Additional Instructions for using the MLP example

MLP is part of the ML (Machine Learning) module in OpenCV.
This is a simple set of instructions on how to modify the sample code to use in assignment 3.

1) Download and run the example

to compile MLP_example.cpp:

```
g++ MLP_example.cpp -o MLP_example `pkg-config --cflags --libs opencv`
```

Now you need to use a dataset with some classes and features. The sample dataset is called Glass.data. The data is in the following format (each row is a sample of the data):

```
1,1.52101,13.64,4.49,1.1,71.78,0.06,8.75,0,0
```

The attributes are separated by a comma. The first attribute is the **class** to which the row belongs to.
In the case of the Glass.data, there are only 7 classes, but class 4 does not show in any part of the dataset. The rest of the row is compose of 9 attributes, these are the **features**.

In the case of the gestures, you need to create a similar txt file with the classes from 0 to 9. The rest of the attributes are the features (the list of Fourier Descriptors for each image of the gestures).

The training step is to let the MLP algorithm use the dataset to train a classifier. If the classifier is called example.xml, the command is:

```
./MLP_example -save example.xml -data Glass.data
```

To test the training error with an existing classifier example.xml, use:

```
./MLP_example -load example.xml -data Glass.data
```

The answer will come in the format:

```
...
Sample: 7 row [1.51711, 14.23, 0, 2.0799999, 73.360001, 0, 8.6199999,
1.67, 0]
Predict:  r = 0

Test Recognition rate: training set = 60.7%

Prediction: 1
...
```

The `Sample:` lines repeats the information from the txt file. It says that there was a sample with class 7, and the predicted class was 0 (which is a mistake made by the classifier, the training is not perfect yet).
The `Test Recognition rate` line tells what was the total number of samples correctly classified. It was only 60.7%.
The `Prediction:` line shows an example of a single sample being predicted. You need to look at the source code to see if the prediction was correct or not (in the case of the first prediction it is correct).

2) Use a classifier to predict a single sample of data

The OpenCV original example only used data from the text file. To help you out, I also included an example of using a single sample and predicting what is its class.

The single sample should present the same number of features that the training data had when the classifier was trained. If you trained with 10 **classes** and 23 **features**, you need to present a row with 23 features for the prediction too, and the answer is going to be one of the 10 classes.

The single sample prediction code for the Glass.data (7 classes with 9 features) is like that:

```
Mat sample1 = (Mat_<float>(1,9) << 1.52101, 13.64, 4.4899998, 1.1,
71.779999, 0.059999999, 8.75, 0, 0);
float r = model->predict( sample1 );
cout << "Prediction: " << r << endl;
```

The method model->predict is going to use the classifier example.xml (already loaded by the function `load_classifier()` in line 186) to predict what is the class with the data loaded onto sample1. You can use a similar code to try to predict the gesture for every frame. Remember that you only need to load the classifier once, at the beginning of the program before opening the webcam. Then the frame is used to compute the contour of the hand, compute *n* Fourier Descriptors and use the `model->predict` method to see what gesture it is.

You can use the code with only minor modifications to train your classifier (the xml file). Then you can copy the lines to load the classifier, and use the single sample prediction code (above) to predict each frame.

3) Changing the training parameters on MLPs

You need to change at least the **number of classes** and the **number of features**.
These are found in lines 170 and 174:

```
170 const int class_count = 7;//CLASSES
171 Mat data;
172 Mat responses;
173
174 bool ok = read_num_class_data( data_filename, 9, &data,
&responses );//third parameter: FEATURES
```

The example above has 7 classes with 9 features in each row (each row has 10 attributes, the first being the class).
Modify that for 10 classes and *n* features (you should try something between 10 to 30 features).

In line 181 there is a parameter split. This is to split the set into training and test. Keep it at 1.0, as this will use all the available data to train the gestures.

```
181 int ntrain_samples = (int)(nsamples_all*1.0);//SPLIT
```

If you know about neural networks you can also change the number of hidden layers (in the program it is a function of the `layer_sz[]`). The number of inputs is the number of features (`data.cols`) and the

number of outputs is the number of classes (`class_count`). The number of layers is decided by line 216:

```
216 int layer_sz[] = { data.cols, 100, 100, class_count };
```

This example is going to have a ANN of 4 layers, the input with `data.cols` nodes, two hidden layers with **100** nodes each, and the output layer with `class_count` nodes.

For example, to train an ANN with only **3** layers with a **50** node hidden layer you would have to change line 216 to:

```
216 int layer_sz[] = { data.cols, 50, class_count };
```

Or to increase it to 5 layers, with 75 nodes in the first hidden layer, 50 on the second and 40 on the last, you would code:

```
216 int layer_sz[] = { data.cols, 75, 50, 40, class_count };
```

You have to experiment with these parameters until you get something above 97% of the training samples classified correctly to have a good rate with the camera.

Other training parameters can be changed in lines 223 to 225:

```
223         int method = ANN_MLP::BACKPROP;
224         double method_param = 0.001;
225         int max_iter = 5300;
```

The default `method` uses back-propagation to train the ANN. This is the method of choice. There are other methods available, but the default works well.
The `method_param` is the learning speed. This is a good default, but you can experiment with slightly different numbers.
The most important parameter to change is the max_iter. This indicates how many cycles of training the ANN goes through. Start with a small number like 300 to keep the training process fast.
If the correct predictions in the training data are below 97%, increase the number of iterations (`max_iter`). Don't increase it too rapidly because ANNs overfits very easily. Try to increase by 200 or 300 at a time until you get good results.

Once you train the classifier (the xml file), then you don't need to use the `MLP_example.cpp` anymore. You can copy the relevant code into your assignment code (load the classifier, copy the prediction lines from the functions etc).

4) Using the classifier with the camera in the assignment code

Once you trained the classifier, store the xml file carefully. Make a note of the parameters that you used to train it in a separate file. In the main function, initialise the model:

```
Ptr<ANN_MLP> model;
model = load_classifier<ANN_MLP>("yourclassifier.xml");
```

The load_classifier function can be coded like:

```
template<typename T>
static Ptr<T> load_classifier(const string& filename_to_load)
{
    // load classifier from the specified file
    Ptr<T> model = StatModel::load<T>( filename_to_load );
    if( model.empty() )
        cout << "Could not read the classifier " << filename_to_load <<
endl;
    else
        cout << "The classifier " << filename_to_load << " is loaded.\n";

    return model;
}
```

Now suppose that the Fourier Descriptors are float numbers on an array called CE[].
The feature are computer for ***every frame***. You can load these features onto a matrix sample1:

```
Mat sample1 = (Mat_<float>(1,10) <<
CE[1],CE[2],CE[3],CE[4],CE[5],CE[6],CE[7],CE[8],CE[9],CE[10]);
```

And predict which gesture it is:

```
float r = model->predict( sample1 );
```

`(int) r` will contain the number of the gesture, which can be printed in the window showing the hand.