Assignment-01:

Present the different types of feature selection, extraction, and visualization techniques for image processing.

Image Upload and Preprocessing

Technique: File upload and basic image preprocessing (conversion to RGB and Grayscale).

Purpose: Provides a user-friendly way to input custom images and prepares them for analysis by converting formats appropriately.

Functionality:

Utilizes files.upload() to allow image selection from the local system.

Reads the image using cv2.imread() (which defaults to BGR format).

Converts the image to RGB for accurate display using matplotlib.

Converts to Grayscale for use in many feature extraction techniques (e.g., edges, texture).

Applications:

Forms the foundation for applying multiple image processing techniques.

Supports custom experimentation and dynamic input handling in educational or research notebooks.

Enables consistent image preprocessing for reproducibility across different features.

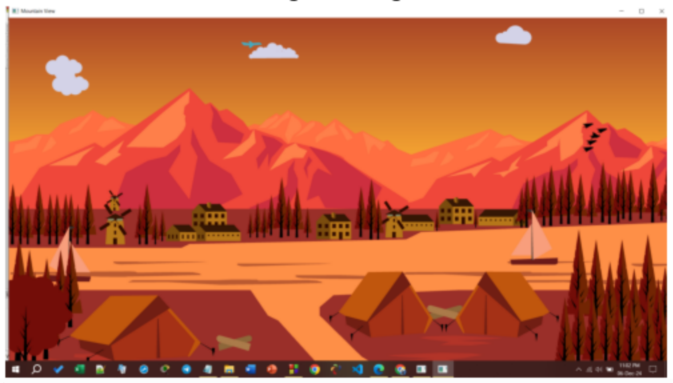
```
from google.colab import files
import cv2
import matplotlib.pyplot as plt

# Step 1: Upload the image
print("Upload your image file:")
uploaded = files.upload()

# Step 2: Load the uploaded image
image_path = list(uploaded.keys())[0]
image = cv2.imread(image_path)  # BGR format
image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Convert to RGB
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)  # Convert to Grayscale

# Step 3: Display the RGB image
plt.imshow(image_rgb)
plt.title("Original Image")
plt.axis('off')
plt.show()
```

```
Upload your image file:
```
Choose Files Screenshot… 230301.png

- **Screenshot 2024-12-06 230301.png**(image/png) - 517145 bytes, last modified: 12/6/2024 - 100% done

```
Saving Screenshot 2024-12-06 230301.png to Screenshot 2024-12-06 230301.png
```

Original Image



Feature 1: Color Histogram

Technique: Color Histogram

Purpose: A color histogram represents the distribution of pixel intensities for each color channel (Red, Green, and Blue). It provides a statistical summary of the colors in an image.

Functionality:

For each color channel (R, G, B), the histogram counts how many pixels correspond to each intensity level (ranging from 0 to 255).

The histograms for each color channel are computed separately using cv2.calcHist() and visualized to show how the image's color intensities are distributed.

Applications:

Image retrieval: Color histograms can be used to retrieve images with similar color distributions.

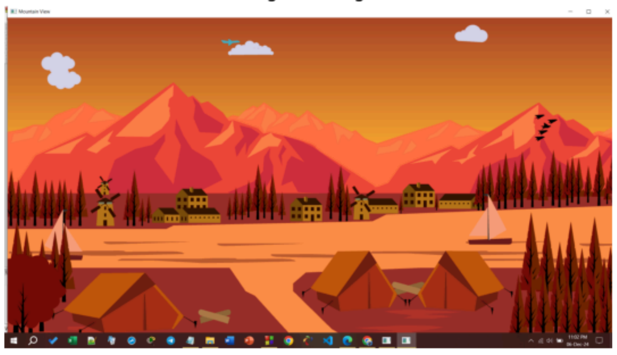
Color-based classification: In tasks like segmenting an object based on its color.

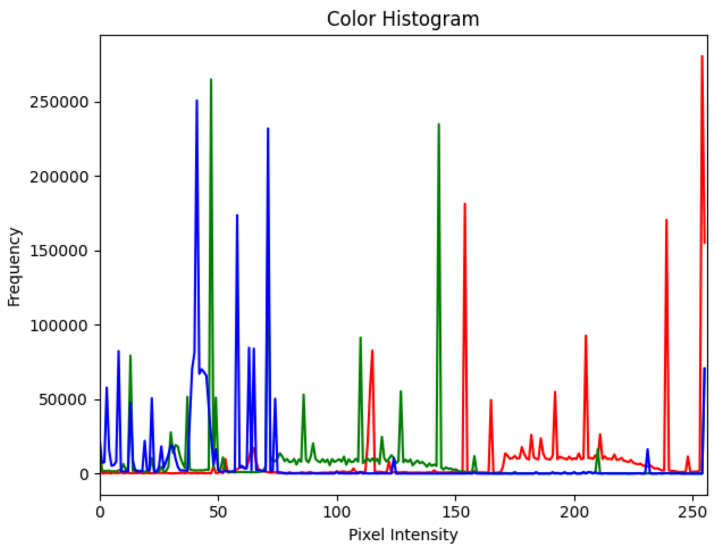
Dominant color detection: Identifying the most prominent colors within an image, which is useful in various fields like design, fashion, and art.

```
import cv2
import matplotlib.pyplot as plt

# Function to extract and plot the color histogram
def extract_color_histogram(image_rgb):
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Display the original image
    axes[0].imshow(image_rgb)
    axes[0].set_title("Original Image")
    axes[0].axis('off')

    # Compute and plot histogram for each channel
    colors = ('r', 'g', 'b')
    for i, color in enumerate(colors):
        hist = cv2.calcHist([image_rgb], [i], None, [256], [0, 256])
        axes[1].plot(hist, color=color)

    axes[1].set_xlim([0, 256])
    axes[1].set_title("Color Histogram")
    axes[1].set_xlabel("Pixel Intensity")
    axes[1].set_ylabel("Frequency")

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

```
# Run the function
extract_color_histogram(image_rgb)
```

Original Image





Double-click (or enter) to edit

Feature 2: Canny Edge Detection

Technique: Canny Edge Detection

Purpose: The Canny edge detector is used to identify the boundaries of objects within an image. It helps highlight areas where the pixel intensities change sharply, providing an outline of objects.

Functionality:

The algorithm applies gradient-based edge detection by calculating the intensity gradient at each pixel.

By setting low and high threshold values, Canny detects both strong and weak edges, retaining the strong ones and suppressing the weak edges, which are often noise.

The result is a binary image where edges are represented by white pixels, and the background is black.

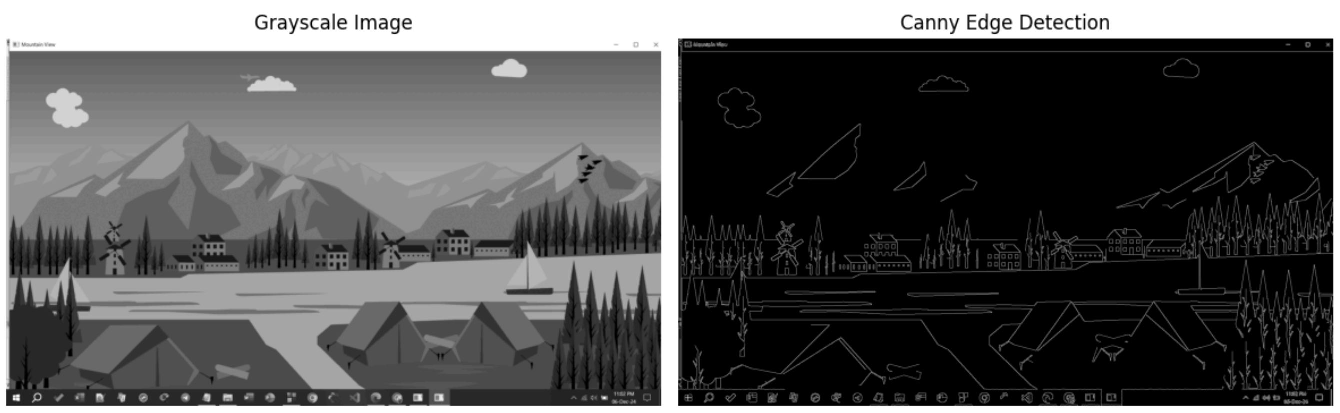
Applications:

Object detection: Useful for identifying and segmenting objects in images.

Shape analysis: Helps to detect contours and boundaries for shape recognition.

Lane detection: In autonomous vehicles, Canny edge detection helps identify the lanes on the road.

```
def extract_edges(gray_image):
    # Apply Canny Edge Detection
    edges = cv2.Canny(gray_image, 100, 200)

    # Plot original grayscale image and edge-detected output
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    axes[0].imshow(gray_image, cmap='gray')
    axes[0].set_title("Grayscale Image")
    axes[0].axis('off')

    axes[1].imshow(edges, cmap='gray')
    axes[1].set_title("Canny Edge Detection")
    axes[1].axis('off')

    plt.tight_layout()
    plt.show()

# Run the function
extract_edges(gray_image)
```



Feature 3: Local Binary Patterns (LBP)

Technique: Local Binary Patterns (LBP)

Purpose: LBP is used for texture classification by comparing each pixel's intensity with its neighbors to form a binary pattern. It captures local texture information in an image.

Functionality:

For each pixel, the intensity of its surrounding neighbors is compared to its own intensity, and a binary code is formed.

The binary pattern is converted to a decimal value, which is then used to represent the local texture of that pixel.

This method works well for textures that are homogeneous within local regions.

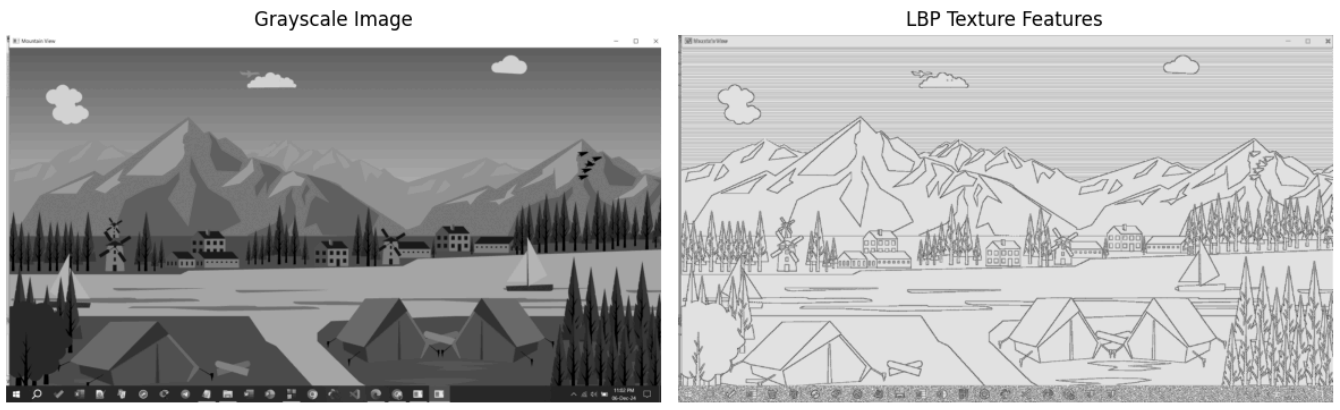
Applications:

Texture classification: Detecting and classifying textures (e.g., distinguishing between rough or smooth surfaces).

Facial recognition: LBP can be used to capture the unique texture features of faces, aiding in recognition.

Surface inspection: Detecting defects in manufactured products, such as scratches or bumps.

```
from skimage.feature import local_binary_pattern

def extract_texture_features(gray_image):
    # Parameters for LBP
    radius = 1
    n_points = 8 * radius

    # Compute LBP
    lbp = local_binary_pattern(gray_image, n_points, radius, method="uniform")

    # Plot original grayscale and LBP image
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    axes[0].imshow(gray_image, cmap="gray")
    axes[0].set_title("Grayscale Image")
    axes[0].axis('off')

    axes[1].imshow(lbp, cmap="gray")
    axes[1].set_title("LBP Texture Features")
    axes[1].axis('off')

    plt.tight_layout()
    plt.show()

# Run the function
extract_texture_features(gray_image)
```



Feature 4: Scale-Invariant Feature Transform (SIFT)

Technique: Scale-Invariant Feature Transform (SIFT)

Purpose: SIFT is a feature extraction technique that identifies keypoints in an image that are invariant to scale, rotation, and illumination. These keypoints are distinctive and can be used to match objects across different images.

Functionality:

SIFT detects keypoints in an image by looking for local extrema in a scale-space representation of the image. These keypoints are stable across different scales and rotations.

Descriptors are computed around each keypoint, representing the local image patch in a robust way.

These keypoints can be matched between images for tasks like object recognition or panorama stitching.

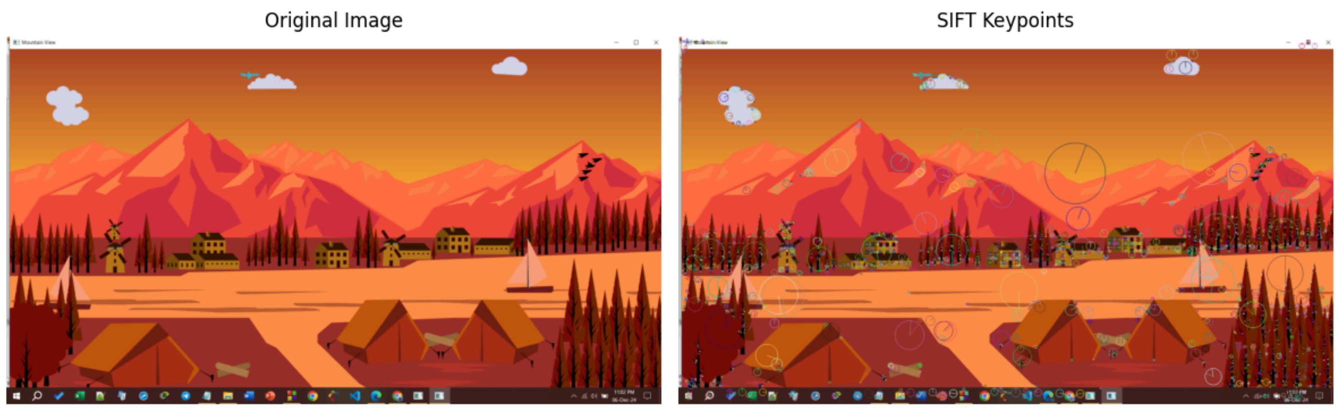
Applications:

Object recognition: Identifying objects in images regardless of scale or rotation.

Image stitching: Combining multiple images to create a panorama by matching keypoints across them.

Feature-based image matching: Matching parts of images to find correspondences (e.g., in stereo vision systems).

```
def extract_keypoints(gray_image, image_rgb):
    # Initialize SIFT detector
    sift = cv2.SIFT_create()
    # Detect keypoints and compute descriptors
    keypoints, descriptors = sift.detectAndCompute(gray_image, None)

    # Draw keypoints on the image
    sift_image = cv2.drawKeypoints(
        image_rgb, keypoints, None,
        flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
    )

    # Plot original and keypoint-detected image
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    axes[0].imshow(image_rgb)
    axes[0].set_title("Original Image")
    axes[0].axis('off')

    axes[1].imshow(sift_image)
    axes[1].set_title("SIFT Keypoints")
    axes[1].axis('off')

    plt.tight_layout()
    plt.show()

# Run the function
extract_keypoints(gray_image, image_rgb)
```



Feature 5: Histogram of Oriented Gradients (HOG)

Technique: Histogram of Oriented Gradients (HOG)

Purpose: bold text HOG is used to describe the shape of objects by capturing the distribution of gradient orientations within an image.

Functionality:

The image is divided into small cells (e.g., 8x8 pixels), and a histogram of gradient orientations is calculated within each cell.

These histograms are then normalized over larger blocks of cells to make the feature descriptor invariant to changes in illumination.

A feature vector is formed by concatenating these histograms from all the blocks, which captures the overall shape information of the image.

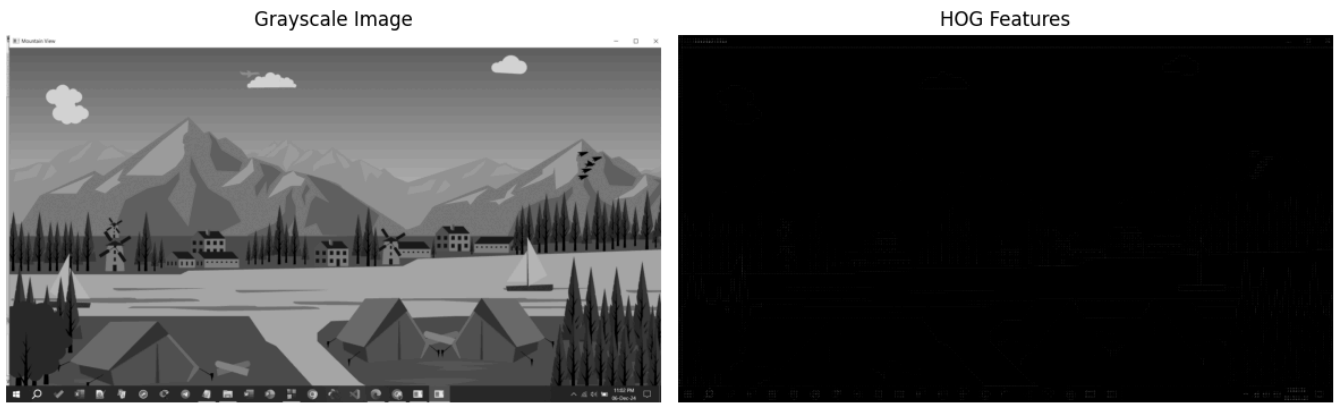
Applications:

Pedestrian detection: HOG features are commonly used in detecting people in images or video streams, especially in surveillance systems.

Object detection: Recognizing various objects in images based on shape and edge information.

Image classification: Classifying objects in images based on the shape and gradient patterns within them.

```
from skimage.feature import hog

def extract_hog_features(gray_image):
    # Compute HOG features and visualization
    hog_features, hog_image = hog(
        gray_image,
        orientations=9,
        pixels_per_cell=(8, 8),
        cells_per_block=(2, 2),
        visualize=True,
        channel_axis=None
    )
    # Plot original and HOG image
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    axes[0].imshow(gray_image, cmap="gray")
    axes[0].set_title("Grayscale Image")
    axes[0].axis('off')

    axes[1].imshow(hog_image, cmap="gray")
    axes[1].set_title("HOG Features")
    axes[1].axis('off')

    plt.tight_layout()
    plt.show()

# Run the function
extract_hog_features(gray_image)
```



Feature 6: Pre-Trained CNN (VGG16)

Technique: Pre-Trained Convolutional Neural Networks (VGG16)

Purpose: VGG16 is a pre-trained deep learning model that extracts high-level semantic features from images, providing a rich representation of objects and scenes.

Functionality:

The VGG16 network is pre-trained on a large dataset (ImageNet), which means it has learned to recognize a wide variety of features and objects.

The image is resized to 224x224 pixels to match the input size required by the VGG16 model.

The convolutional layers of the network extract features such as edges, textures, and object parts, which can then be used for classification or other tasks.

The fully connected layers (which perform classification) are excluded to get a rich set of features, known as feature maps.

Applications:

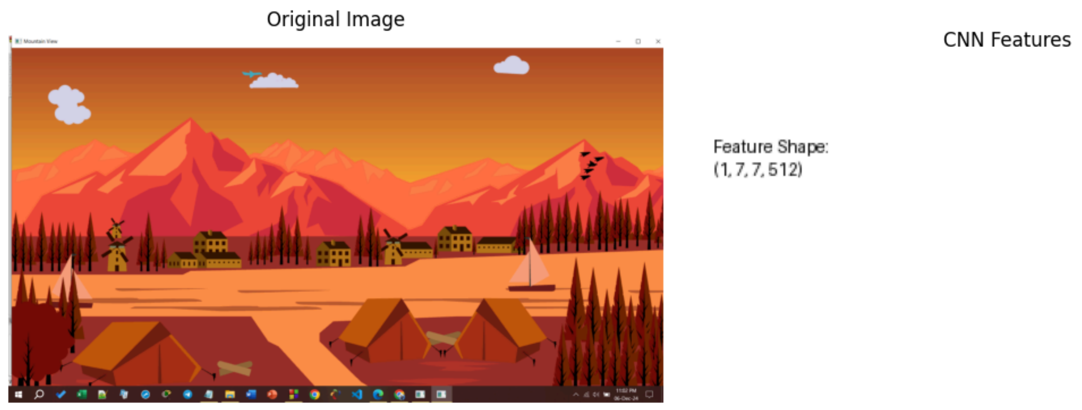
Image classification: Identifying objects or scenes in an image based on learned features.

Object detection: Identifying the presence of specific objects within an image.

Transfer learning: Using the pre-trained model for new tasks, such as fine-tuning the model for custom image classification problems.

```
from tensorflow.keras.applications import VGG16
from tensorflow.keras.applications.vgg16 import preprocess_input
import numpy as np
import cv2
from PIL import Image, ImageDraw
import matplotlib.pyplot as plt

def extract_cnn_features(image_rgb):
    # Load the pre-trained VGG16 model (exclude the top layers)
    model = VGG16(weights="imagenet", include_top=False)

    # Preprocess the image for VGG16 (resize to 224x224)
    resized_image = cv2.resize(image_rgb, (224, 224))
    image_array = np.expand_dims(resized_image, axis=0)
    image_array = preprocess_input(image_array)
    # Extract features from the model (excluding the fully connected layers)
    features = model.predict(image_array)

    # Create an image to display feature shape information
    text_image = Image.new('RGB', (400, 200), color='white')
    draw = ImageDraw.Draw(text_image)
    text_content = f"Feature Shape:\n{features.shape}"
    draw.text((20, 50), text_content, fill="black")

    # Convert the PIL image to a NumPy array for Matplotlib
    text_image_np = np.array(text_image)

    # Plot original image and the feature shape information
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    axes[0].imshow(image_rgb)
    axes[0].set_title("Original Image")
    axes[0].axis('off')

    axes[1].imshow(text_image_np)
    axes[1].set_title("CNN Features")
    axes[1].axis('off')

    plt.tight_layout()
    plt.show()

# Run the function
extract_cnn_features(image_rgb)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_n
58889256/58889256 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 896ms/step
```



Feature Shape:
(1, 7, 7, 512)

Feature 7: Contour Detection

Technique: Contour Detection

Purpose: Contour detection helps identify and outline the boundaries of objects in an image. It represents shapes and object outlines in a continuous curve.

Functionality:

Contours are detected by first thresholding the image to create a binary version, where the objects are distinct from the background.

The contours are then extracted by following the boundaries of the foreground objects, producing a series of points that represent these boundaries.

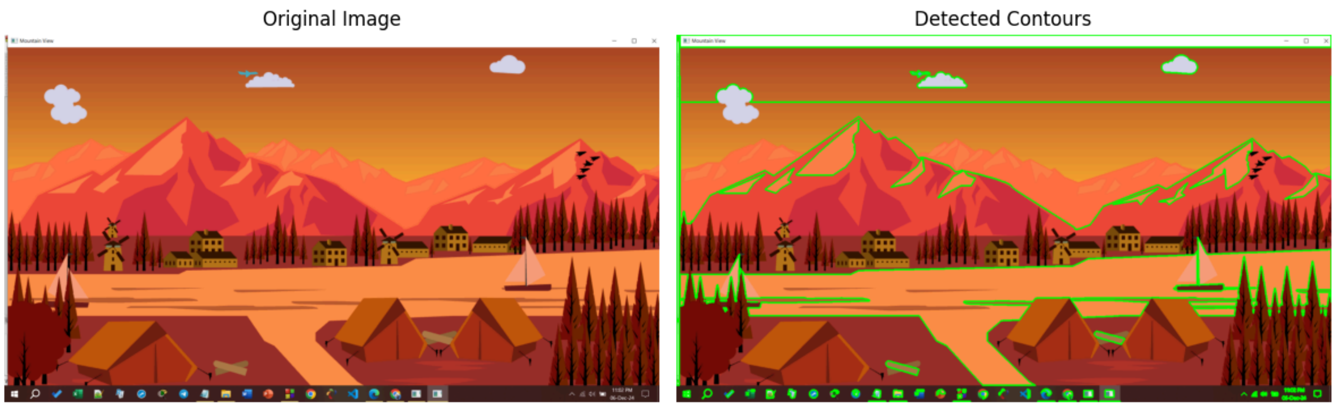
Applications:

Shape analysis: Recognizing or analyzing the shapes of objects in the image.

Object recognition: Matching the extracted contours with predefined shapes.

Medical imaging: Identifying the edges of structures in images, such as tumors or organs.

```
import cv2
import matplotlib.pyplot as plt

def extract_contours(gray_image, image_rgb):
    # Thresholding to prepare the image for contour detection
    _, thresh = cv2.threshold(gray_image, 127, 255, cv2.THRESH_BINARY)
    # Find contours using the binary threshold image
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Draw contours on the original image
    contour_image = cv2.drawContours(image_rgb.copy(), contours, -1, (0, 255, 0), 3)

    # Plot the original and contour-detected images
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    axes[0].imshow(image_rgb)
    axes[0].set_title("Original Image")
    axes[0].axis('off')

    axes[1].imshow(contour_image)
    axes[1].set_title("Detected Contours")
    axes[1].axis('off')

    plt.tight_layout()
    plt.show()

# Run the function
extract_contours(gray_image, image_rgb)
```



Feature 8: Heatmap (Hitmap) Visualization

Technique: Heatmap Visualization

Purpose: A heatmap is a graphical representation of data where individual values are represented by colors, used to visualize areas with high or low activity.

Functionality:

A heatmap is generated by mapping the intensity or importance of different areas to a color scale (e.g., red for high values, blue for low values).

This can be used to highlight areas in an image that have particular significance, like areas of high interest in neural networks or areas with the most activity.

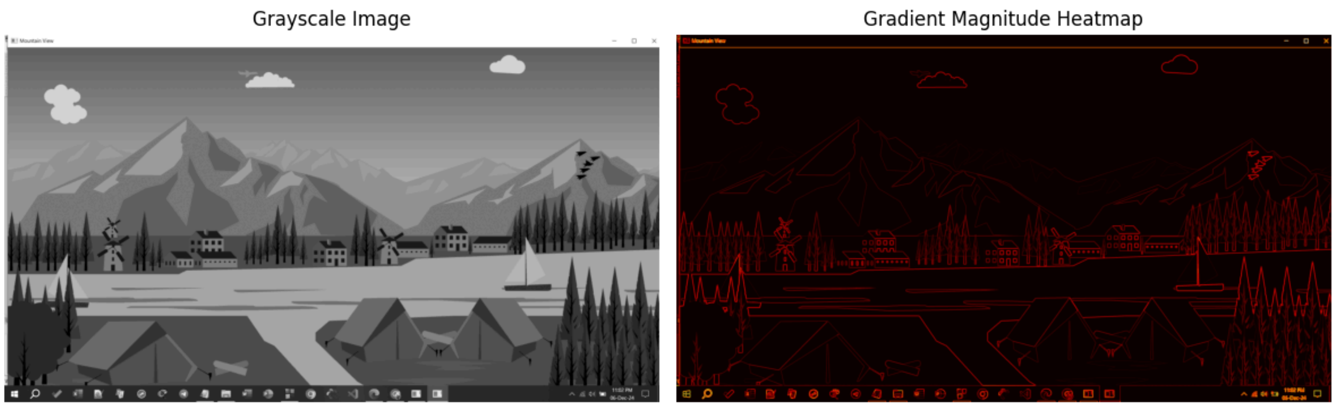
Applications:

Visualizing attention: Used in deep learning models to show which areas of an image are being focused on during classification or detection tasks.

Medical imaging: Highlighting areas of concern, such as tumors or abnormalities.

Surveillance: Displaying regions with the most motion or activity in a video stream.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def generate_heatmap(gray_image):
    # Compute the gradients in the x and y direction using Sobel operator
    grad_x = cv2.Sobel(gray_image, cv2.CV_64F, 1, 0, ksize=3)
    grad_y = cv2.Sobel(gray_image, cv2.CV_64F, 0, 1, ksize=3)

    # Compute the magnitude of the gradients (edge strength)
    grad_mag = cv2.magnitude(grad_x, grad_y)
    # Normalize the gradient magnitude to the range [0, 255]
    grad_mag_normalized = cv2.normalize(grad_mag, None, 0, 255, cv2.NORM_MINMAX)

    # Convert to uint8 for visualization
    grad_mag_normalized = np.uint8(grad_mag_normalized)

    # Plot original grayscale and heatmap
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    axes[0].imshow(gray_image, cmap="gray")
    axes[0].set_title("Grayscale Image")
    axes[0].axis('off')

    axes[1].imshow(grad_mag_normalized, cmap="hot")  # Apply 'hot' colormap for heatmap effect
    axes[1].set_title("Gradient Magnitude Heatmap")
    axes[1].axis('off')

    plt.tight_layout()
    plt.show()

# Run the function
generate_heatmap(gray_image)
```



Feature 9: Color Moments

Technique: Statistical color representation.

Purpose: Describes the color distribution of an image using first (mean), second (variance), and third (skewness) moments.

Functionality: Calculates moments for each channel to form a concise and robust feature vector for color information.

Applications:

Content-based image retrieval.

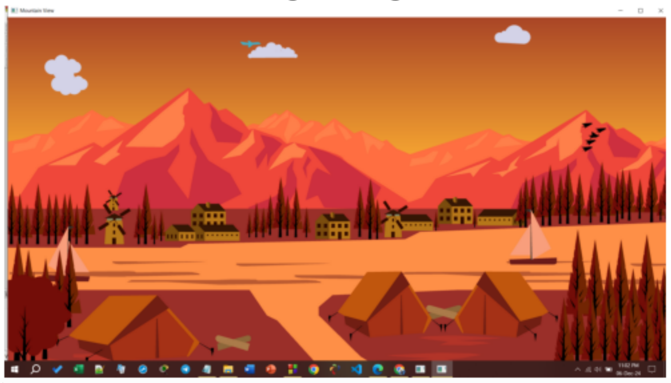
Image classification and clustering.

Compact color description in low-resource environments.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def extract_color_moments(image_rgb):
    # Convert the image to float32 type for accurate moment calculations
    image_float = np.float32(image_rgb)

    # Separate the color channels (Red, Green, Blue)
    channels = cv2.split(image_float)

    # Initialize lists to store moments
    color_moments = []

    for channel in channels:
        # Compute the mean of the channel
        mean = np.mean(channel)

        # Compute the variance of the channel
        variance = np.var(channel)
        # Compute the skewness of the channel
        skewness = np.mean((channel - mean) ** 3) / (np.std(channel) ** 3)

        # Append the moments to the list
        color_moments.extend([mean, variance, skewness])

    # Return the calculated color moments
    return color_moments

def visualize_color_moments(image_rgb):
    # Extract the color moments from the image
    color_moments = extract_color_moments(image_rgb)

    # Display the color moments
    print("Color Moments (Mean, Variance, Skewness) for each channel (R, G, B):")
    print(f"Red Channel - Mean: {color_moments[0]}, Variance: {color_moments[1]}, Skewness: {color_moments[2]}")
    print(f"Green Channel - Mean: {color_moments[3]}, Variance: {color_moments[4]}, Skewness: {color_moments[5]}")
    print(f"Blue Channel - Mean: {color_moments[6]}, Variance: {color_moments[7]}, Skewness: {color_moments[8]}")

    # Plot the image
    plt.imshow(image_rgb)
    plt.title("Original Image")
    plt.axis('off')
    plt.show()

# Run the function to extract and display color moments
visualize_color_moments(image_rgb)
```

```
Color Moments (Mean, Variance, Skewness) for each channel (R, G, B):
Red Channel - Mean: 190.55526733398438, Variance: 3385.656005859375, Skewness: -0.9487600922584534
Green Channel - Mean: 91.37419128417969, Variance: 2853.91845703125, Skewness: 0.8643872141838074
Blue Channel - Mean: 54.19117736816406, Variance: 2344.391357421875, Skewness: 2.9529271125793457
```

Original Image



Feature 10: Wavelet Transform

Technique: Discrete Wavelet Transform (DWT)

Purpose: Captures both spatial and frequency information through multi-resolution analysis.

Functionality: Decomposes the image into approximation and detail coefficients (LL, LH, HL, HH), preserving high- and low-frequency information.

Applications:

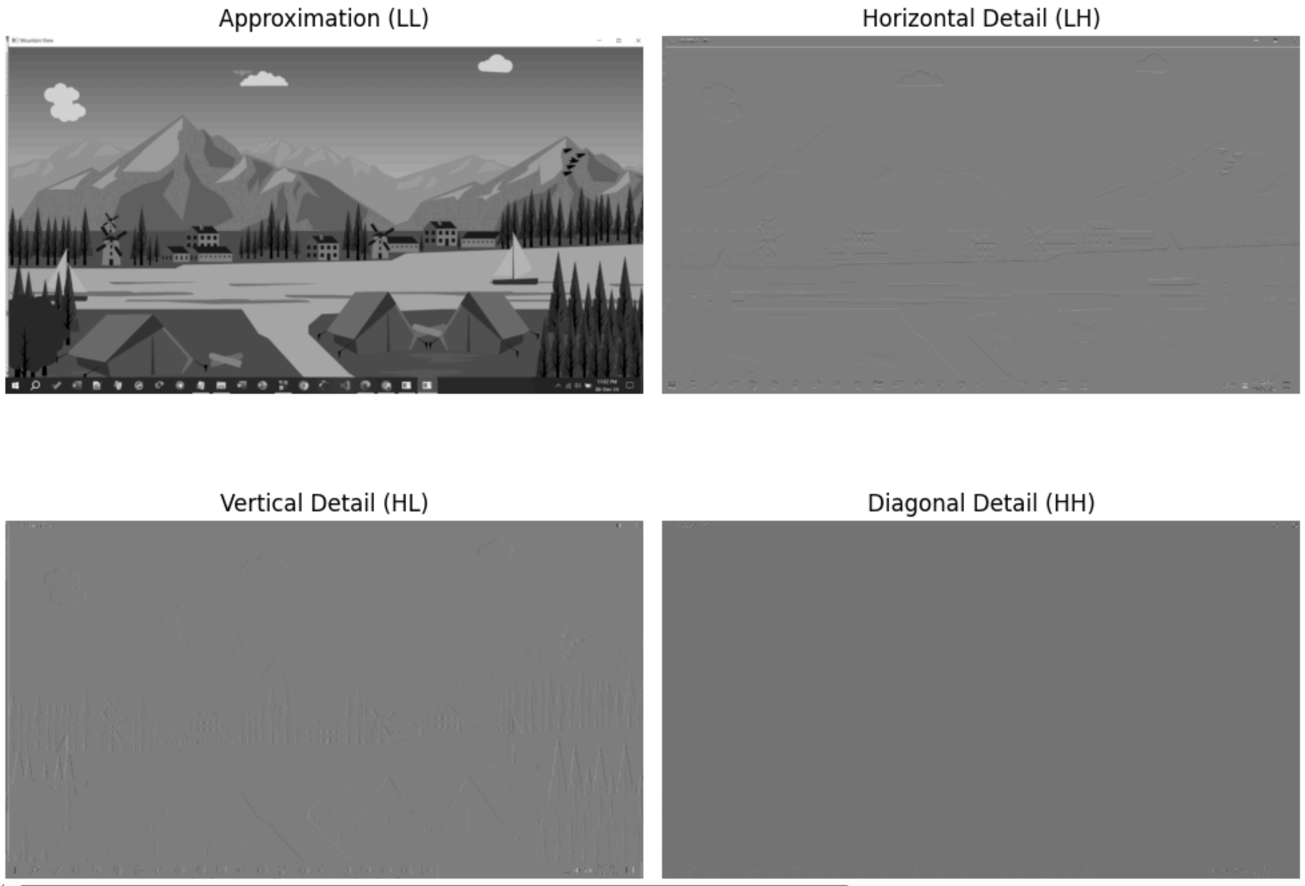
Image compression and denoising.

Texture and edge-based feature extraction.

Fingerprint and face recognition.

```
# Install PyWavelets
!pip install PyWavelets

# Import libraries
import pywt
import cv2
import matplotlib.pyplot as plt

# Function: Wavelet Feature Extraction
def extract_wavelet_features(gray_image):
    # Apply 2D Discrete Wavelet Transform (Haar wavelet)
    coeffs2 = pywt.dwt2(gray_image, 'haar')
    LL, (LH, HL, HH) = coeffs2

    # Display the sub-bands
    fig, axes = plt.subplots(2, 2, figsize=(10, 8))
    axes[0, 0].imshow(LL, cmap='gray')
    axes[0, 0].set_title('Approximation (LL)')
    axes[0, 1].imshow(LH, cmap='gray')
    axes[0, 1].set_title('Horizontal Detail (LH)')
    axes[1, 0].imshow(HL, cmap='gray')
    axes[1, 0].set_title('Vertical Detail (HL)')
    axes[1, 1].imshow(HH, cmap='gray')
    axes[1, 1].set_title('Diagonal Detail (HH)')

    for ax in axes.flatten():
        ax.axis('off')

    plt.tight_layout()
    plt.show()
# Run the function
extract_wavelet_features(gray_image)
```

```
Requirement already satisfied: PyWavelets in /usr/local/lib/python3.11/dist-packages (1.8.0)
Requirement already satisfied: numpy<3,>=1.23 in /usr/local/lib/python3.11/dist-packages (from PyWavelets) (2.0.2)
```



Double-click (or enter) to edit