**Computer Vision Mini Project**

**Texture Recognition**

**Shruti Nashine BT15CSE055**

**Ankita Gonnade BT15CSE028**

**LBP**

LBPs compute a local representation of texture. This local representation is constructed by comparing each pixel with its surrounding neighborhood of pixels.

1. The first step in constructing the LBP texture descriptor is to convert the image to grayscale.
2. For each pixel in the grayscale image, we select a neighborhood of size r surrounding the center pixel. A LBP value is then calculated for this center pixel and stored in the output 2D array with the same width and height as the input image.
3. We then convert this binary string to decimal, yielding a corresponding value.
4. This process of thresholding, accumulating binary strings, and storing the output decimal value in the LBP array is then repeated for each pixel in the input image.
5. The last step is to compute a histogram over the output LBP array.

A primary benefit of this original LBP implementation is that we can capture extremely fine-grained details in the image.

To account for variable neighborhood sizes, two parameters were introduced:

1. The number of points p in a circularly symmetric neighborhood to consider (thus removing relying on a square neighborhood).
2. The radius of the circle r, which allows us to account for different scales.

A LBP is considered to be uniform if it has at most two 0-1 or 1-0 transitions. For example, the pattern 00001000 (2 transitions) and 10000000 (1 transition) are both considered to be **uniform patterns** since they contain at most two 0-1 and 1-0 transitions. The pattern 01010010 ) on the other hand is not considered a uniform pattern since it has six 0-1 or 1-0 transitions.

Uniform LBP patterns are interesting because they add an extra level of rotation and grayscale invariance, hence they are commonly used when extracting LBP feature vectors from images.

**Implementation**

We use the scikit-image implementation of LBPs as they offer more control of the types of LBP histograms you want to generate. Furthermore, the scikit-image implementation also includes variants of LBPs that improve rotation and grayscale invariance.

1. We first need to create a dataset of textures.
2. Given this dataset of **area rug**, **carpet**, **keyboard**, and **wrapping paper**, our goal is to extract Local Binary Patterns from these images and apply machine learning to automatically recognize and categorize these texture images.
3. We know that LBPs require two parameters: the radius of the pattern surrounding the central pixel, along with the number of points along the outer radius, so we initialize these values.
4. The lbp variable returned by the local\_binary\_patterns is a 2D array with the same width and height as our input image — each of the values inside lbp ranges from [0, numPoints + 2], a value for each of the possible numPoints + 1 possible rotation invariant prototypes (see the discussion of uniform patterns at the top of this post for more information) along with an extra dimension for all patterns that are not uniform, yielding a total of numPoints + 2 unique possible values.
5. Thus, to construct the actual feature vector, we need to make a call to np.histogram which counts the number of times each of the LBP prototypes appears. The returned histogram is numPoints + 2-dimensional, an integer count for each of the prototypes. We then take this histogram and normalize it such that it sums to 1, and then return it to the calling function.
6. In order to store the LBP feature vectors and the label names associated with each of the texture classes, we’ll initialize two lists: data to store the feature vectors and labels to store the names of each texture
7. We start looping over our training images. For each of these images, we load them from disk, convert them to grayscale, and extract Local Binary Pattern features. The label (i.e., texture name) is then extracted from the image path and both our labels and data lists are updated, respectively.
8. Once we have our features and labels extracted, we can train our Linear Support Vector Machine to learn the difference between the various texture classes.
9. Once our Linear SVM is trained, we can use it to classify subsequent texture images.
10. Again, all we need to do is load our image from disk, convert it to grayscale, extract Local Binary Patterns from the grayscale image, and then pass the features onto our Linear SVM for classification.

**SVM**

A SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.

Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other.