Machine Problem 4 - Feature Extraction and Image Matching in Computer Vision

---

Overview:

In this machine problem, you will implement various feature extraction and matching algorithms to process, analyze, and compare different images. You will utilize techniques such as SIFT, SURF, ORB, HOG, and Harris Corner Detection to extract keypoints and descriptors. You will also perform feature matching between pairs of images using the FLANN and Brute-Force matchers. Finally, you will explore image segmentation using the Watershed algorithm.

Objectives:

1. To apply different feature extraction methods (SIFT, SURF, ORB, HOG, Harris Corner Detection).
2. To perform feature matching using Brute-Force and FLANN matchers.
3. To implement the Watershed algorithm for image segmentation.
4. To visualize and analyze keypoints and matches between images.
5. To evaluate the performance of different feature extraction methods on different images.

Problem Statement:

1. You are tasked with building a Python program using OpenCV and related libraries to accomplish the
2. following tasks. Each task should be implemented in a separate function, with appropriate comments and
3. visual outputs. You will work with images provided in your directory or chosen from any online source.

Important libraries and repo

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow
from google.colab import files
```

```
from PIL import Image
```

Cloning OpenCV repo to enable SURF feature extraction

```
# Remove any existing OpenCV installations
!pip uninstall -y opencv-python opencv-python-headless opencv-contrib-python

# Install required dependencies
!apt update && apt install -y python3-opencv build-essential cmake git libgtk2.0-dev pkg-config libavcodec-dev libavformat-dev libswscale-dev

# Clone the OpenCV and OpenCV Contrib repositories
!rm -rf opencv opencv_contrib  # Clear any previous installations
!git clone https://github.com/opencv/opencv.git
!git clone https://github.com/opencv/opencv_contrib.git

# Create a build directory and navigate to it
!mkdir -p opencv/build
%cd opencv/build

# Run CMake configuration with OPENCV_ENABLE_NONFREE enabled to include SURF
!cmake -D CMAKE_BUILD_TYPE=Release \
       -D CMAKE_INSTALL_PREFIX=/usr/local \
       -D OPENCV_ENABLE_NONFREE=ON \
       -D OPENCV_EXTRA_MODULES_PATH=../../opencv_contrib/modules \
       -D BUILD_EXAMPLES=OFF ..

# Compile OpenCV (this will take some time)
!make -j8

# Install the compiled OpenCV library
!make install
!ldconfig
```

To check if SURF is Available:

```
import cv2

try:
   # Attempt to create a SURF detector
```

```
    surf = cv2.xfeatures2d.SURF_create()
    print("SURF is available.")
except AttributeError:
    print("SURF is not available.")
```

Output:

```
SURF is available.
```

Explanation:

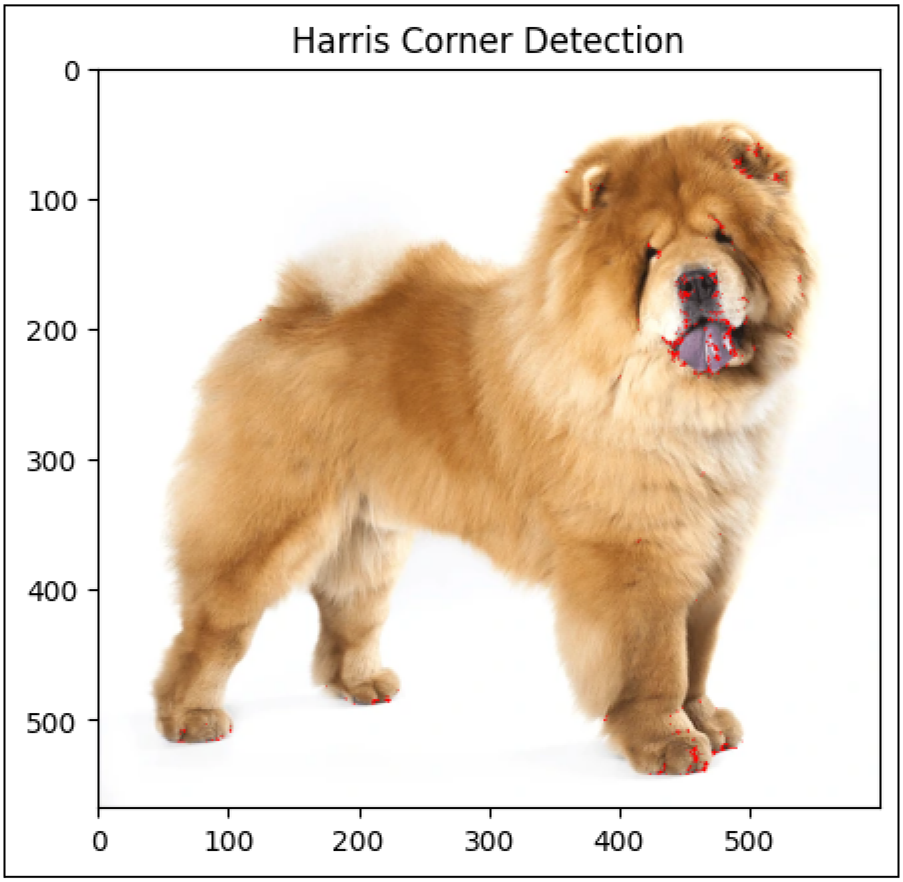
This code installs all necessary dependencies, clones the OpenCV repositories, configures the build, compiles the library, and installs it on the system.

Task 1: Harris Corner Detection

Code:

```
# Harris Corner Detection
def harris_corner_detection(image_path):
  # Load the image
    img = cv2.imread('/content/dog1.png')
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Harris Corner Detection
    gray = np.float32(gray_img)
    corners = cv2.cornerHarris(gray, blockSize=2, ksize=3, k=0.04)

    # Mark corners on the original image
    img[corners > 0.01 * corners.max()] = [0, 0, 255]

    # Display the result
    plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
    plt.title('Harris Corner Detection')
    plt.show()
harris_corner_detection('/content/dog1.png')
```

Output:



Explanation:

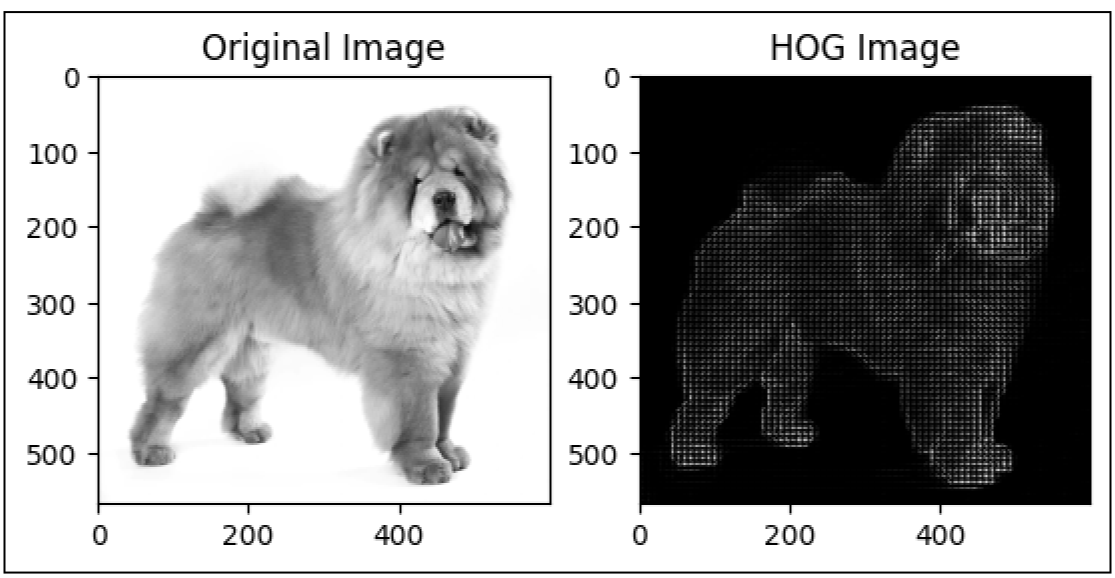
This code performs **Harris Corner Detection** on an image to identify corners or points of interest. It starts by loading an image, converting it to grayscale, and then applying the Harris Corner Detection algorithm. The grayscale image is converted to a float32 type, which is necessary for accurate corner detection. The cornerHarris function takes parameters that define the sensitivity of corner detection. Once corners are identified, they are marked in red by setting pixel values to [0, 0, 255] where the corners' response exceeds a certain threshold. Finally, the modified image is displayed, showing red dots at the detected corners.

Task 2: HOG Feature Extraction

Code:

```
# HOG Feature Extraction
def hog_feature_extraction(image_path):
  # Extract HOG feature
 img = cv2.imread('/content/dog1.png')
 gray_image = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
   hog_feature,  hog_image  =  hog(gray_image,  pixels_per_cell=(8,  8),
cells_per_block=(2, 2),
                                 visualize=True, feature_vector=True)

  hog_image_rescale  =  exposure.rescale_intensity(hog_image,  in_range=(0,
10))

 # Display the original the HOG images
 plt.subplot(1, 2, 1)
 plt.imshow(gray_image, cmap='gray')
 plt.title('Original Image')

 plt.subplot(1, 2, 2)
 plt.imshow(hog_image_rescale, cmap='gray')
 plt.title('HOG Image')
 plt.show()
hog_feature_extraction('/content/dog1.png')
```

Output:



Explanation:

The process starts by loading an image and converting it to grayscale. The hog function then calculates the HOG features, focusing on gradients in the image, which highlight shapes and edges. It specifies parameters for the size of cells and blocks to structure how gradients are calculated. The visualize=True option produces a visual representation of the HOG features, and feature_vector=True returns a 1D feature array. To enhance the HOG image for display, the intensity is rescaled using exposure.rescale_intensity. Finally, the original and the HOG images are displayed side-by-side for comparison.

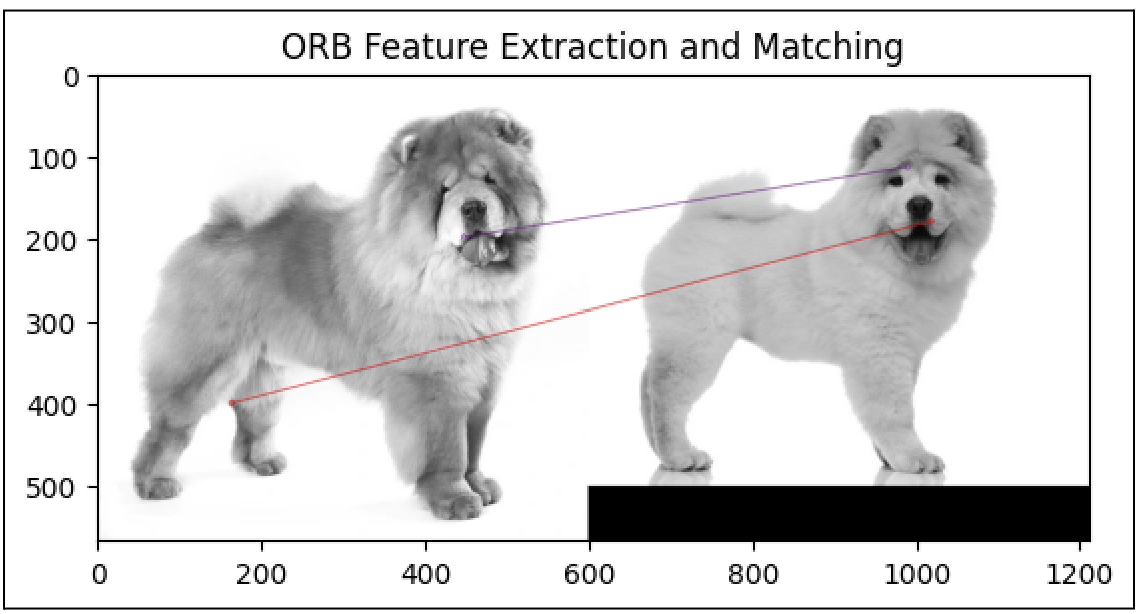
Task 3: ORB Feature Extraction and Matching

Code:

```
def orb_feature_matching(image_path1, image_path2):
   # Load the images in grayscale
   img1 = cv2.imread(image_path1, cv2.IMREAD_GRAYSCALE)
   img2 = cv2.imread(image_path2, cv2.IMREAD_GRAYSCALE)

   # ORB detector
   orb = cv2.ORB_create()

   # Find keypoints and descriptors with ORB
   kp1, des1 = orb.detectAndCompute(img1, None)
   kp2, des2 = orb.detectAndCompute(img2, None)

   # FLANN-based matcher setup
   FLANN_INDEX_LSH = 6
        index_params   =   dict(algorithm=FLANN_INDEX_LSH,   table_number=6,
key_size=12, multi_probe_level=1)
   search_params = dict(checks=50)
   flann = cv2.FlannBasedMatcher(index_params, search_params)

   # Match descriptors using KNN
   matches = flann.knnMatch(des1, des2, k=2)

   good_matches = []
```

```
    for match in matches:
        if len(match) == 2:  # Ensure there are two matches (m, n)
            m, n = match
            if m.distance < 0.7 * n.distance:
                good_matches.append(m)

    # Check if there are any good matches
    if len(good_matches) > 0:
        # Draw matching keypoints
        matched_image = cv2.drawMatches(img1, kp1, img2, kp2, good_matches,
None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
        # Display the matched image
        plt.imshow(matched_image)
        plt.title('ORB Feature Extraction and Matching')
        plt.show()
    else:
        print("No good matches found between the images.")

# Run the function
orb_feature_matching('/content/dog1.png', '/content/dog2.png')
```

Output:



Explanation:

The **ORB (Oriented FAST and Rotated BRIEF)** feature detection and matching between two images, identifying similar patterns. It begins by loading two images in grayscale. Using ORB, it detects keypoints (unique points of interest) and computes descriptors (feature representations) for each image. The FLANN (Fast Library for Approximate Nearest Neighbors) matcher, specifically set up for binary descriptors, finds matches between the two sets of descriptors by applying K-Nearest Neighbors (KNN) matching. A filtering step selects "good matches" based on distance: if the closest match is significantly closer than the second closest, it's considered a good match. Finally, if any good matches are found, they are drawn on a combined image and displayed. Otherwise, a message is printed indicating no good matches were found.

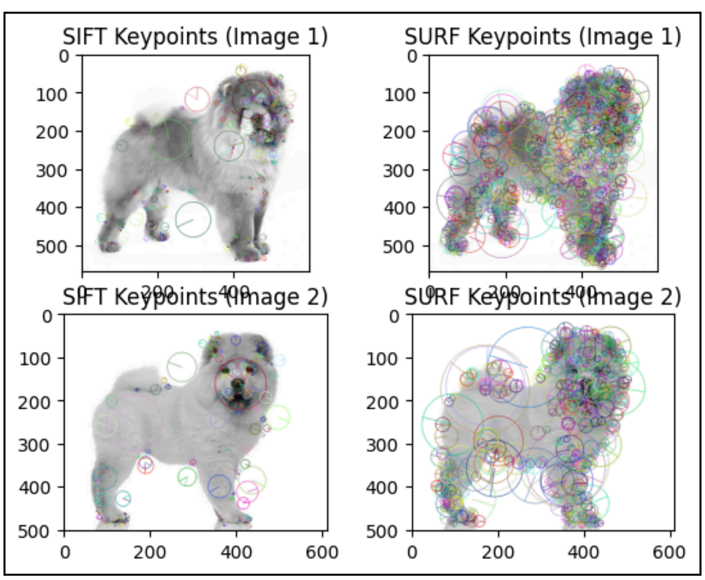
Task 4: SIFT and SURF Feature Extraction

Code:

```
#Task 4: SIFT and SURF Feature Extraction

def sift_and_surf_feature_extraction(image_path1, image_path2):
   # Load the images in grayscale
   img1 = cv2.imread(image_path1, cv2.IMREAD_GRAYSCALE)
   img2 = cv2.imread(image_path2, cv2.IMREAD_GRAYSCALE)

   # SIFT detector
   sift = cv2.SIFT_create()
   kp1, des1 = sift.detectAndCompute(img1, None)
   kp2, des2 = sift.detectAndCompute(img2, None)

   # SURF detector (ensure you have opencv-contrib-python installed)
   surf = cv2.xfeatures2d.SURF_create()
   kp1_surf, des1_surf = surf.detectAndCompute(img1, None)
   kp2_surf, des2_surf = surf.detectAndCompute(img2, None)

   # Draw keypoints for both SIFT and SURF
   img1_sift_kp = cv2.drawKeypoints(img1, kp1, None, 
flags=cv2.DrawMatchesFlags_DRAW_RICH_KEYPOINTS)
   img1_surf_kp = cv2.drawKeypoints(img1, kp1_surf, None, 
flags=cv2.DrawMatchesFlags_DRAW_RICH_KEYPOINTS)
```

```
            img2_sift_kp    =    cv2.drawKeypoints(img2,    kp2,    None,
flags=cv2.DrawMatchesFlags_DRAW_RICH_KEYPOINTS)
          img2_surf_kp   =   cv2.drawKeypoints(img2,   kp2_surf,   None,
flags=cv2.DrawMatchesFlags_DRAW_RICH_KEYPOINTS)

   # Display SIFT and SURF keypoints for both images
   plt.subplot(2, 2, 1)
   plt.imshow(img1_sift_kp, cmap='gray')
   plt.title('SIFT Keypoints (Image 1)')

   plt.subplot(2, 2, 2)
   plt.imshow(img1_surf_kp, cmap='gray')
   plt.title('SURF Keypoints (Image 1)')

   plt.subplot(2, 2, 3)
   plt.imshow(img2_sift_kp, cmap='gray')
   plt.title('SIFT Keypoints (Image 2)')

   plt.subplot(2, 2, 4)
   plt.imshow(img2_surf_kp, cmap='gray')
   plt.title('SURF Keypoints (Image 2)')

   plt.show()

# Run the function
sift_and_surf_feature_extraction('/content/dog1.png', '/content/dog2.png')
```

Output:



Explanation:

First, it applies SIFT to find keypoints and descriptors, capturing details about important points in the images that are scale and rotation-invariant. Then, it uses SURF, which is similar to SIFT but faster, to detect keypoints and descriptors as well.

The detected keypoints from each method are then visualized separately for both images. The drawKeypoints function highlights each keypoint location, showing the distinctive areas identified by SIFT and SURF. Finally, four images—SIFT and SURF keypoints for both the first and second images—are displayed side-by-side in a 2x2 grid, allowing for easy comparison of keypoints found by each method.

Task 5: Feature Matching using Brute-Force Matcher

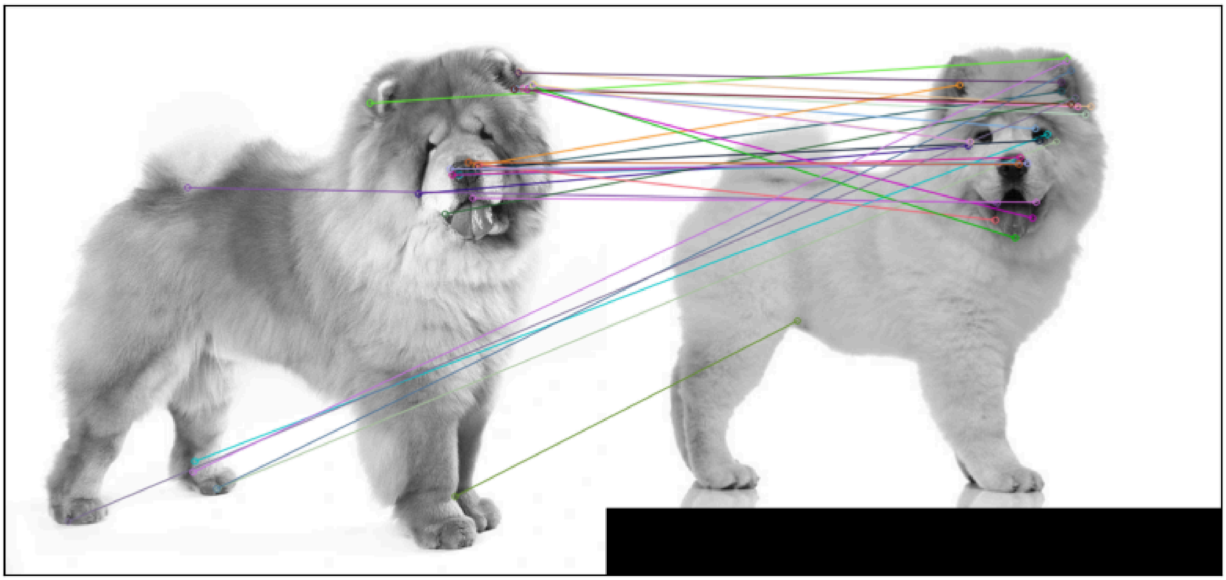
Code:

```
#Task 5: Feature Matching using Brute-Force Matcher

def brute_force_feature_matching(image_path1, image_path2):
    # Initialize ORB detector
    orb = cv2.ORB_create()
    img1 = cv2.imread(image_path1, cv2.IMREAD_GRAYSCALE)
    img2 = cv2.imread(image_path2, cv2.IMREAD_GRAYSCALE)

    # Find the keypoints and descriptors with ORB
    kp1, des1 = orb.detectAndCompute(img1, None)
    kp2, des2 = orb.detectAndCompute(img2, None)
    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

    # Match descriptors
    matches = bf.match(des1, des2)
    matches = sorted(matches, key=lambda x: x.distance)
     matched_image = cv2.drawMatches(img1, kp1, img2, kp2, matches[:30],
None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

    # Display the matched keypoints
    cv2_imshow(matched_image)
```

```python
brute_force_feature_matching('/content/dog1.png', '/content/dog2.png')
```

Output:

Brute-Force Feature Matching



Explanation:

It starts by loading two grayscale images and detecting their keypoints and descriptors using ORB. The Brute-Force Matcher is initialized with cv2.NORM_HAMMING, which is suitable for binary descriptors like those from ORB, and crossCheck=True to ensure bidirectional matching consistency.

Descriptors from both images are then matched, and the matches are sorted based on distance (the lower the distance, the better the match). The code takes the top 30 best matches and draws them on a combined image of both inputs. Finally, the cv2_imshow function is used to display the matched keypoints, showing the corresponding points across the images.

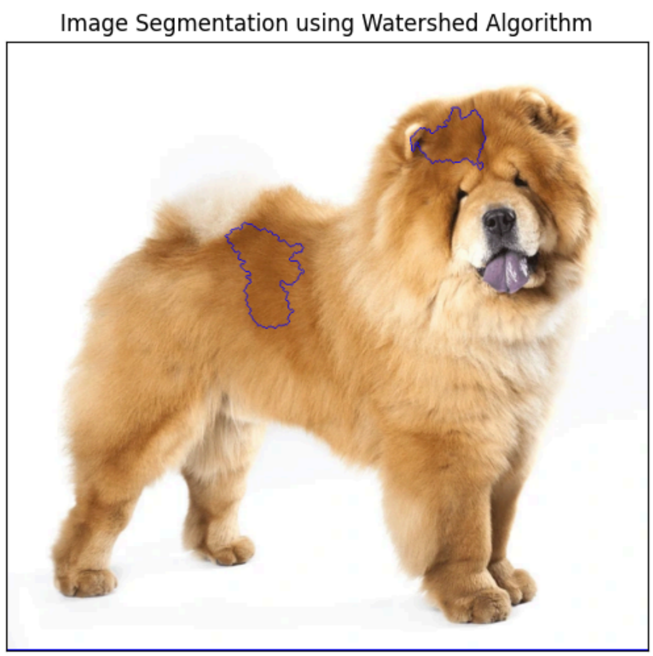
Task 6: Image Segmentation using Watershed Algorithm

Code:

```
#Task 6: Image Segmentation using Watershed Algorithm

def watershed_segmentation(image_path):
    # Load the image
    img = cv2.imread(image_path)

    # Convert the image to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    _, thresh = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY_INV)

    # Noise removal using morphological operations
    kernel = np.ones((3, 3), np.uint8)
    opening = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel, iterations=2)
    sure_bg = cv2.dilate(opening, kernel, iterations=3)

    # Finding sure foreground area
    dist_transform = cv2.distanceTransform(opening, cv2.DIST_L2, 5)
    _, sure_fg = cv2.threshold(dist_transform, 0.7 * dist_transform.max(), 255, 0)

    # Identifying unknown regions
    unknown = cv2.subtract(sure_bg, np.uint8(sure_fg))

    # Marker labelling
    _, markers = cv2.connectedComponents(np.uint8(sure_fg))
    markers = markers + 1
    markers[unknown == 255] = 0

    # Apply the Watershed algorithm
    markers = cv2.watershed(img, markers)
    img[markers == -1] = [255, 0, 0]  # Mark boundaries in red

    # Convert BGR to RGB for matplotlib
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    # Display the segmented image with matplotlib
    plt.figure(figsize=(8, 6))
    plt.imshow(img_rgb)
    plt.title('Image Segmentation using Watershed Algorithm')
```

```
    plt.xticks([])
    plt.yticks([])
    plt.show()

# Example usage
watershed_segmentation('/content/dog1.png')
```

Output:



Explanation:

It converts the image to grayscale, applies binary thresholding, and removes noise with morphological operations. The sure background and foreground areas are identified, and unknown regions are marked. Markers are assigned to the foreground, and the Watershed algorithm detects object boundaries, marking them in red. The final segmented image is displayed.

- *End of Documentation -*