## Creast

University of the Western Cape

Department of Computer Science

Private Bag X17, Bellville, 7535

South Africa

Tel: +27 21 959 3010

E-mail: mghaziasgar@uwc.ac.za

CSC311 – Machine Learning

Project 10: Hand Shape Recognition Using Input From a Camera

This project involves trying to recognize a handshape (there are six specific hand shapes in total) performed by a person, as captured by a single camera; given the image of the person’s hand (performing one of the six hand shapes), the goal here is to use machine learning to correctly classify the image into one of the six hand shapes. This has applications in gesture-based interfaces such as in gaming and augmented/virtual reality, sign language recognition and other areas. The table below has two examples of each of the hand shapes from the data set.

|  |  |
| --- | --- |
| **Hand Shape** | **Two Examples** |
| **0** |  |
| **1** |  |
| **2** |  |
| **3** |  |
| **4** |  |
| **5** |  |

The data file you have been given contains

* 1200 rows i.e. 1200 data examples (200 examples per hand shape);
* 12289 columns that are as follows; 12288 features that represent the image of the specific hand shape followed by (the last column – column 12289) which is the hand shape label (0 to 5) as shown in the table above.

It is quite possible for you just treat the 12288 features as some relevant features that can be trained on and used to determine if an image contains a cat or not. You don’t necessarily need to know what the features represent.

However, if you really want to know (what the 12288 features represent), here it is. Each example is a “flattened” image representation of one of the six hand shapes shown in the table above.

Each image is a 64×64 pixel image. A pixel is essentially a (very tiny) dot in a digital image that represents a specific colour in some form; so the images in this database are 64 pixels wide and 64 pixels high. The images in this data set are colour images, so each pixel (dot) in the image is actually represented as a three-tuple (R,G,B) which defines the amount of Red, Green and Blue in the specific pixel; and each of these ranges from 0 to 255. When the red, green and blue colours of a given pixel in their specific amounts are mixed, they appear as a dot of a specific colour when displayed on the screen. A few examples:

* (0,0,0) means no colour at all, so this pixel appears black when displayed on the screen;
* (255,255,255) means full amounts of R,G and B, so this pixel appears as white light on the screen;
* (255,0,0) represents full amount of R, and no G or B, so this pixel represents a very pure red colour when displayed on the screen
* (128,128,128) represents half R, G and B, so this pixel (believe it or not) appears as a gray pixel (not too dark or light) on the screen.

If you’re unsure about all of this, feel free to do some research on it if you really want to. Now, although the original data came in the form of these 2D 64×64 images with 3 colour components (called “colour channels”) for each pixel, I “flattened” all pixels and colour channels into a single row by stacking together all rows of the image into one long row; to be specific I called the reshape(259,64\*64\*3) command on the entire matrix containing the images; this resulted in a matrix with 259 rows and 12288 columns (features). I did this to be able to write all the features (pixels) of each image onto a single line in the data file (as we’ve become used to).

So now, with the flattened features, each data example is organized as follows: features 1, 2 and 3 represent the R,G,B values of the first pixel in the first row; features 4, 5 and 6 represent the R,G,B values of the second pixel in the first row; … ; features 190, 191 and 192 represent the R,G,B values of the last (64th) pixel in the first row; features 193, 194 and 195 represent the R,G,B values of the first pixel in the second row; and so on and so forth until; features 12286, 12287 and 12288 represent the R,G,B values of the last (64th) pixel in the last (64th) row; alas, the rows have been concatenated.

When reading the csv file, I would strongly suggest using the following code (if you want to be able to display the images):

data = np.genfromtxt(datafilename,delimiter=",", dtype="int32")

You can then proceed to split “data” up into X (all but the last column) and y (the last column). Then, it is possible for you to reconstruct the (now) 1D flattened images in X into 2D images using the following code:

Xrecons = X.reshape(TotalNumberOfExamples,64,64,3)

You can now display any of the 1200 images using the matplotlib (yes, it is quite a box of wonders) using the following code:

import matplotlib.pyplot as plt

indextoshow = 0 #Change this as desired

plt.imshow(Xrecons[indextoshow])

**Tips**: I would strongly suggest that you ensure that you include equal numbers of each hand shape in the training set e.g. 100-120 images of each hand shape for training, and then the rest split 50-50 between CV and Test.

Also, although you can use the usual feature scaling technique we spoke about in class, when it comes to image pixels (that always range from 0 to 255), you can achieve feature scaling much faster and easier by simply dividing your entire X matrix by 255 (taking care to ensure that your Numpy array converts to a float). This results in values between 0 and 1.