Features and How to Extract Them in MatLab

The final task for our image processing system will be to take an object region in an image and classify it (thereby "recognizing" it). In other words, we must generate a collection of *classes*, such as 'bird', 'boat', 'bear', etc., and then be able to take a region in an image and determine which, if any, of the classes that region falls into. Such a mechanism is called a *classifier*.

To feed the classifier information to use for classification, we need to extract mathematical measurements (*features*) from that object. There are two categories of features: *shape* (features of the shape's geometry captured by both the boundary and the interior region) and *texture* (features of the grayscale values of the interior). Ideally, classification uses the fewest features possible to adequately separate the classes. At the same time, the more features used, the more information the classifier has to work with.

The features selected for classification should be stored in a feature vector. If A is the image, and the selected features are $\mu(\text{image})$ = (average gray-values) and $n(\text{image})$ = (number of pixels), then the associated feature vector will be $\mathbf{v}(A) = (\mu(A), n(A))$. (Note that this is a stupid collection of features if you actually want to learn anything about your images.)

Below is a list of features, what they measure, and how to extract them in MatLab.

1. Shape

1.1. The `regionprops` command.

The `regionprops` command is your new best friend, because it will extract many of the relevant shape features for you without too much trouble. Check out the Help page for this command. Starting with a binary image `bw` where white regions represent objects of interest, first form a labeled version `L` of that image, then feed `L` to the `regionprops` function:

`>> L = bwlabel(bw, n)`, where `n` is the number of objects to be labeled;
`>> feature = regionprops(L, 'property');`.

A few of the property options are:

- Area: the number of pixels in the interior.
- Diameter: the Euclidean distance between the two farthest points on the perimeter of a region.
- Perimeter: the number of pixels in the boundary.
- Euler Number: the number of objects minus the number of holes in the objects.
- Centroid: the center of mass of the object.

1.2. Fourier coefficients.

Recall from the very first class that the Fourier coefficients of a function correspond to the strength of the frequencies present in the function. For example, if a function is periodic with period 2π, then the Fourier coefficient corresponding to a frequency of $1/2\pi$ would be large and all other coefficients would be small. It turns out that you can take Fourier coefficients of the boundary curve to find out how oscillatory it is and which frequencies are present in that oscillation. I wrote two functions for you to use, one (`frdescrip`) that outputs the Fourier coefficients of the boundary and another (`ifrdescrip`) that returns the approximation to the boundary given by taking the first n coefficients. In class, I showed an example of a butterfly contour and its approximation with the first 50 coefficients (which looked like a warped diamond). Note that the first coefficients measure large scale oscillation while later coefficients measure smaller oscillations (which means that taking the first 50 coefficients will lose details in the boundary corresponding to frequencies above $1/100\pi$).

The function `frdescrip` takes in an ordered list of boundary points `S` and spits out a column vector `z` of Fourier coefficients. The syntax for taking the magnitude of the first 10 coefficients as your first 10 features would be as follows, where `v` is the feature vector.

```
>> z = frdescrip(S);
>> v(1:10) = sqrt(real(z(1:10)).^2 + imag(z(1:10)).^2);
```

If you then want to find out what the approximation to your boundary is with only 10 terms, use `ifrdescrip`:

```
>> Sapprox = ifrdescrip(z, 10);
```

and if you'd like to see the two curves plotted on the same axis, type:

```
>> plot(Sapprox(:,1), Sapprox(:,2), 'k-'); hold on
>> plot(S(:,1), S(:,2), 'r-')
```

(assumes S is a column vector of boundary points).

1.3. **Medial Axis features.** Use the skeletonization operation on a binary representation of the boundary to find the skeleton, then use hit or miss to find the branch points and the end points of the skeleton. These points, or the distances between the points, or the number of points, or some other thing you think of involving the points, can become features.

2. Texture

If you mask an image so that the only non-zero pixels are the gray-scale values in the object to be classified, statistical measurements of the object can help in classification. I have written an `m`-file `statxture` to compute those texture statistics. It outputs a vector `T` containing the following measurements:

- `T(1)`: average gray-scale value (which is the first moment of the texture).
- `T(2)`: average contrast (the standard deviation σ, which is the square root of the second moment).
- `T(3)`: smoothness measure ($R = 1 - \frac{1}{1+\sigma^2}$).
- `T(4)`: skewness (the third moment).
- `T(5)`: uniformity measure (the sum of the squared relative frequencies p_i of the gray-scale values; maximum when image is constant).
- `T(6)`: entropy (measures predictability, is zero when image is constant and goes up from there; sum of $p_i \log_2 p_i$).

If `B` is the masked image, the syntax for using the stats function is:

```
>> T = statxture(B);.
```

You can then feed the desired features from `T` into the feature vector `v`.