Image pre-processing and feature extraction

Introduction

To use image data in machine learning models, it's common to perform some sort of pre-processing. This could involve opening a set of images, converting them to grayscale, and resizing them, as we will do here, or perhaps other techniques such as normalisation.

Once your image data has been pre-processed to meet your requirements, you might want to extract specific features from the images rather than using pixel data directly. In this practical we will use the common method known as Histogram of Oriented Gradients, or HOG for short.

As with the previous practical we will provide snippets of code here, which we recommend you piece together in a Python text file or IDE (such as Spyder) and then try running for yourself.

Part 1: Image pre-processing

Opening a batch of images

The first thing we need to do is get some image data to work on. We will use the Flower dataset published by Maria-Elena Nilsback and Andrew Zisserman (https://www.robots.ox.ac.uk/~vgg/data/flowers/) .

The data is also provided on the project GitHub at https://github.com/LAR/PhenoDataCampp/MachineLearning

Once you have downloaded the data, unzipped it, and moved it to a folder of your choice we will need a way to import all the images.

One way to do this is to get a list of filenames (including the filepath). We can do this using the standard Python module **glob.** To use glob here all we need to do is provide a string giving the path or relative path to the folder containing our data, and include the 'wildcard' asterisk character * to indicate we want *all* the files in the folder matching a certain pattern.

So if our data is in a folder named **FlowerData** within the folder where you are saving your script you can use:

```
import glob
img_list = glob.glob('FlowerData/*.jpg')
```

and by adding the **'.jpg'** after the asterisk, we make sure we only pick files with a .jpg extension, i.e. we only select suitable image files.

So what have we got stored in the variable **img_list**? You can check by printing out some information:

```
print(img_list)
print(type(img_list))
print(type(img_list[0]))
print('directory has', len(img_list),'images')
```

You should see that img_list is just a list of strings referring to each of the image files in the folder. To import the image data we will need to loop through this list and open each image in turn. We can use a **for** loop and OpenCV to do this:

```
import cv2
for im_name in img_list:
    img = cv2.imread(im_name)
    print(img.shape)
```

All we do for now in the loop is open the image using the string in the image list, and print out the **shape** of the data for each. You should see a series of numbers, in groups of three, printed out. These show the number of pixel rows, pixel columns, and number of colour channels for every image. You should notice that while the third number in each triple is the same, showing we have RGB colour data, the first two numbers vary, suggesting that many of the images have different sizes.

Resizing and reshaping images

The problem with image data of varying size is that we need our set of features to be equal in size for every data point. One way around this is to resize each image to be identical in size. We can also make the dimensions of each image much smaller if we want to reduce the number of features. We can resize images quite easily using OpenCV:

```
nrow = 128
ncol = 128
col_features = nrow*ncol*3
print('each colour image will have',col_features,'features')
img = cv2.imread(img_list[13]) # an example image from our list
cv2.imshow('original image',img)
img_resized = cv2.resize(img,(nrow,ncol))
cv2.imshow('colour resized',img_resized); cv2.waitKey(0)
```

Another way to reduce the number of image features is to use grayscale rather than colour data. Again, its easy to do this in OpenCV, we just need to add a zero when we use the **imread** function:

```
gray_features = nrow*ncol
print('each grayscale image will have',gray_features,'features')
img = cv2.imread(img_list[113], 0) # another example image from our list
cv2.imshow('original image',img)
img_resized = cv2.resize(img,(nrow,ncol))
cv2.imshow('resized image',img_resized); cv2.waitKey(0)
```

It's important to remember that both these steps lose information from your data, so care should be taken depending on what you are trying to do. For example, colour data is likely to be important when identifying flower species.

Reshaping pixel data to feature data

We now have a way to resize our image data, but to put it all together we now need to reshape it into a features matrix, i.e. a flat two-dimensional table with the rows as separate data-points (in this case images) and the columns as all the features (in this case every pixel in each resized image).

The best way to do this is to make an empty matrix, full of zeros, before looping though each image and adding the resized and reshaped pixel data to each row of the blank matrix.

To make the empty matrix we can use the **zeros** function in numpy, noting that we need as many rows as there are images in our dataset, and as many columns as the total number of pixels in each image (we would need three times as many columns for colour data):

```
import numpy as np
nrow = 128
ncol = 128
n_features = nrow * ncol
n_samples = len(img_list)
X = np.zeros((n_samples,n_features))
```

Then, as we loop through the images we can use the **flatten** function to convert the 2D image data to a 1D array of numbers, and assign that to the correct row of the features matrix **X**:

```
for i,im_name in enumerate(img_list):

    img = cv2.imread(im_name,0)

    # resize/reshape
    img_resized = cv2.resize(img,(nrow,ncol))

    # flatten into array
    img_flat = img_resized.flatten()

    X[i,:] = img_flat

print(X.shape)
```

We now have a 2D matrix with 320 rows and 16,384 columns, that we could use as a features matrix in a machine learning model.

Part 2: Feature extraction Histogram of Oriented Gradients (HOG)

Often, rather than using pixel data directly, it is better to extract some image features to use to train a machine learning model. As an example of this, we will use HOG. For more details on HOG see the article earlier this week.

Scikit-image

The function we will use to calculate the HOG representation of the data is in a library we haven't used yet called scikit-image. If you have Anaconda it's likely to be installed already, otherwise you will can install it in the usual way:

```
pip install scikit-image
```

HOG representation of image data

The first thing we need to do is get some data in the right format. We can use one of the image filenames we have from earlier in the practical and open it with OpenCV:

```
image = cv2.imread(img_list[13])
```

Before we look at the image though we note that OpenCV stores colour data in the format BLUE, GREEN RED by default, so we should convert it to the more common format used by Matplotlib before we go further:

```
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

Now we have the image we can make the HOG representation as follows:

```
from skimage.feature import hog
fd, hog_image = hog(image, orientations=8,
                    pixels_per_cell=(16, 16),
                    cells_per_block=(1, 1),
                    visualize=True, channel_axis=-1)
```

This returns two variables. The first, we've called **fd**, which is the HOG data itself that we can use as a set of features in a machine learning model. The second variable, we've called **hog_image**, and is a visualisation of the result that we can now plot. This visualisation is only returned if we set **visualize=True** when we call the **hog** function.

Before we plot the hog representation, we can normalise it for clarity with the following code:

```
from skimage import exposure
hog_image_rescaled = exposure.rescale_intensity(hog_image,
                        in_range=(0, 10))
```

Now to look at the HOG image next the original image we can use the following code:

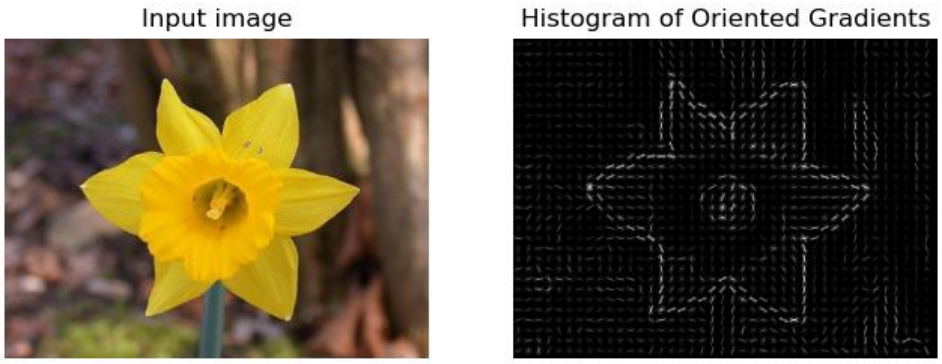
```
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4),
                  sharex=True, sharey=True)

ax1.axis('off')
ax1.imshow(image, cmap=plt.cm.gray)
ax1.set_title('Input image')

ax2.axis('off')
ax2.imshow(hog_image_rescaled, cmap=plt.cm.gray)
ax2.set_title('Histogram of Oriented Gradients')
plt.show()
```

This should display an image like the following:



The HOG visualization shows we have extracted key information about the shape displayed in the image. When you have run the code try zooming the HOG image to see exactly what it is displaying. Experiment by running the code again but changing the key parameters **orientation** (e.g. try 2 and 4 rather than eight) and **pixels_per_cell** (e.g. try 8 by 8 or 32 by 32 rather than 16 by 16).

HOG data versus pixel data

Lets go back and think about the data itself. We can find out how many pixels our original image has easily enough:

```
print('original image has shape',image.shape,
      'and so',image.shape[0]*image.shape[1],'pixels')
```

Over 300,000 pixels, so with three colour channels that could be nearly a million features for every image.

If we look at our HOG data instead:

```
print('hog representation instead has',fd.shape,'features')
```

We see that we have just around ten thousand features in comparison. Still quite a lot but a more manageable number than the original pixel data, while hopefully retaining some of the important shape data in the image.

In the next practical we will try training machine learning models using this data, so remember to save your code and image data from this practical so you can reuse it.