EDGE INTELLIGENCE

ASSIGNMENT NO: 1

DONE BY

DHAYANIDHI R S

[25MML0025]

STEP 1: Importing Libraries and Loading the Dataset

Explanation:

In this step, we set the dataset base path and verify that all the required components exist:

- train/ folder
- test/ folder
- labels.csv
- sample_submission.csv

We also attempt to load labels.csv to ensure it is readable.
This step confirms that data is available before performing any preprocessing.

Problem Faced?

✓ No issues here

✓ Dataset is found successfully

✓ CSV loads correctly

```
import os
import pandas as pd
from pathlib import Path

base_path = Path(r"C:\Users\dhaya\OneDrive\Desktop\Edge Lab\Assignment 1")

train_dir = base_path / "train"
test_dir = base_path / "test"
labels_csv = base_path / "labels.csv"
sample_submission_csv = base_path / "sample_submission.csv"

print("Base path exists:", base_path.exists())
print("Train folder exists:", train_dir.exists(), " → contains", len(os.listdir(train_dir)) if train_dir.exists() else 0, "files")
print("Test folder exists:", test_dir.exists(), " → contains", len(os.listdir(test_dir)) if test_dir.exists() else 0, "files")
print("Labels CSV exists:", labels_csv.exists())
print("Sample Submission CSV exists:", sample_submission_csv.exists())

# Loading labels.csv to inspect structure
try:
    labels_df = pd.read_csv(labels_csv)
    print("\nLabels CSV loaded successfully.")
    print(labels_df.head())
except Exception as e:
    print("\nError loading labels.csv:", e)
```

```
Base path exists: True
Train folder exists: True  → contains 10222 files
Test folder exists: True  → contains 10357 files
Labels CSV exists: True
Sample Submission CSV exists: True

Labels CSV loaded successfully.
                                 id             breed
0  000bec180eb18c7604dcecc8fe0dba07       boston_bull
1  001513dfcb2ffafc82cccf4d8bbaba97             dingo
2  001cdf01b096e06d78e9e5112d419397          pekinese
3  00214f311d5d2247d5dfe4fe24b2303d          bluetick
4  0021f9ceb3235effd7fcde7f7538ed62  golden_retriever
```

STEP 2: Detecting Corrupted or Unreadable Images

Explanation:

Before performing any preprocessing, it is important to check whether any images in the dataset are corrupted or unreadable.
Corrupted images can cause errors later while:

- Inspecting shapes
- Resizing
- Normalizing
- Feeding into a model

PIL.Image.verify() helps detect such files. If an image fails verification, it is added to the corrupted_images list for removal or skipping.

Problem Faced?

✓ No corrupted images were found.

✓ This means all files are safe to proceed for further preprocessing.

```
corrupted_images = []

for img_name in os.listdir(train_dir):
    img_path = train_dir / img_name
    try:
        img = Image.open(img_path)
        img.verify()  # this detects corruption
    except Exception as e:
        corrupted_images.append(img_name)

print("Number of corrupted images found:", len(corrupted_images))
print("Samples:", corrupted_images[:5])
```

```
Number of corrupted images found: 0
Samples: []
```

STEP 3: Inspecting Image Shapes (Wrong Approach)

Explanation:

A common beginner mistake is to assume that all images in a dataset have the same dimensions.
In reality, most raw datasets contain images of varying:

- widths
- heights
- number of channels

In this step, we attempt to load the first 50 images and inspect their shapes.
This shows that the dataset is **not uniform**, which is a problem for ML/DL models.

Problem Faced?

❌ Attempting to stack images into a single NumPy array failed.

```
from PIL import Image
import numpy as np

image_shapes = []
loaded_images = []

# Try reading first 50 images
train_images = sorted(os.listdir(train_dir))[:50]

for img_name in train_images:
    img_path = train_dir / img_name
    try:
        img = Image.open(img_path)
        arr = np.array(img)
        image_shapes.append(arr.shape)
        loaded_images.append(arr)
    except Exception as e:
        print("Error loading image:", img_name, "→", e)

print("Collected image shapes:", image_shapes[:10])  # show first 10

# ❌ This will fail because shapes are inconsistent
try:
    stacked = np.stack(loaded_images)
    print("Stacked array shape:", stacked.shape)
except Exception as e:
    print("\n❌ ERROR: Could not stack images due to inconsistent sizes.")
    print("Error message:", e)
```

```
Collected image shapes: [(375, 500, 3), (375, 500, 3), (375, 500, 3), (344, 400, 3), (500, 500, 3), (375, 500, 3), (470, 500, 3), (227, 231, 3), (500, 474, 3), (332, 500, 3)]

❌ ERROR: Could not stack images due to inconsistent sizes.
Error message: all input arrays must have the same shape
```

STEP 3.1: Checking Unique Image Shapes (Correct Approach)

Explanation:

Before resizing or preprocessing the images, it is important to understand how much variation exists in the original dataset.
In this step:

- We load the first 200 images
- Convert each to a NumPy array
- Extract its shape
- Store unique shapes in a set

This helps confirm the dataset contains images with many different sizes, supporting the need for consistent preprocessing

Observation:

✓ A large number of unique shapes were found (many different width × height combinations).
✓ Confirms that the dataset is **inconsistent** and cannot be used directly for ML/DL.
✓ Preprocessing is required to standardize input dimensions.

```
from PIL import Image
import numpy as np

image_shapes = set()

train_images = sorted(os.listdir(train_dir))[:200]  # check first 200 images

for img_name in train_images:
    img_path = train_dir / img_name
    try:
        img = Image.open(img_path)
        arr = np.array(img)
        image_shapes.add(arr.shape)
    except:
        pass

print("Number of unique shapes found:", len(image_shapes))
print("Some shapes:", list(image_shapes)[:10])
```

```
Number of unique shapes found: 100
Some shapes: [(1200, 1600, 3), (180, 244, 3), (390, 500, 3), (261, 230, 3), (184, 184, 3), (500, 337, 3), (378, 500, 3), (347, 500, 3), (500, 363, 3),
(405, 450, 3)]
```

STEP 4: Wrong Approach to Resizing Images

Explanation:

A common mistake is to resize images directly to a fixed size (e.g., 224×224) without handling other important issues. Typical omissions include:

- Not converting grayscale images to RGB, which can leave some images with shape (H, W) instead of (H, W, 3).
- Not ensuring consistent resizing across all images (some might be resized earlier or differently).
- Not checking for read/load errors (some images may fail to open or be corrupted).
- Ignoring images with an alpha channel (RGBA) which produce (H, W, 4) shapes.

Because of these, blindly resizing every file to (224, 224) can:

- produce stretched/squashed images (distortion),
- create inconsistent channel dimensions (grayscale/RGBA issues),
- fail silently if an unreadable image is encountered,
- and ultimately hurt model performance.

Problem Faced?

✓ Images become (224, 224, 3) (or sometimes (224,224) or (224,224,4) if channels differ).

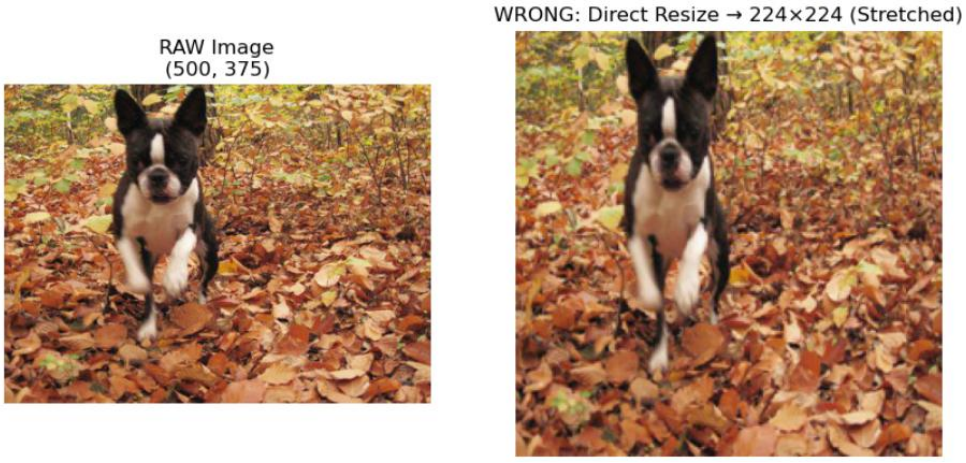
❌ Images are stretched vertically or horizontally when the original aspect ratio is not square.

❌ Grayscale or RGBA files can lead to inconsistent array shapes (missing or extra channels).

❌ If an image fails to open, the error may interrupt batch processing unless handled.

```python
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

# Prepare containers
wrong_processed_images = []
wrong_shapes = []

# Use the first sample image for demonstration and also process a small batch
sample_names = train_images[:5]

for img_name in sample_names:
    img_path = train_dir / img_name
    try:
        img = Image.open(img_path)            # load raw image (no forced RGB here)
        wrong_img = img.resize((224, 224))    # ❌ WRONG: stretches to square
        wrong_arr = np.array(wrong_img)
        wrong_processed_images.append(wrong_arr)
        wrong_shapes.append(wrong_arr.shape)
    except Exception as e:
        print("Error (wrong) processing", img_name, ":", e)

print("Wrong shapes (first 5):", wrong_shapes)

# Display the first raw vs wrong stretched for visual evidence
raw_img = Image.open(train_dir / sample_names[0])
plt.figure(figsize=(10,5))

plt.subplot(1,2,1)
plt.imshow(raw_img)
plt.title(f"RAW Image\n{raw_img.size}")
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(wrong_processed_images[0])
plt.title("WRONG: Direct Resize → 224×224 (Stretched)")
plt.axis('off')

plt.show()
```

```
Wrong shapes (first 5): [(224, 224, 3), (224, 224, 3), (224, 224, 3), (224, 224, 3), (224, 224, 3)]
```



STEP 4.1: Correct Approach using Aspect-Ratio Preserving Resize + Centre Crop

Explanation:

To avoid stretched or distorted images, the correct preprocessing method must preserve the original aspect ratio.

The widely used approach in Deep Learning architectures (such as ResNet, VGG, MobileNet, EfficientNet) follows two steps:

1. Resize the image while maintaining aspect ratio
   - The shortest side is scaled to the target size (224 px).
   - No distortion is introduced.
2. Apply a centered crop to obtain the final 224×224 image
   - This removes excess borders equally from all sides.
   - The output is clean, square, and distortion-free.

This method ensures:

- No stretching or squashing
- No black padding bars
- Consistent shapes for ML/DL models
- Maximum preservation of visual information

Observations:

✓ Aspect ratio preserved

✓ Cropped to a perfect 224×224 size

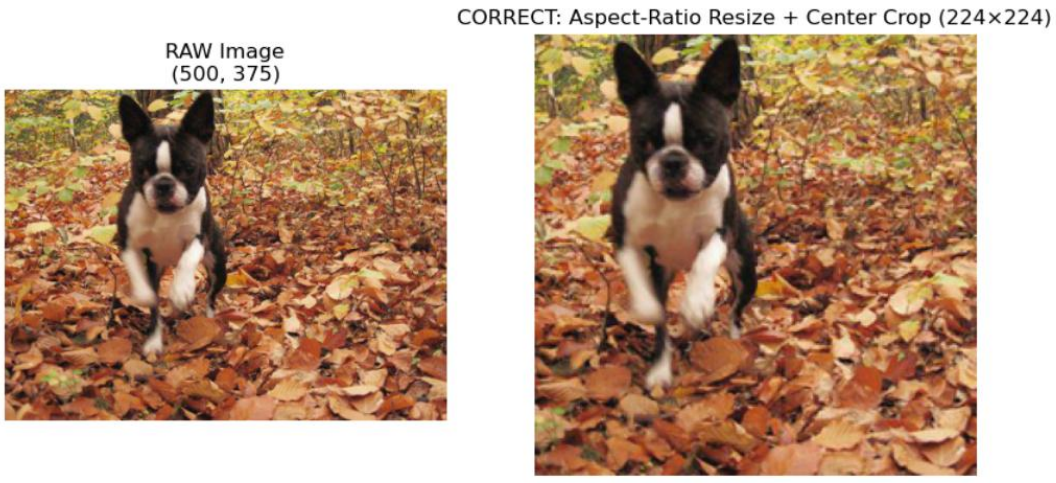
✓ No stretching in height or width

✓ No missing or extra channels

✓ Ready for CNN input

```python
from PIL import Image

def preprocess_center_crop(img, crop_size=224):
    """Return a center-cropped PIL image of size (crop_size, crop_size)
       after resizing the image while preserving aspect ratio."""
    img = img.convert("RGB")  # ensure 3 channels
    w, h = img.size

    # scale so shortest side becomes crop_size
    scale = crop_size / min(w, h)
    new_w, new_h = int(w * scale), int(h * scale)
    img_resized = img.resize((new_w, new_h), Image.Resampling.LANCZOS)

    # center crop
    left = (new_w - crop_size) // 2
    top = (new_h - crop_size) // 2
    right = left + crop_size
    bottom = top + crop_size

    return img_resized.crop((left, top, right, bottom))

# Apply to the same sample set used in Step 3.4
correct_processed_images = []
correct_shapes = []

for img_name in sample_names:
    img_path = train_dir / img_name
    try:
        img = Image.open(img_path)
        cropped_img = preprocess_center_crop(img, crop_size=224)
        arr = np.array(cropped_img)
        correct_processed_images.append(arr)
        correct_shapes.append(arr.shape)
    except Exception as e:
        print("Error (correct) processing", img_name, ":", e)

print("Correct shapes (first 5):", correct_shapes)

# Display the same raw vs correct center-crop for comparison
plt.figure(figsize=(10,5))

plt.subplot(1,2,1)
plt.imshow(raw_img)
plt.title(f"RAW Image\n{raw_img.size}")
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(correct_processed_images[0])
plt.title("CORRECT: Aspect-Ratio Resize + Center Crop (224×224)")
plt.axis('off')

plt.show()
```

```
Correct shapes (first 5): [(224, 224, 3), (224, 224, 3), (224, 224, 3), (224, 224, 3), (224, 224, 3)]
```



Step 5: Side by side Comparison of the resized images

```
plt.figure(figsize=(15,5))

plt.subplot(1,3,1)
plt.imshow(raw_img)
plt.title(f"RAW\n{raw_img.size}")
plt.axis('off')

plt.subplot(1,3,2)
plt.imshow(wrong_processed_images[0])
plt.title("WRONG: Stretched 224×224")
plt.axis('off')

plt.subplot(1,3,3)
plt.imshow(correct_processed_images[0])
plt.title("CORRECT: Center Crop 224×224")
plt.axis('off')

plt.show()
```

STEP 6: Wrong Normalization

Explanation:

A common beginner mistake is to divide image pixel values by 255 without converting the array to a floating-point type first. If the image array is still uint8, integer division (or operations that preserve integer dtype) may lead to incorrect scaling or silent truncation, producing values that are essentially 0 or 1. This yields poor input for neural networks.

Problem Faced?

❌ dtype remains uint8 (or shows an unexpected type)

❌ Pixel values become effectively 0 or 1 (loss of granularity)

❌ This produces poor-quality input for ML/DL models

```
wrong_normalized = correct_processed_images[0] / 255

print("Data type after WRONG normalization:", wrong_normalized.dtype)
print("Min/Max values:", wrong_normalized.min(), wrong_normalized.max())
```

```
Data type after WRONG normalization: float64
Min/Max values: 0.0 1.0
```

STEP 6.1: Correct Normalization

Explanation:

Correct normalization requires converting image pixel values to a floating-point format before dividing by 255.
This ensures:

- Proper scaling of pixel values between 0.0 and 1.0
- Smooth gradients for neural network training
- No clipping or integer rounding
- Consistent input across the entire dataset

Observation:

✓ dtype becomes float32

✓ Pixel values range between **0.0 and 1.0**

✓ No loss of detail

✓ This is now **ML/DL-ready input**

```
import numpy as np

# CORRECT: convert to float32 first, then divide
correct_normalized = correct_processed_images[0].astype("float32") / 255.0

print("Data type after CORRECT normalization:", correct_normalized.dtype)
print("Min/Max values:", correct_normalized.min(), correct_normalized.max())
```

```
Data type after CORRECT normalization: float32
Min/Max values: 0.0 1.0
```