalFracture
LongitudinalFracture
LongitudinalFracture
LongitudinalFracture


IntraArticularFracture
IntraArticularFracture
IntraArticularFracture
IntraArticularFracture
IntraArticularFracture


ImpactedFracture
ImpactedFracture
ImpactedFracture
ImpactedFracture
ImpactedFracture


HairlineFracture
HairlineFracture
HairlineFracture
HairlineFracture
HairlineFracture



Now we are ready to do the classification task!

Classification Model

For the classification task, I will use a pre-trained ResNet-18 model. The reason why I'm using a pre-trained ResNet-18 for image classification is because it allows us to harness the power of transfer learning, such as the ability to detect edges, textures, and patterns. This approach often leads to better performance. Moreover, the architecture of ResNet-18, with its residual connections, enables the training of deep neural networks without the vanishing gradient problem;this will improve the model's ability to learn complex features.

Firstly, I will perform several tasks related to image data preprocessing, augmentation, dataset creation, and splitting for the purpose of training a machine learning model. Here's the code:

```
# Define the transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    transforms.RandomRotation(degrees=30),  # Random rotation
    transforms.RandomAffine(degrees=0, translate=(0.1, 0.1)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
# Load the ImageFolder dataset
dataset = datasets.ImageFolder(root="BoneBreak_Clean/", transform=transform)

# Split the dataset into training and test sets
X = [x[0] for x in dataset.imgs]  # List of file paths
y = [x[1] for x in dataset.imgs]  # List of labels

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y,
random_state=42)

# Create new ImageFolder datasets for training and test sets
train_dataset = datasets.ImageFolder(root="BoneBreak_Clean/", transform=transform)
train_dataset.imgs = [(X_train[i], y_train[i]) for i in range(len(X_train))]

test_dataset = datasets.ImageFolder(root="BoneBreak_Clean/", transform=transform)
test_dataset.imgs = [(X_test[i], y_test[i]) for i in range(len(X_test))]
```

Define Transformations

A series of image transformations is defined using transforms.Compose, which sequentially applies the following:

- transforms.Resize((224, 224)): Resizes the images to 224x224 pixels.
- transforms.RandomResizedCrop(224): Performs random cropping on the image and then resizes it to 224x224 pixels.
- transforms.RandomHorizontalFlip(): Randomly flips the image horizontally with a default 50% chance.
- transforms.ColorJitter(...): Randomly changes the brightness, contrast, saturation, and hue of the image to introduce more variability.
- transforms.RandomRotation(degrees=30): Randomly rotates the image by up to ±30 degrees.
- transforms.RandomAffine(...): Applies a random affine transformation to the image, in this case with translation of up to 10% in the x and y directions.
- transforms.ToTensor(): Converts the image to a PyTorch tensor.
- transforms.Normalize(...): Normalizes the image tensor using the specified mean and standard deviation for each color channel (assuming the use of the ImageNet dataset mean and std. deviation).

Load ImageFolder Dataset

The datasets.ImageFolder class is used to load a dataset of images stored in a directory where each subdirectory corresponds to a class. The transform defined above is applied to each image in the dataset.

Split Dataset

- The dataset is split into file paths (X) and labels (y).
- These lists are then split into training and test sets using train_test_split, with 20% of the data reserved for testing. The split is stratified to maintain the original distribution of classes, and a fixed random state ensures reproducibility.

Create Training and Test Sets

New instances of datasets.ImageFolder are created for both the training and test sets. The imgs attribute of these datasets is replaced with the respective subsets obtained from the split.

The next step is setting up and executing a training loop for a convolutional neural network model using PyTorch, specifically targeting image classification with a dataset of images representing different bone fracture types. Here's the code:

```
# Create a DataLoader with batch size 10 to get 10 random samples
dataloader = DataLoader(train_dataset, batch_size=10, shuffle=True)

# Load a pre-trained ResNet model
pretrained_resnet = models.resnet18(pretrained=True)

# Modify the last layer for number of classes
num_classes = 12
pretrained_resnet.fc = nn.Linear(pretrained_resnet.fc.in_features, num_classes)

# Transfer the model to my device
pretrained_resnet.to(device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(pretrained_resnet.parameters(), lr=0.0001)


# Create DataLoaders
train_dataloader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# Training loop
num_epochs = 25
for epoch in range(num_epochs):
    # Training
    pretrained_resnet.train()
    train_loss = 0.0
    for inputs, labels in train_dataloader:
        inputs, labels = inputs.to(device), labels.to(device)
```

```
            # Zero the gradients
            optimizer.zero_grad()

            # Forward pass
            outputs = pretrained_resnet(inputs)
            loss = criterion(outputs, labels)

            # Backward pass and optimization
            loss.backward()
            optimizer.step()

            train_loss += loss.item() * inputs.size(0)

        # Calculate average training loss
        train_loss = train_loss / len(train_dataloader.dataset)

        # Validation
        pretrained_resnet.eval()
        val_loss = 0.0
        correct_predictions = 0
        total_predictions = 0

        with torch.no_grad():
            for inputs, labels in test_dataloader:
                inputs, labels = inputs.to(device), labels.to(device)

                # Forward pass
                outputs = pretrained_resnet(inputs)
                loss = criterion(outputs, labels)

                val_loss += loss.item() * inputs.size(0)

                _, predicted = torch.max(outputs, 1)
                correct_predictions += torch.sum(predicted == labels).item()
                total_predictions += labels.size(0)

        # Calculate average validation loss and accuracy
        val_loss = val_loss / len(test_dataloader.dataset)
        accuracy = correct_predictions / total_predictions
```

DataLoader Initialization

A DataLoader for the training dataset is created with a batch size of 32. This object shuffles the data and provides batches of data to the training loop.

Model Preparation

- A pre-trained ResNet-18 model is loaded. ResNet-18 is a CNN that is commonly used for image classification tasks.
- The final fully connected layer (fc) of the pre-trained ResNet model is modified to output num_classes (12 in this case) to match the number of fracture classes in the dataset.

Model Transfer

The modified model is transferred to the my device (CPU)

Loss Function and Optimizer

The loss function is set to CrossEntropyLoss, which is standard for classification tasks.
The optimizer is set to Adam, with a learning rate of 0.0001, which will be used to update the model's weights during training.

Training Loop

- The training loop is set to run for 25 epochs.
- Inside the loop, the model is set to training mode with pretrained_resnet.train().
- For each batch of data from the train_dataloader, the model performs a forward pass, calculates the loss, and then performs a backward pass to update the weights.
- The training loss is accumulated to calculate the average loss at the end of the epoch.

Validation

- After each training epoch, the model is set to evaluation mode with pretrained_resnet.eval().
- It then processes the test dataset without updating the model's weights, calculating the validation loss and the number of correct predictions to determine the accuracy.

Model Accuracy

```
Epoch [1/25], Training Loss: 2.4877, Validation Loss: 2.1903, Accuracy: 0.25
Epoch [2/25], Training Loss: 2.1660, Validation Loss: 1.9995, Accuracy: 0.33
Epoch [3/25], Training Loss: 2.0227, Validation Loss: 1.8751, Accuracy: 0.38
Epoch [4/25], Training Loss: 1.9066, Validation Loss: 1.7996, Accuracy: 0.40
Epoch [5/25], Training Loss: 1.8630, Validation Loss: 1.7369, Accuracy: 0.42
Epoch [6/25], Training Loss: 1.7774, Validation Loss: 1.6231, Accuracy: 0.45
Epoch [7/25], Training Loss: 1.7049, Validation Loss: 1.5672, Accuracy: 0.48
Epoch [8/25], Training Loss: 1.6706, Validation Loss: 1.4890, Accuracy: 0.49
Epoch [9/25], Training Loss: 1.5585, Validation Loss: 1.4804, Accuracy: 0.51
Epoch [10/25], Training Loss: 1.4889, Validation Loss: 1.3536, Accuracy: 0.57
Epoch [11/25], Training Loss: 1.4515, Validation Loss: 1.2877, Accuracy: 0.59
Epoch [12/25], Training Loss: 1.4287, Validation Loss: 1.2252, Accuracy: 0.60
Epoch [13/25], Training Loss: 1.3309, Validation Loss: 1.1415, Accuracy: 0.65
Epoch [14/25], Training Loss: 1.2294, Validation Loss: 1.1449, Accuracy: 0.64
Epoch [15/25], Training Loss: 1.2089, Validation Loss: 1.1178, Accuracy: 0.65
Epoch [16/25], Training Loss: 1.2079, Validation Loss: 1.0444, Accuracy: 0.67
Epoch [17/25], Training Loss: 1.1347, Validation Loss: 0.9823, Accuracy: 0.69
Epoch [18/25], Training Loss: 1.0531, Validation Loss: 0.9814, Accuracy: 0.70
Epoch [19/25], Training Loss: 1.0258, Validation Loss: 0.9570, Accuracy: 0.68
Epoch [20/25], Training Loss: 1.0991, Validation Loss: 0.9922, Accuracy: 0.68
Epoch [21/25], Training Loss: 0.9804, Validation Loss: 0.7944, Accuracy: 0.76
Epoch [22/25], Training Loss: 0.9357, Validation Loss: 0.8653, Accuracy: 0.73
Epoch [23/25], Training Loss: 0.8872, Validation Loss: 0.7834, Accuracy: 0.75
Epoch [24/25], Training Loss: 0.9178, Validation Loss: 0.7402, Accuracy: 0.77
Epoch [25/25], Training Loss: 0.8668, Validation Loss: 0.7159, Accuracy: 0.77
Epoch [25/25], Training Loss: 0.8373, Validation Loss: 0.7949, Accuracy: 0.75
```

Finally, The model shows consistent improvement, as evidenced by the decreasing training and validation losses and the increasing accuracy. Starting with a training loss of 2.4877 and an accuracy of 0.25 in the first epoch, the model achieves a significant gain by the 25th epoch, reducing the training loss to 0.8373 and increasing the accuracy to **0.77 (77%)**. The validation accuracy of **77%** is quite good, and surely can be increased with additional fine-tuning of the model, more advanced data augmentation, increasing the dataset size, or perhaps incorporating ensemble methods to combine the strengths of multiple models. Further hyperparameter optimization, including adjusting the learning rate, increasing the complexity of the model, or employing techniques like dropout and batch normalization, could also contribute to performance improvements.

Thank You!