Seventh report - Texture identification

Kimberly Grace Sevillano C.

***Subject*— Implementing texture indentification algorithm.**

I. Objective

The objective of this report is to present the results of a texture identification challenge using Discrete Fourier Transform (DFT) and autocorrelation. The report describes the problem, methods used, methodology, results obtained, and conclusions drawn from the implementation.

A. Problem Description

Texture identification is an essential task in image processing and computer vision. It involves characterizing the texture of an image, which can be used for various applications such as image segmentation, object recognition, and feature extraction. The effectiveness of a texture identification technique depends on its ability to discriminate between different textures and its robustness to variations in scale, orientation, and illumination.

II. Methods

There are several methods for texture identification in images. These include:

A. Statistical Methods

Statistical methods analyze the distribution of pixel intensities or their relationships to characterize texture. Common statistical methods include:

- Gray-Level Co-occurrence Matrix (GLCM)
- Local Binary Patterns (LBP)
- Histogram of Oriented Gradients (HOG)

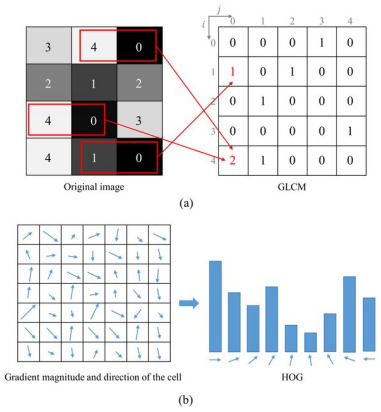


Fig. 1. Feature extraction methods in image processing, GLCM and HOG. (a) An example of calculating the GLCM from the original image. (b) A process of determining the gradient based histogram from the cell within the image.

B. Model-Based Methods

Model-based methods represent textures using mathematical models, such as:

- Markov Random Fields (MRF)
- Fractal Models
- Gaussian Processes

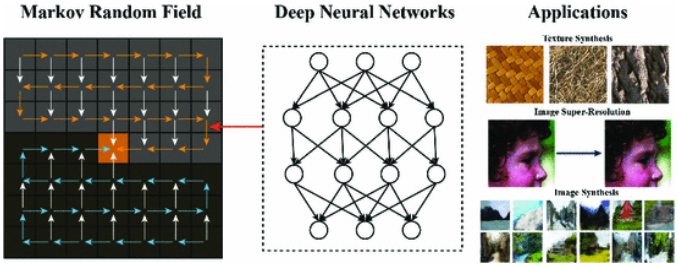


Fig. 2. An example of calculating the MRF from the original image.

C. Transform-Based Methods

Transform-based methods analyze the frequency content of textures by applying transformations, such as:

- Fourier Transform
- Wavelet Transform
- Gabor Filters

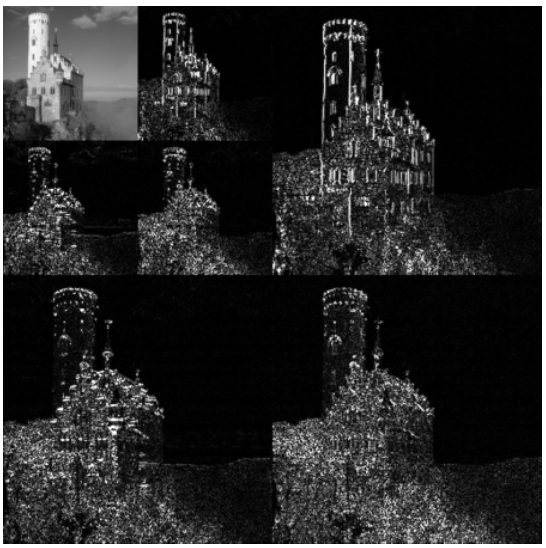


Fig. 3. An example of the 2D discrete wavelet transform.

III. Methodology

The methodology for the texture identification challenge is as follows:

1) Load the images from the given directory and convert them to grayscale.

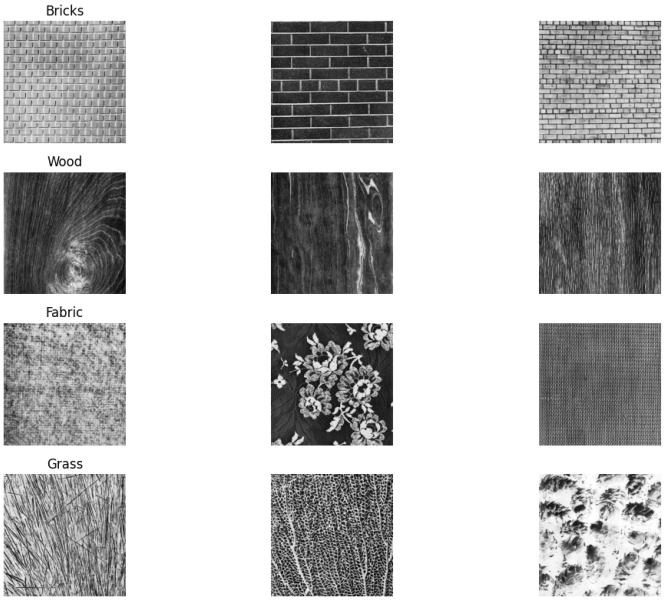


Fig. 4. This figure displays sample images from the Brodatz dataset, categorized into Bricks, Wood, Fabric, and Grass, illustrating the diversity of natural textures for analysis.

2) Calculate the attributes (m1, m2) for each image using DFT and autocorrelation.
3) Compute the mean and standard deviation for each texture family.

```
def extract_features(images):
    features = {}
    for file_name, image in images.items():
        dft = compute_dft(image)
        autocorr = compute_autocorrelation(
    image)
        m1, m2 = np.mean(dft), np.mean(
    autocorr)
        features[file_name] = (m1, m2)
    return features
```

Listing 1. Function for extract_features in python

4) Represent each texture family by an ellipse using the calculated mean and standard deviation.

```
def compute_family_stats(features, families):
    family_stats = {}
    for family, file_names in families:
        family_features = [features[file_name]
        for file_name in file_names]
        mean = np.mean(family_features,
        axis=0)
        std_dev = np.std(family_features,
        axis=0)
        family_stats[family] = (mean, std_dev)
    return family_stats
```

Listing 2. This function computes the mean and standard deviation of feature vectors for each texture family in the given dataset in python

5) Visualize the ellipses and images in the feature space.

```
family_colors = {'Bricks': 'red',
                 'Wood': 'green',
                 'Fabric': 'blue',
                 'Grass': 'yellow'}
def create_ellipses(family_stats,
    family_colors):
    ellipses = []
    for family, (mean, std_dev) in
    family_stats.items():
        color = family_colors.get(family, '
    black')
        ellipse = Ellipse(xy=mean, width=
    std_dev[0] * 2, height=std_dev[1] * 2,
    alpha=0.5, color=color)
        ellipses.append((family, ellipse))
    return ellipses
```

Listing 3. This function creates ellipses for each texture family using their mean and standard deviation in python

IV. Results

The texture identification method was effectively implemented, and the ellipses representing each texture family were visualized in the feature space. The majority of images were accurately classified into their respective families, demonstrating the method's ability to identify different textures based on their spatial patterns and frequency content. However, there were a few instances where images did not fall within the boundary of any ellipse. This discrepancy could be attributed to factors such as insufficiently discriminative features, the presence of noise, variations in the texture images, or inaccurate assumptions about the texture distributions.

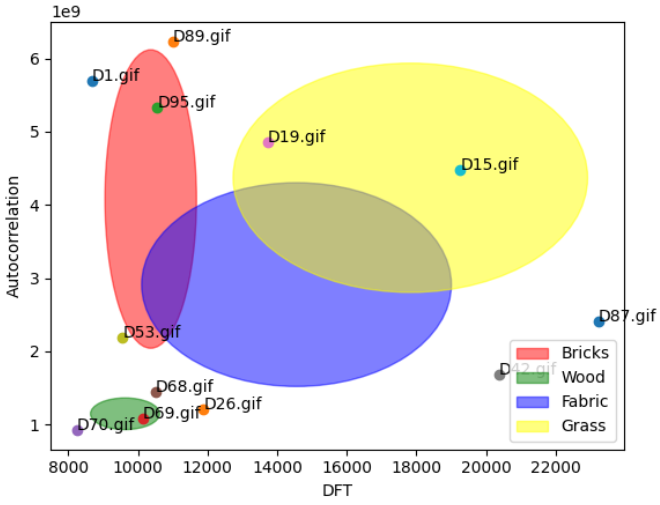


Fig. 5. Texture identification results.

To address these limitations, future work could investigate alternative or additional features that better capture the unique characteristics of each texture type. Moreover, refining the preprocessing steps and employing robust feature extraction techniques could help mitigate the impact of noise and variations. Increasing the sample size and exploring non-parametric methods for modeling the distribution of textures

might also improve the overall performance of the texture identification method.

V. CONCLUSIONS

The approach presented in this study, which combines Discrete Fourier Transform and autocorrelation, demonstrates its effectiveness in distinguishing various textures in images. This method serves as a promising starting point for more advanced texture analysis techniques in image processing and computer vision applications.

To further improve the accuracy of texture identification, it is recommended to explore the integration of additional features that capture more intricate properties of textures. Furthermore, incorporating machine learning techniques, such as deep learning-based models, could potentially enhance the overall performance of the method. As a result, this would enable more sophisticated and precise texture analysis, catering to a broader range of applications and challenges in the field of computer vision.

REFERENCES

[1] Gonzalez, R. C., and Woods, R. E. (2008). Digital Image Processing (3rd ed.). Pearson.

[2] Haralick, R. M., Shanmugam, K., and Dinstein, I. (1973). Textural Features for Image Classification. IEEE Transactions on Systems, Man, and Cybernetics, SMC-3(6), 610-621.

[3] Mallat, S. (2008). A Wavelet Tour of Signal Processing: The Sparse Way (3rd ed.). Academic Press.

Seventh Report (7th)

Mar 7, 2023

[1]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from scipy.signal import fftconvolve
from scipy.spatial.distance import mahalanobis
from matplotlib.patches import Ellipse
import imageio.v2 as imageio
```

1. Define functions to compute DFT and autocorrelation features

[2]:
```
def compute_dft(image):
    return np.abs(np.fft.fft2(image))

def compute_autocorrelation(image):
    autocorr = fftconvolve(image, image[::-1, ::-1], mode='full')
    return autocorr[autocorr.shape[0] // 2, autocorr.shape[1] // 2:]
```

2. Load the Brodatz database of textures :

[3]:
```
def load_images(folder_path, families):
    images = {}
    for family, file_names in families:
        for file_name in file_names:
            image_path = os.path.join(folder_path, file_name)
            image = Image.open(image_path).convert('L')
            images[file_name] = np.array(image)
    return images
```

[4]:
```
folder_path = r"C:\Users\ksevi\OneDrive\Desktop\MASTER\SCENE_LALIGANT\septimo␣
 ↪report\Original Brodatz"

families = [('Bricks', ['D1.gif', 'D26.gif', 'D95.gif']),
            ('Wood', ['D69.gif', 'D70.gif', 'D68.gif']),
            ('Fabric', ['D19.gif', 'D42.gif', 'D53.gif']),
            ('Grass', ['D15.gif', 'D87.gif', 'D89.gif'])]

# Load images from the given directory and families list
```

```
images = load_images(folder_path, families)
```

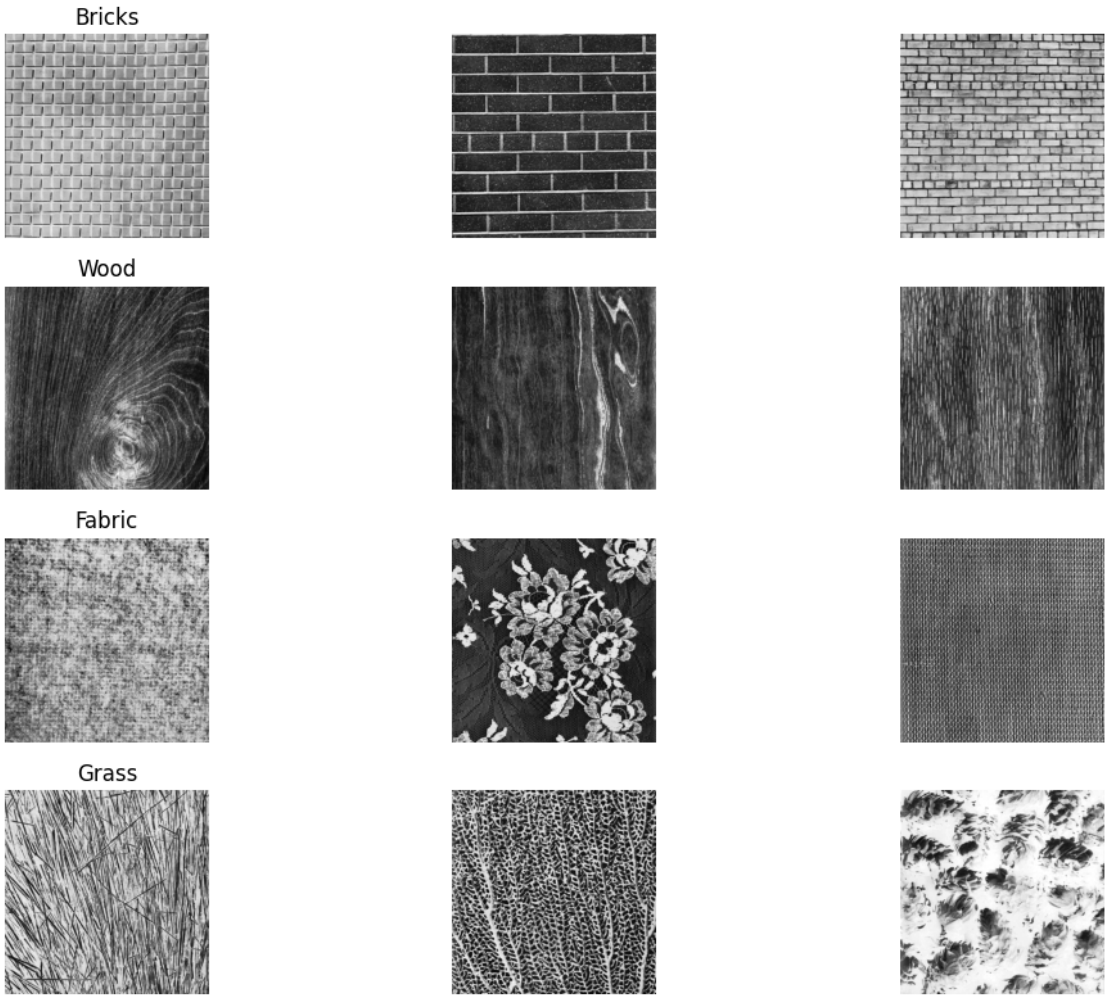
[5]:
```
def load_images(folder_path, families):
    images = {}
    for family, file_names in families:
        images[family] = [imageio.imread(os.path.join(folder_path, file_name))␣
 ↪for file_name in file_names]
    return images
```

[6]:
```
#plotting dataset
def load_images_datas(folder_path, families):
    images = {}
    for family, file_names in families:
        images[family] = [imageio.imread(os.path.join(folder_path, file_name))␣
 ↪for file_name in file_names]
    return images
def plot_images(images):
    fig, axes = plt.subplots(len(families), len(families[0][1]), figsize=(12, 8))

    for i, (family, imgs) in enumerate(images.items()):
        for j, img in enumerate(imgs):
            axes[i, j].imshow(img, cmap='gray')
            axes[i, j].axis('off')
            if j == 0:
                axes[i, j].set_title(family)

    plt.tight_layout()
    plt.show()

images_dataset = load_images_datas(folder_path, families)
plot_images(images_dataset)
```



3. Define a function to compute the distance between a test image and a texture family:

[7]:
```
def extract_features(images):
    features = {}
    for file_name, image in images.items():
        dft = compute_dft(image)
        autocorr = compute_autocorrelation(image)
        m1, m2 = np.mean(dft), np.mean(autocorr)
        features[file_name] = (m1, m2)
    return features
```

4. Load the texture families, compute the features, and classify a new image:

[8]:
```
# Calculate the attributes (m1, m2) for each image using DFT and Autocorrelation
features = extract_features(images)
```

[9]:
```
def compute_family_stats(features, families):
    family_stats = {}
    for family, file_names in families:
        family_features = [features[file_name] for file_name in file_names]
        mean = np.mean(family_features, axis=0)
        std_dev = np.std(family_features, axis=0)
        family_stats[family] = (mean, std_dev)
    return family_stats
```

[10]:
```
# Compute the mean and standard deviation for each family
family_stats = compute_family_stats(features, families)
```

[11]:
```
family_colors = {
                'Bricks': 'red',
                'Wood': 'green',
                'Fabric': 'blue',
                'Grass': 'yellow'}

def create_ellipses(family_stats, family_colors):
    ellipses = []
    for family, (mean, std_dev) in family_stats.items():
        color = family_colors.get(family, 'black')
        ellipse = Ellipse(xy=mean, width=std_dev[0] * 2, height=std_dev[1] * 2,␣
 ↪alpha=0.5, color=color)
        ellipses.append((family, ellipse))
    return ellipses
```

[12]:
```
# Represent each family by an ellipse
ellipses = create_ellipses(family_stats, family_colors)
```

[13]:
```
def plot_results(ellipses, features, images, families):
    fig, ax = plt.subplots()

    for family, ellipse in ellipses:
        ax.add_artist(ellipse)
        ellipse.set_clip_box(ax.bbox)
        ellipse.set_label(family)

    for family, file_names in families:
        for file_name in file_names:
            plt.scatter(*features[file_name], marker='o')
            plt.annotate(file_name, (features[file_name][0],␣
 ↪features[file_name][1]))

    plt.legend()
    plt.xlabel('DFT')
    plt.ylabel('Autocorrelation')
```

```
    plt.show()
```

[14]:
```
# Visualize the ellipses and images
plot_results(ellipses, features, images, families)
```