Python For Applied Machine Learning - Practical 3

Michael Halstead

April 14, 2024

1 Introduction

What will we do in this weeks practical?

- Argparse
- Histogram of oriented gradients
- Local binary patterns
- Sobel edge detection
- Principal component analysis

A key component of doing most computer vision based tasks is the texture within the image. If we were to classify a horizontally striped shirt we would expect there to be horizontal stripes not spots or logos. For traditional machine learning, extracting these "features" required using texture or shape extracting approaches. We will consider two of the many texture features: histogram of oriented gradients (HOG) and local binary patterns (LBP). There are many others and it's always a good idea to read into some of the others, such as Gabor wavelets, SIFT, SURF, and even the Radon transform. We will also look at an edge extracting technique called Sobel edge detection as another type of feature within an image.

2 Argparse

In last weeks practical we used the "play" button or shortcut to run our code. One of the benefits of PyCharm is the fact you can the *Terminal* option to run your code. The *Terminal* can be accessed on the bottom left of your screen, as shown by the word "Terminal" in Figure 1.

Using the run option isn't always the best option. What if you want to change some of the parameters in your models, or just change what exercise you want to run? Last week we had boolean flags for each exercise, and to change

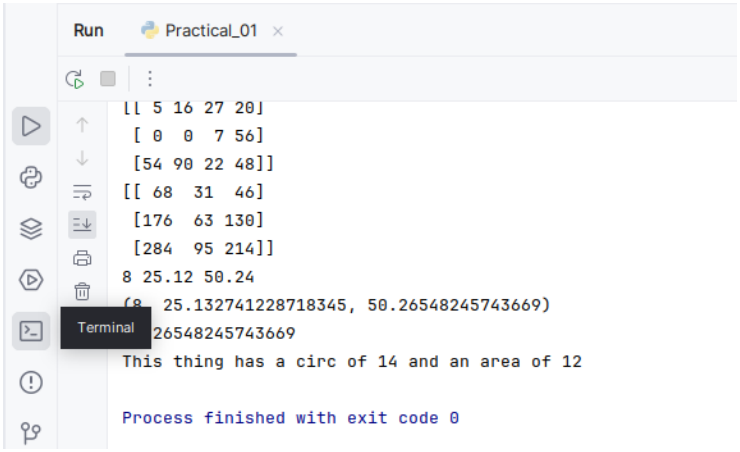

Figure 1: The terminal option in the bottom left of the screen.

the exercise we had to hard code the value to either True or False. This week we will introduce a command line parsing tool: argparse.

Why are we introducing this library now? Because, in your assignment you will need to parse command line arguments so that you can do different things in a dynamic manner. How do I run different exercises using this library? First, what is a command line argument?

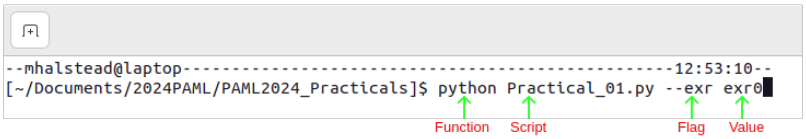

Figure 2: Example of how to call python code with argparse flags on the command line.

A command line is a flag that we need/want in our code. An example of this is which exercise we want to run: *exr0, exr1* etc. So if we want to run exercise 0 (*exr0*) in our *Practical_01.py* we would use the bash commands in Figure 2. How is this then interpreted by the python script?

First we need to import argparse, which is done like any other library. Then we need to call argparse, usually at the start of your script, it would look something like:

```
# argparse
parser = argparse.ArgumentParser( description='Extracting command
    line arguments', add_help=True )
```

```
parser.add_argument( '--exr', action='store', required=True )
flags = parser.parse_args()
```

v Where *flags* stores the information passed via the command line. In this example we have one flag called *exr* that is required at run time (when you press enter on the command line). We then need to use these flags, following the same exercise example we would do the following:

```
exr0 = True if flags.exr == 'exr0' else False
exr1 = True if flags.exr == 'exr1' else False
exr2 = True if flags.exr == 'exr2' else False
exr3 = True if flags.exr == 'exr3' else False
```

If you pass the following bash command *python Practical_01.py –exr exr0* (as seen in Figure 2) you will store a True value in *exr0* and a False in *exr1*, *exr2*, and *exr3*. You can then use these boolean variables to control which exercise you are going to run. There are lots of different options to parse information from the command line and I recommend you have a look online at these options. As we work through this weeks practical I will give you examples of how to use some of these commands.

3 Histogram of Oriented Gradients

In your code, set this exercise as exr0.

HOG is one of the most popular pure texture descriptors and is based on a sliding window approach. Using the current image window and the number of orientations it produces a histogram of those orientations. We will be using *hog* from the *from skimage.feature* library. This function has the following important variables:

1. grey scale input image,
2. orientations,
3. pixels_per_cell,
4. cells_per_block,
5. visualize,
6. feature vector.

For this practical we will be using a grey scale image that you will convert from the *texture_snippets* dataset. For a simple, horizontal and vertical, 2 orientation feature extractor you would use the following 3×3 filter banks.

```
H0 = [[-1,  0, 1],
      [-1,  0, 1],
      [-1,  0, 1]]
H1 = [[-1, -1, -1],
      [ 0,  0, 0],
      [ 1,  1, 1]]
```

These filters supply the gradients within the window being scanned. We can have more complex orientations that provide more expressive information about the image, however, more orientations does not necessarily result in better features. We will use three images from the *texture_snippets* dataset: strip_h/AA0004.jpg, plain/patternless_00003.jpg, and plaid/check09_4.jpg. You will supply the image path to the program as a flag from *argparse*:

```
parser.add_argument( '--image', action='store', default='data/
    texture_snippets/strip_h/AA0004.jpg')
```

In this case I have given it a default argument as the horizontal striped image, if I want to change it I have to give it the specific path as a command line argument. The next step is to read in the image then convert it to grayscale.

```
from skimage.io import imread, imsave
from skimage.color import rgb2gray
```

The above two lines will give you the capacity to load an image (*imread*) and convert it to grayscale. The next step is to set up your *hog* object. You will need to define the orientations (8), pixels_per_cell (8), and cells_per_block (1), where the defaults are in brackets. An example for the pixels per cell is below.

```
parser.add_argument( '--ppc', action='store', type=int, default=8 )
```

Next, create the *hog* object, we will set the visualize to True and the feature vector to False.

```
orient = flags.orient
ppc = [flags.ppc, flags.ppc]
cpb = [flags.cpb, flags.cpb]
feat, map = hog( gry,
                    orientations=orient,
                    pixels_per_cell=ppc,
                    cells_per_block=cpb,
                    visualize=True,
                    feature_vector=False )
```

Finally, we will use *matplotlib.pyplot* to show the grayscale image and the map. You should do this in subplots with two columns, where the left is the grayscale image and the right is the map (*imshow* is needed here). Once you have done this for the default values, play around with the orientation and pixels per cell to see if you can get something that visually looks better. Try it for the other two images mentioned, you'll need to supply the *–image* flag on the command line.

`https://scikit-image.org/docs/dev/api/skimage.feature.html?highlight=hog#skimage.feature.hog` is another example tutorial.

4 Local Binary Patterns

In your code, set this exercise as exr1.

Another popular texture feature are LBPs, which come in a number forms. The pure form of the local binary pattern is based on a center pixel and the

eight pixels surrounding it. The center pixel is compared to the surrounding eight pixels, if the surrounding pixel value is higher than the center we consider that pixel as binary positive. This creates a binary representation which are unravelled in the following manner:

$$\begin{matrix} 4 & 10 & 9 \\ 1 & 5 & 3 \\ 8 & 6 & 7 \end{matrix} \rightarrow \begin{matrix} 0 & 1 & 1 \\ 0 & - & 0 \\ 1 & 1 & 1 \end{matrix} \rightarrow 01100111 \rightarrow 103$$

This means, that in its standard form, LBP has values on the range of $0 \rightarrow 256$. We will once again use a grayscale image, using the same images in the previous exercise. The *local_binary_pattern* function in the *skimage.feature* library takes three primary inputs. The image to be scanned, the radius (R), and the points (P). R is the number pixels from the center that will be checked against the center pixel (we will set this to 1 as a default). And P is the number of points sampled around the circumference of that circle, which we will set to 8 as the default. You should include these as arguments to be parsed by argparse so you can play with them later.

Now we will use the local binary pattern function using the following function,

```
feats = local_binary_pattern(grayimage, R=flags.radius, P=flags.
    points)
```

After this you will plot the grayscale image and *feats* matrix in a similar manner to the previous section. What does this tell you? To see if we can make more sense of this we will plot the *feats* data as a histogram using *matplotlib*. Save this output and then redo the exercise with the other two images, can you visually tell them apart? Are there common, distinct features?

Once you have finished this we will slightly alter the LBP algorithm.

```
feats = local_binary_pattern(grayimage, R=flags.radius, P=flags.
    points,
                             method=flags.method)
```

Where, flags.method has a string default of 'default'. We will now set that to 'uniform' and redo the exercise. What do you see? Is this a better descriptor? Do you know what it is doing?

5 Sobel Edge Detection

In your code, set this exercise as exr2.

The final texture descriptor we will look at is the Sobel edge detection. It works by sliding two windows over an image and calculating the magnitude between the two window values. The following is the psuedo-code for the Sobel operator:

```
def mysobel( grayimage ):
  define hor, ver # output zero matrices of size grayimage
  define kh, kv # the vertical and horizontal kernels
  for r from 1 to rows-1
    for c from 1 to columns-1
```

```
      snippet = grayimage[r-1:r+2,c-1:c+2]
      hor[r,c] = np.sum(kh*snippet)
      ver[r,c] = np.sum(kv*snippet)
  return hor, ver, np.sqrt(hor**2+ver**2)
```

where kh and kv are:

$$kv = \begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix} \quad kh = \begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix}$$

We will also compare our Sobel edge detection algorithm to the *skimage* version. This can be accessed from *skiamge.filters* import *sobel*. This function will just return the magnitude, where as our function returns the horizontal edges, the vertical edges, and the magnitude.

Once you have calculated both our Sobel map and the *skimage* version we will plot them on subplots. It will be a 2×3 subplot with the top left being the grayscale image. In the rest you will show our sobel, *skimage*'s sobel, the horizontal edges, and the vertical edges. There will be one slot left, we will just leave that one blank.

Is your version of the Sobel operator the same as the *skimage* version?

6 Principal Component Analysis

In your code, set this exercise as exr3.

Our final exercise today will be to use principal component analysis. This is a way to reduce high dimensionality of data into more manageable dimensions. Consider something with 1000's of dimensions, this is hard for a human to recognise, it's the same for a computer. PCA does this reduction in a smart way, although deep learning has really moved past this, it's still a great tool for traditional machine learning.

We will use the inbuilt wine dataset in sklearn. To do this you will need to *from sklearn.datasets import load_wine*. You will also need to import *PCA* from *sklearn.decomposition*.

You will complete the following pseudo-code

```
X,y = load_wine # with return_X_y set to True

convert X and y to numpy arrays

create the pca object with n_components=2
define X_ as fit and transform of X

scatter plot X_ based on the labels y
```

In this code we have loaded both the data (X) and the labels (y). The final scatter plot will show how our dimensionality reduction technique has organised our data. Do you think it's done a good job?