Automatic Facial Expression Recognition
A Practical Tutorial

Michel Valstar

Monday 6th July, 2015

Chapter 1

Introduction

This tutorial manual was written for an *atelier* (i.e. a workshop) on Automatic Facial Expression Recognition as part of the 2013 Summer School on Social Human-Robot Interaction, held on 29 August 2013 in Christ's College, Cambridge, UK. It is meant to be a practical introduction to automatic facial expression recognition, which I hope can serve as a starting point for those interested in this rapidly growing area of research. So, while I have tried to keep things simple, I have also tried to clearly point out where I've simplified matters and where possible I have given pointers to more advanced techniques.

The tutorial assumes you have Matlab installed, and have basic Matlab programming skills. The package in which you've found this manual includes all the necessary Matlab code, as well as the example data that we will use, which has been taken from the MMI Facial Expression Database [7]. If your work results in any publications and uses either the facial images or the LGBP(-TOP) dynamic appearance descriptor code [1] I hope you will return the courtesy to properly credit those works.

I hope this tutorial is of some use to all of you, and perhaps may even inspire some to take up this wonderful mix of computer vision, machine learning, and human psychology and anatomy. This manuscript will hopefully improve over time, so I am very happy to receive any feedback that will allow me to improve the tutorial for future users.

Chapter 2

Tutorial - Building a Simple Smile Detector

Face Detection

We're going to start with face detection.

Step 1: Load an image. The first thing to do so we can play around a little is to load an image. We'll do that in steps. Go to the directory where you stored this tutorial's matlab code, for example:

```
cd /Users/Me/Documents/SomeDir/matlab/
```

All example images we will be using are in a directory relative to this, to wit `data/AU12/Positive` for images including Action Unit 12 (AU12, smiles) and `data/AU12/Negative` for images without AU12. See the Facial Action Coding System manual [2] for a complete description of Action Units. An image can now be loaded, converted to greyscale, and displayed as follows:

Listing 2.1: Load image

```
1 cim = imread('../data/AU12/Positive/6-50.png');
2 im = rgb2gray(cim);
3
4 fig = figure;
5 subplot(1,2,1);
6 imshow(cim);
7 subplot(1,2,2);
8 imshow(im);
```

Lines 4-8 are optional, they're just used to check that the image was loaded correctly. Note that the variable `cim` holds the colour image, and `im` the greyscale image. Both are needed at various stages of this tutorial.

Step 2: Face detection. We will detect the face using the toolbox provided by Zhu and Ramanan [8]. Their code is publicly available, and we will be using it here because it is probably the most accurate face detector available. Another popular face detector is that by Viola & Jones, which is integrated in e.g. Matlab and OpenCV. While OpenCV's detector is much

faster, Zhu and Ramanan's detector (ZR detector) attains higher accuracy in most cases and can in particular deal with non-frontal head pose.

To use the ZR detector, add the face detection code to Matlab's search path and compile the c-code that is part of their package as follows::

```
addpath face-release1.0-basic;
cd face-release1.0-basic/
compile;
cd ..
```

The code should now be ready to detect faces. Please note that it operates on colour images. Listing 2.2 below shows how to use the ZR detector to detect faces:

Listing 2.2: Face detection

```
load face_p146_small.mat;

% -- 5 levels for each octave
model.interval = 5;
% -- set up the threshold
model.thresh = min(-0.65, model.thresh);

% -- define the mapping from view-specific mixture id to
    viewpoint
if length(model.components)==13
    posemap = 90:-15:-90;
elseif length(model.components)==18
    posemap = [90:-15:15 0 0 0 0 0 0 -15:-15:-90];
else
    error('Can not recognise this model');
end

bs = detect(cim, model, model.thresh);
bs = clipboxes(cim, bs);
bs = nms_face(bs,0.3);

figure,showboxes(cim, bs(1),posemap),title('Highest scoring
    detection');
```

Please try out this code for yourself. Also try it out on a number of other images, either from the example set or from random images downloaded from the internet. Please note that the detector is quite slow, partly because of the large number of poses tested. If you know beforehand what the expected head pose in your images is, you could create new models for this with fewer poses. This would make the detector faster.

Because this is quite a handful of code to type in all the time, we turned it into a handy little function called `zrdetectface`, which is included in this package:

```
load face_p146_small.mat;
bs = zrdetectface(cim, model, show);
```

Where `show = 0` if you don't want the results to be shown, which can be handy if you are processing a large number of images (which we'll do later). Note that this function needs to be passed the face detection model (e.g. the one stored in the file face_p146_small.mat). This avoids having to load the model each time, which would slows things down. It would also put constraints on the path to execute the code if we loaded the model inside this function.

Face Registration

We want our appearance descriptors to encode changes in appearance caused by facial expression, not by changes in head position or differences between individuals. In the face registration step, we therefore try to normalise for head pose by removing any in-plane head rotation.

Note that what we will do here is only a very basic approach to intra-person registration, and will virtually not address inter-person registration at all, besides a scaling based on the distance between the eyes. A slightly more advanced approach would also take into account the location of other major facial features such as the nose, mouth, and chin. Ideally you would also normalise for out-of-plane head pose and to a greater extent for identity than we do here, but that is for a large part still active research.

To normalise for the scale of the face what we will do here is simply setting the distance between the eyes to a fixed value (100 pixels here). To normalise for in-plane head pose we make sure that the angle between the line that connects the eyes and the horizontal is equal to zero, which is a roundabout way of saying we're turning the face upright.

The following code should give us a registered face:

Listing 2.3: Face registration

```
function f = zrregisterface(im, bs)
%Register face using Zhu and Ramanan's face detection results
%POST: this implementation will result in rotation artefacts, but it makes
%the code more clear. Consider it an exercise to remove the black
%triangular areas created by imrotate

    % -- Get right and left eye coordinates from the boxes
    re = [mean(bs.xy(15,[1, 3])), mean(bs.xy(15,[2,4]))];
    le = [mean(bs.xy(26,[1, 3])), mean(bs.xy(26,[2,4]))];
    nose = [mean(bs.xy(1,[1, 3])), mean(bs.xy(1,[2,4]))];

    % -- Scale the face
    d = sqrt(sum((re-le).^2));
    s = 100/d;
    f = imresize(im, s);
    re = s*re; % - right/left eye coords in new space
    le = s*le;
    nose = round(s*nose);

    % -- Center face on the nose
    f = f(nose(2)-100:nose(2)+99, nose(1)-100:nose(1)+99);
```

```
22
23      % -- Determine angle with horizontal and rotate image
24      alpha = atan2(-(le(2)-re(2)), le(1)-re(1));
25      f = imrotate(f, -alpha*57.2958, 'crop'); % - Specify 'crop'
            to not change point coords
26
27
28  end
```

Again, we've turned this into a function so you can call it directly:

```
f = zrregisterface(im, bs);
```

where `im` is a greyscale image, and `bs` is the Zhu and Ramanan face detection result. `f` stores what we call the face box, it should be 200x200 pixels and have the eyes horizontally aligned. You can use the matlab built-in function `imshow` to check if `f` indeed contains the registered face.

The code in listing 2.3 uses only one point for the left eye, and one for the right. This means that even small errors in the localisation of those points will result in large registration errors. A simple but efficient improvement would be to use more points for the registration. Note that you can only use stable facial points for this, that is, facial points that don't move due to facial expression, only due to rigid head motion.

Local Gabor Binary Pattern Appearance Descriptors

Using the greyscale image directly is not optimal for a number of reasons, such as high dimensionality, high Lambertian variation with illumination changes, and a high sensitivity to small face misalignments. Finding the right appearance descriptors is an active research field. One of the most successful recent developments are Local Gabor Binary Patterns (LGBP) for static faces [1, 4], and its dynamic extension LGBP from Three Orthogonal Planes (LGBP-TOP), that can encode facial actions as well as static appearance changes.

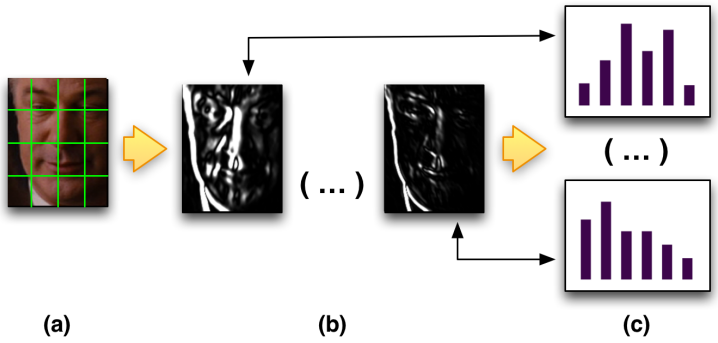

Figure 2.1: Local Gabor Binary Patterns (LGBP)

For this tutorial we will be working with the static variant only, because that only requires the data to be individual images. For TOP or other dy-

namic appearance descriptors, one would need consecutive images to create images from. As Almaev & Valstar pointed out recently, the dynamic appearance descriptors are more accurate and more robust to alignment errors [1], so if you're going to be serious about expression recognition I would recommend you do use that. The LGBP and LGBP-TOP matlab implementations are provided with this tutorial. An overview of LGBP-TOP is provided in Fig. 2.2.

To provide some alignment robustness and reduce the dimensionality of the problem, we split the face into 4 x 4 blocks, and obtain a histogram of the descriptor output per block, which are in turn concatenated into a single feature vector (see Fig. 2.1). Given the face f, you can get this feature vector as follows:

```
addpath LGBP-TOP;
gb = [2, 3, 3, 2.1, 0.55, 1.2];
x = LGBP(f, [4 4], gb);
```

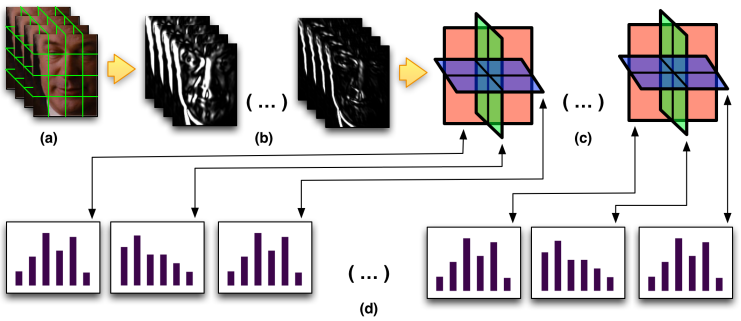

Figure 2.2: Local Gabor Binary Patterns from Three Orthogonal Planes (LGBP-TOP)

Note that the number of blocks to use is a design parameter, which should be found using either cross-validation or a separate validation set. The number of blocks is proportional to the number of features and therefore to the amount of training data you have. If you have few training data, you should keep the number of blocks low. For the same reason the number of Gabor scales and orientations has been reduced to 2 and 3, respectively, compared to the paper on LGBP-TOP.

Building a smile detector

We have now dealt with all the computer-vision aspects of our smile detector - face detection, registration, and feature extraction. We now need to create a machine learning model, or hypothesis, that can distinguish between images with or without smiles.

To do so, we will use supervised learning, which means that we need to collect a set of training data of positive examples (images with smiles) and negative examples (images with any expression but smiles). Because the

focus of this tutorial isn't on machine learning, we will use a simple non-parametric model, k-Nearest Neighbours (kNN). Of course, if you wish you can replace this later with a more suitable classifier such as Support Vector Machines (SVM).

But first, let's collect all the positive and negative data. To do so, we need to loop over all images in the Positive and Negative directories.

The following code does exactly that:

Listing 2.4: Collect dataset

```
function [x, y] = collectTrainingSet(d)
%PRE - d is the directory with training data, organised by having all
%positive examples in a sub-directory called 'Positive', and the negative
%examples in a sub-directory called 'Negative'

    dpos = [d, '/Positive/'];
    dneg = [d, '/Negative/'];

    dir_pos = dir([dpos, '*.png']);
    dir_neg = dir([dneg, '*.png']);

    % -- Load model for face detection
    load face_p146_small.mat;
    fig = figure;
    set(fig, 'Position', [216 335 1093 420]);

    % -- Set Gabor filter options:
    gb = [2, 3, 3, 2.1, 0.55, 1.2];

    n = 5664; % - Dimensionality of our features (hardcoded for speed)
    m = length(dir_pos) + length(dir_neg);
    x = zeros(m,n); % - Empty feature matrix
    y = zeros(m,1); % - Empty label matrix
    j = 0;
    for i = 1:length(dir_pos)
        j = j+1;
        fn = [dpos, dir_pos(i).name];
        cim = imread(fn);
        subplot(1, 3, 1), imshow(cim), title('Input image');
        im = rgb2gray(cim);
        [bs, posemap] = zrdetectface(cim, model, 0);   % - Don't visualise face det
        subplot(1,3,2);
        showboxes(cim, bs(1),posemap),title('Zhu & Ramanan face detection');
        f = zrregisterface(im, bs);
        subplot(1,3,3), imshow(f), title('Registered image');
        hold on;
        M = size(f,1);
        N = size(f,2);
        for k = 1:50:M
            Ix = [1 N];
            Iy = [k k];
            plot(Ix,Iy,'Color','w','LineStyle','-');
            plot(Ix,Iy,'Color','k','LineStyle',':');
```

```
44          end
45
46          for k = 1:50:N
47              Ix = [k k];
48              Iy = [1 M];
49              plot(Ix,Iy,'Color','w','LineStyle','-');
50              plot(Ix,Iy,'Color','k','LineStyle',':');
51          end
52          hold off;
53          drawnow;
54          x(j,:) = LGBP(f, [4 4], gb);
55          y(j,1) = 1;
56      end
57
58      for i = 1:length(dir_neg)
59          j = j+1;
60          fn = [dneg, dir_neg(i).name];
61          cim = imread(fn);
62          subplot(1, 3, 1), imshow(cim), title('Input image');
63          im = rgb2gray(cim);
64          bs = zrdetectface(cim, model, 0);   % - Don't visualise
                face det
65          subplot(1,3,2);
66          showboxes(cim, bs(1),posemap),title('Zhu & Ramanan face
                detection');
67          f = zrregisterface(im, bs);
68          subplot(1,3,3), imshow(f), title('Registered image');
69          hold on;
70          M = size(f,1);
71          N = size(f,2);
72          for k = 1:50:M
73              Ix = [1 N];
74              Iy = [k k];
75              plot(Ix,Iy,'Color','w','LineStyle','-');
76              plot(Ix,Iy,'Color','k','LineStyle',':');
77          end
78
79          for k = 1:50:N
80              Ix = [k k];
81              Iy = [1 M];
82              plot(Ix,Iy,'Color','w','LineStyle','-');
83              plot(Ix,Iy,'Color','k','LineStyle',':');
84          end
85          hold off;
86          drawnow;
87          x(j,:) = LGBP(f, [4 4], gb);
88          y(j,1) = -1;
89      end
90
91  end
```

You will notice that the registration is not always perfect. This is because the Zhu & Ramanan method, while being very good for face detection, isn't that well suited for facial point localisation. A better method of doing registration is by applying the *Project-Out Cascaded Regression* facial point localisation technique of Tzimiropoulos [5]. Code for this is online available: `http://www.cs.nott.ac.uk/~yzt/code/PO_CR_code_v1.zip`

The codes takes a rather long time to execute. For your convenience, we've run it for you earlier, and included the results in this package as `x.m` and `y.m` for the features and labels, respectively. Load them as follows:

```
load x;
load y;
```

The training data is now stored in the variables `x` and `y`.

Let's see how good our data is at predicting whether a smile is present or not using kNN. To do so, we are going to do leave-one-out cross validation. In this process we will split our data in as many folds as we have instances, and in every fold we 'train' our system on all folds but the single fold (and thus instance) that we will test on. We repeat this iteratively until all folds are used as the test fold exactly once, and we thus have a prediction on all our data, and all predictions on instances are obtained without observing that particular instance during training.

The code in listing 2.5 shows how this is done.

Listing 2.5: Leave one out cross validation

```
function [p, c] = crosvalknn_loo(x, y, k, P)
%IN:  x: features
%     y: labels
%     k: number of nearest-neigbhours to use in kNN voting
%     P: number of principal components to retain
%OUT: p: class predictions
%     c: classification rate

    n = size(y,1);
    p = zeros(size(y));

    if ~exist('P', 'var')
        P = 5;
    end

    % -- Apply PCA to reduce dimensionality
    [U, mu, ~] = pca(x');  % - Piotr's toolbox uses inverse
        definition of r/c
    Xk = pcaApply(x', U, mu, P);
    Xk = Xk';

    % -- Start cross validation!
    for i = 1:n
        I = setdiff(1:n, i);
        X1 = Xk(I, :);
        Y1 = y(I);
        X2 = Xk(i, :);
        idx = knnsearch(X1,X2,'dist','cityblock','k',k);
        p(i) = sign(mean(Y1(idx)));
    end

    p(p==0) = -1;   % - Matlab says sign(0) == 0
    c = sum(p == y)/n;  % - Classification rate

end
```

.

This will show, that with $k = 1$ and $P = 9$, we will get a classification accuracy of 94%. Try this for yourself, varying for example the variable `k` and `P`. Also inspect the code and observe how the dimensionality of the original problem is reduced by applying Principal Component Analysis (PCA). PCA is a simple linear method for dimensionality reduction that does not take the labels of the data into account, purely the variance of the data. If you are thinking of improving the approach we outlined here, replacing PCA might be another thing you can do.

Note that we use Piotr's machine learning toolbox for PCA, as it employs a more intuitive way of using PCA than Matlab's built-in functions. To use it, you have to add the toolbox to the Matlab search path:

```
addpath 'Piotr Toolbox/toolbox/classify/';
addpath 'Piotr Toolbox/toolbox/matlab/';
```

Discussion and further exercises

The leave-one-out cross validation results with the first five principal components and kNN, $k = 1$ sounds rather good, but really this result should be viewed in the context of this tutorial. Firstly, a naive classifier would get as high as 66% simply by predicting all instances to be of the negative class. Secondly, if you looked at the data, you may have noticed that there are a rather large number of images for each person included. This means that it's likely that for every test image there will be a close match in the training data. If you want to see how well the approach followed here generalises to unseen people, you will have to do a subject independent evaluation, e.g. leave-one-subject-out.

Another simplification is that we use posed, large intensity smiles from frontal faces. In reality, smiles vary in intensity, they are often mixed with other Action Units, and head pose is rarely frontal. AU detection is still very much an unsolved problem, as a recent challenge pointed out [6].

On the other hand, we used very little data to 'train' our system, and we used a very simple non-parametric classifier. Also, while the Zhu and Ramanan face detector is very good at finding faces, it's performance at locating facial points is not that good, as you may have noticed (at least not when using the particular model used here). You could switch to a better facial point detector such as that published by Martinez et al. [3]. The point detection results on this set are included in the folder. An interesting exercise would be to use the facial points on this dataset and create a simple geometric feature approach, where you employ the locations of the facial points to detect a smile.

Another good exercise would be to use a more complex classifier that is known to attain very good results, such as Support Vector Machines (SVMs). Note that this is a parametric classifier, with a few parameters that need to be optimised such as the slack variable C and one or more kernel parameters, depending on the kernel you choose. An important thing to bear in mind

here is that to attain systems that can generalise well to unseen subjects it is of paramount importance that the parameters are optimised in a subject-independent manner as well. Usually this means creating cross-validation loops where all data of a subject is included in a single fold. Unfortunately popular machine learning tools such as WEKA do not allow you to do this, which means that you will always get sub-optimal results if you choose to use that.

Bibliography

[1] T. Almaev and M. Valstar. Local gabor binary patterns from three orthogonal planes for automatic facial expression recognition. In *Proc. Affective Computing and Intelligent Interaction*, 2013. in print.

[2] P. Ekman, W. V. Friesen, and J. C. Hager. *FACS Manual*. Network Information Research Corporation, May 2002.

[3] B. Martinez, M. Valstar, X. Binefa, and M. Pantic. Local evidence aggregation for regression based facial point detection. *Transactions on Pattern Analysis and Machine Intelligence*, 35(5):1149–1163, 2013.

[4] T. Senechal, V. Rapp, H. Salam, R. Seguier, K. Bailly, and L. Prevost. Facial Action Recognition Combining Heterogeneous Features via Multikernel Learning. *IEEE Trans. Systems, Man, and Cybernetics, Part B (Cybernetics)*, 42(4):993–1005, 2012.

[5] G. Tzimiropoulos. Project-out cascaded regression with an application to face alignment. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[6] M. Valstar, M. Mehu, B. Jiang, M. Pantic, and K. Scherer. Meta-analysis of the first facial expression recognition challenge. *IEEE Transactions on Systems, Man, and Cybernetics-B*, 42(4):966–979, 2012.

[7] M. F. Valstar and M. Pantic. Induced disgust, happiness and surprise: an addition to the mmi facial expression database. In *Proc. 3rd Intern. Workshop on EMOTION (satellite of LREC): Corpora for Research on Emotion and Affect*, pages 65–70, 2010.

[8] X. Zhu and D. Ramanan. Face detection, pose estimation and landmark localization in the wild. In *Proc. IEEE Int'l Conf. Computer Vision and Pattern Recognition*, 2012.