### **Introduction:**

There are two main types of image quality assessment: **Full-Reference (FR-IQA)** and **No-Reference (NR-IQA)**. Since the user does not have a reference image, **NR-IQA** is the most suitable approach.

One common NR-IQA method is **NIMA (Neural Image Assessment)**, which uses a pre-trained **MobileNet** model fine-tuned on the **AVA dataset** to evaluate both the aesthetic and technical quality of images. Another widely used method is **BRISQUE (Blind/Referenceless Image Spatial Quality Evaluator)**, which takes a simpler, more traditional approach by extracting statistical features from the image rather than relying on deep learning.

### **Differences Between NIMA and BRISQUE:**

* **NIMA** leverages deep learning to predict image quality based on large-scale training data.
* **BRISQUE** uses handcrafted statistical features and can be implemented efficiently with libraries like **OpenCV** and **scikit-learn**, without requiring a deep neural network.

### **Why Use BRISQUE with PyTorch and PIQ?**

To make the implementation of BRISQUE easier, we will use **PyTorch** along with the **PIQ (Perceptual Image Quality) library**. PIQ is a specialized library designed for image quality assessment tasks in PyTorch. It provides ready-to-use implementations of several NR-IQA methods, including BRISQUE, allowing for seamless integration with deep learning workflows.

With **PIQ**, we can efficiently compute the **BRISQUE score** using PyTorch tensors, enabling easy processing of images within deep learning pipelines. Additionally, **PyTorch** provides flexibility, GPU acceleration, and a rich ecosystem for handling image data.

Since BRISQUE in **PIQ** returns a tensor where **higher values indicate lower quality**, we can use it as a straightforward metric to assess image quality. Now, let’s move forward with implementing BRISQUE using **PIQ** and **PyTorch**.

### **Implementation of BRISQUE Using PyTorch and PIQ**

Now that we've chosen **BRISQUE** as our image quality assessment method, let's walk through its implementation using **PyTorch** and **PIQ**.

#### **Step-by-Step Breakdown of the Code**

1. **Import the Required Libraries**
   * cv2: Used for reading and processing images.
   * torch: Used to handle tensor operations for PyTorch.
   * piq: Provides the brisque function to compute the image quality score.
   * halo: A small library that displays a loading spinner for better user experience.
2. **Define the assess\_image\_quality Function**
   * This function takes an image path as input and returns the BRISQUE quality score.
   * It starts a **loading spinner** to indicate progress.
   * Reads the image using **OpenCV (cv2.imread)**.
   * Converts it to **RGB** format since OpenCV loads images in **BGR** format.
   * Converts the image to a **PyTorch tensor**, normalizing pixel values between **0 and 1**.
   * Calls **piq.brisque** to calculate the BRISQUE score.
   * Stops the spinner and returns the score.
3. **Main Execution (\_\_main\_\_)**
   * Displays a startup message.
   * Checks for a valid **command-line argument** (image path).
   * Calls assess\_image\_quality and prints the BRISQUE score with an **interpretation guide**.

Complete Code:

| import cv2  import torch  import piq  from halo import Halo # Importing Halo for loading spinner  def assess\_image\_quality(image\_path: str):  """  Assess image quality using the BRISQUE metric.  BRISQUE (Blind/Referenceless Image Spatial Quality Evaluator) is a no-reference  image quality assessment metric. Lower scores indicate better quality.  Parameters:  image\_path (str): Path to the image file.  Returns:  float: BRISQUE score (lower is better).  """  # Start a spinner animation while processing the image  spinner = Halo(text='🔄 Calculating result...', spinner='dots')  spinner.start()  # Read the image using OpenCV (default format is BGR)  img = cv2.imread(image\_path)  # Convert from BGR to RGB since OpenCV loads images in BGR format  img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)  # Convert the image into a PyTorch tensor and normalize pixel values to [0,1]  # - .permute(2, 0, 1) → Changes shape from [H, W, C] to [C, H, W]  # - .unsqueeze(0) → Adds a batch dimension, making it [1, C, H, W]  img\_tensor = torch.tensor(img).permute(2, 0, 1).unsqueeze(0).float() / 255.0  # Compute the BRISQUE score  # - data\_range=1.0 ensures that the function correctly handles normalized images  brisque\_score = piq.brisque(img\_tensor, data\_range=1.0)  # Stop the spinner after the calculation is completed  spinner.succeed("✅ Calculation completed!")  return brisque\_score.item()  if \_\_name\_\_ == "\_\_main\_\_":  import sys  # Start a spinner animation to indicate the script is launching  spinner = Halo(text="🚀 Starting Image Quality Assessment...", spinner="dots")  spinner.start()  # Ensure the script is executed with an image path argument  if len(sys.argv) != 2:  spinner.fail("❌ Error: Missing image path!")  print("Usage: python image\_quality.py <image\_path>")  sys.exit(1)  # Stop spinner after validation  spinner.succeed("✅ Image Quality Assessment started!")  # Get the image path from command-line arguments  image\_path = sys.argv[1]  # Run the assessment function  score = assess\_image\_quality(image\_path)  # Print results  print("\n Image Quality Assessment Results:")  print(f" BRISQUE Score: {score:.2f} (Lower is better)\n")  # Interpretation guide for BRISQUE scores  print("===============================")  print("Interpretation Guide:")  print("0 - 20 : Excellent quality")  print("20 - 40 : Good quality")  print("40 - 60 : Fair quality")  print("60 - 80 : Poor quality")  print("80 - 100: Very poor quality")  print("===============================") |
| --- |

Before computing **BIQSUE**, the image is converted into a **Tensor** following these steps:

1. The image is **converted to a Tensor** using torch.tensor(img).
2. The dimensions are **reordered** using .permute(2, 0, 1), changing the shape from **[Height, Width, Channels]** to **[Channels, Height, Width]**.
3. A new **batch dimension** is added using .unsqueeze(0), transforming the shape to **[Batch, Channels, Height, Width]**.
4. The image is **normalized** by dividing pixel values by **255.0**, scaling them to the **[0,1]** range.

### **What is a Tensor?**

A **Tensor** is a data structure similar to **NumPy arrays**, but it offers several advantages:  
 - Supports computations on the **GPU** for faster processing.  
 - Enables **efficient mathematical operations** for deep learning.  
 - Forms the **core data structure** in PyTorch for handling and transforming data.

When an image is read using **cv2.imread()**, it is stored as a **NumPy array** with dimensions **[Height, Width, Channels]**.

However, **piq.brisque()** requires the image to be in a specific **Tensor format**

### **Why Does piq.brisque() Require a Tensor?**

**When computing the BRISQUE score using piq.brisque(), the image must be a PyTorch Tensor for several reasons:**

**Compatibility with PyTorch → piq.brisque() only accepts torch.Tensor, not NumPy arrays.  
 GPU Acceleration → Converting to a Tensor allows computations to be performed on the GPU, significantly improving performance.  
 Easier Data Processing → Tensors make it simpler to apply transformations like scaling, normalization, and augmentation.  
 Higher Precision → Using .float() / 255.0 ensures pixel values are stored as floating-point numbers, improving the accuracy of quality assessments.**

### **What Happens If We Don't Convert the Image to a Tensor?**

**If we try to pass a NumPy array directly into piq.brisque(), it will throw an error, as the function requires input of type torch.Tensor.**

**In short, converting to a Tensor is not just a suggestion—it’s a requirement for piq.brisque() to work correctly!**

### **Alternative Methods to BRISQUE**

**If you're looking for image quality assessment using AI or Deep Learning, there are more advanced alternatives:**

| **Alternative** | **Description** | **Accuracy** | **Performance** |
| --- | --- | --- | --- |
| **NIMA (Neural Image Assessment)** | **Uses a CNN model to evaluate both aesthetic and technical quality.** | **✅ Higher Accuracy** | **❌ Slower** |
| **MUSIQ (Multi-scale Image Quality)** | **A modern Transformer-based model for image quality assessment.** | **✅ Highest Accuracy** | **❌ Requires High Resources** |
| **NIQE (Natural Image Quality Evaluator)** | **Similar to BRISQUE but does not rely on an SVM model.** | **⭕ Moderate Accuracy** | **✅ Faster** |

### **Potential Enhancements and Future Improvements**

* **Combining Multiple Methods: BRISQUE can be integrated with NIMA or NIQE for more accurate assessments.**
* **Output Normalization: Applying normalization techniques can make scores more intuitive and easier to interpret.**
* **Leveraging Deep Learning: Instead of piq.brisque, a pre-trained CNN model such as MUSIQ or NIMA could be used for more advanced quality assessment.**

**Extra:To improve performance and ease of use, a GUI version has been added, allowing users to upload an image and display the result directly.**

| **import cv2**  **import torch**  **import piq**  **import threading**  **import tkinter as tk**  **from tkinter import filedialog, Label, Button**  **from PIL import Image, ImageTk**  **def assess\_image\_quality(image\_path: str):**  **"""**  **Assess image quality using the BRISQUE metric.**  **BRISQUE (Blind/Referenceless Image Spatial Quality Evaluator) is a no-reference**  **image quality assessment metric. Lower scores indicate better quality.**  **Parameters:**  **image\_path (str): Path to the image file.**  **Returns:**  **float: BRISQUE score (lower is better).**  **"""**  **# Read the image using OpenCV (default format is BGR)**  **img = cv2.imread(image\_path)**  **# Convert from BGR to RGB since OpenCV loads images in BGR format**  **img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)**  **# Convert the image into a PyTorch tensor and normalize pixel values to [0,1]**  **# - .permute(2, 0, 1) → Changes shape from [H, W, C] to [C, H, W]**  **# - .unsqueeze(0) → Adds a batch dimension, making it [1, C, H, W]**  **img\_tensor = torch.tensor(img).permute(2, 0, 1).unsqueeze(0).float() / 255.0**  **# Compute the BRISQUE score**  **# - data\_range=1.0 ensures that the function correctly handles normalized images**  **brisque\_score = piq.brisque(img\_tensor, data\_range=1.0)**  **return brisque\_score.item()**  **def select\_image():**  **"""**  **Open a file dialog to select an image and start the quality assessment process.**  **"""**  **# Open file dialog and get image path**  **file\_path = filedialog.askopenfilename(filetypes=[("Image files", "\*.jpg;\*.jpeg;\*.png")])**  **# If no file is selected, return**  **if not file\_path:**  **return**  **# Update result label while processing**  **result\_label.config(text="🔄 Calculating...")**  **# Start processing in a separate thread to avoid GUI freezing**  **threading.Thread(target=calculate\_quality, args=(file\_path,), daemon=True).start()**  **# Load and display the selected image**  **img = Image.open(file\_path)**  **img.thumbnail((250, 250)) # Resize image for display**  **img = ImageTk.PhotoImage(img)**  **image\_label.config(image=img)**  **image\_label.image = img # Keep reference to avoid garbage collection**  **def calculate\_quality(image\_path):**  **"""**  **Calculate the BRISQUE score and update the result label with the score and interpretation guide.**  **Parameters:**  **image\_path (str): Path to the selected image.**  **"""**  **# Compute the BRISQUE score**  **score = assess\_image\_quality(image\_path)**  **# Format result output**  **result\_text = f"BRISQUE Score: {score:.2f} (Lower is better)\n\n"**  **result\_text += "Score Interpretation:\n"**  **result\_text += "============================\n"**  **result\_text += "0 - 20 : Excellent quality\n"**  **result\_text += "20 - 40 : Good quality\n"**  **result\_text += "40 - 60 : Fair quality\n"**  **result\_text += "60 - 80 : Poor quality\n"**  **result\_text += "80 - 100 : Very poor quality\n"**  **result\_text += "============================\n"**  **# Update the GUI with the result**  **result\_label.config(text=result\_text)**  **# GUI Setup**  **root = tk.Tk()**  **root.title("Image Quality Assessment")**  **root.geometry("600x450")**  **# Title Label**  **Label(root, text="Upload an Image to Assess Quality", font=("Arial", 12)).pack(pady=10)**  **# Select Image Button**  **Button(root, text="📂 Select Image", command=select\_image).pack(pady=10)**  **# Image Display Label**  **image\_label = Label(root)**  **image\_label.pack()**  **# Result Display Label**  **result\_label = Label(root, text="", font=("Arial", 12), wraplength=350)**  **result\_label.pack(pady=10)**  **# Start GUI Main Loop**  **root.mainloop()** |
| --- |