Artificial Intelligence Nanodegree¶

Computer Vision Capstone

Project: Facial Keypoint Detection

Welcome to the final Computer Vision project in the Artificial Intelligence Nanodegree program!

In this project, you’ll combine your knowledge of computer vision techniques and deep learning to build and end-to-end facial keypoint recognition system! Facial keypoints include points around the eyes, nose, and mouth on any face and are used in many applications, from facial tracking to emotion recognition.

There are three main parts to this project:

Part 1 : Investigating OpenCV, pre-processing, and face detection

Part 2 : Training a Convolutional Neural Network (CNN) to detect facial keypoints

Part 3 : Putting parts 1 and 2 together to identify facial keypoints on any image!

Here's what you need to know to complete the project:

1. In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested.

   a. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!
1. In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation.

   a. Each section where you will answer a question is preceded by a **'Question X'** header.

   b. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**.

Note: Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

The rubric contains **optional** suggestions for enhancing the project beyond the minimum requirements. If you decide to pursue the "(Optional)" sections, you should include the code in this IPython notebook.

Your project submission will be evaluated based on your answers to *each* of the questions and the code implementations you provide.

Steps to Complete the Project

Each part of the notebook is further broken down into separate steps. Feel free to use the links below to navigate the notebook.

In this project you will get to explore a few of the many computer vision algorithms built into the OpenCV library. This expansive computer vision library is now almost 20 years old and still growing!

The project itself is broken down into three large parts, then even further into separate steps. Make sure to read through each step, and complete any sections that begin with **'(IMPLEMENTATION)'** in the header; these implementation sections may contain multiple TODOs that will be marked in code. For convenience, we provide links to each of these steps below.

Part 1 : Investigating OpenCV, pre-processing, and face detection

- Step 0: Detect Faces Using a Haar Cascade Classifier
- Step 1: Add Eye Detection
- Step 2: De-noise an Image for Better Face Detection
- Step 3: Blur an Image and Perform Edge Detection
- Step 4: Automatically Hide the Identity of an Individual

Part 2 : Training a Convolutional Neural Network (CNN) to detect facial keypoints

- Step 5: Create a CNN to Recognize Facial Keypoints
- Step 6: Compile and Train the Model
- Step 7: Visualize the Loss and Answer Questions

Part 3 : Putting parts 1 and 2 together to identify facial keypoints on any image!

- Step 8: Build a Robust Facial Keypoints Detector (Complete the CV Pipeline)

Step 0: Detect Faces Using a Haar Cascade Classifier

Have you ever wondered how Facebook automatically tags images with your friends' faces? Or how high-end cameras automatically find and focus on a certain person's face? Applications like these depend heavily on the machine learning task known as *face detection* - which is the task of automatically finding faces in images containing people.

At its root face detection is a classification problem - that is a problem of distinguishing between distinct classes of things. With face detection these distinct classes are 1) images of human faces and 2) everything else.

We use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `detector_architectures` directory.

Import Resources

In the next python cell, we load in the required libraries for this section of the project.

In [1]:

```
# Import required libraries for this section

%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
import math
import cv2                     # OpenCV library for computer vision
from PIL import Image
import time
```

Next, we load in and display a test image for performing face detection.

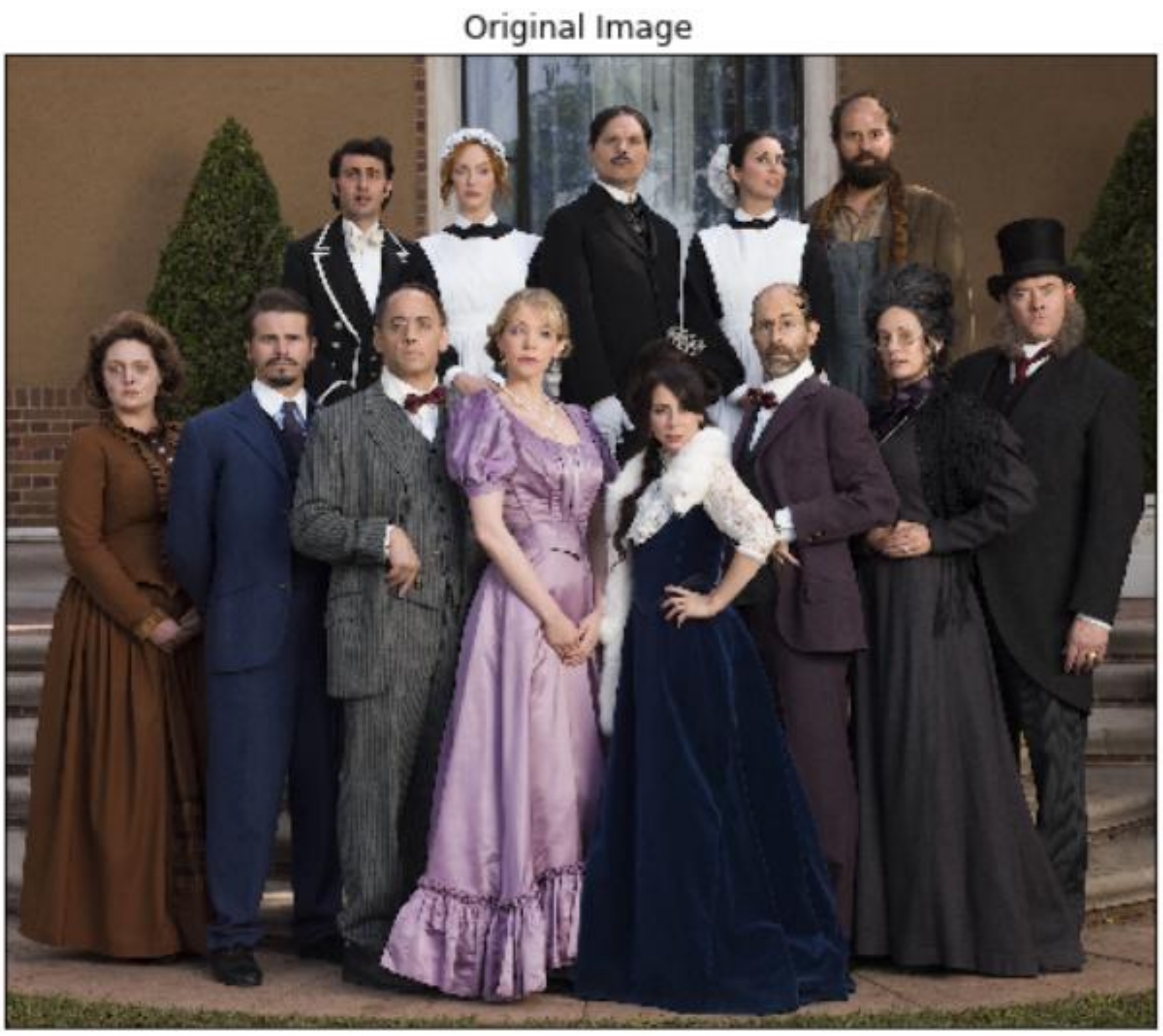
Note: by default OpenCV assumes the ordering of our image's color channels are Blue, then Green, then Red. This is slightly out of order with most image types we'll use in these experiments, whose color channels are ordered Red, then Green, then Blue. In order to switch the Blue and Red channels of our test image around we will use OpenCV's `cvtColor` function, which you can read more about by checking out some of its documentation located here. This is a general utility function that can do other transformations too like converting a color image to grayscale, and transforming a standard color image to HSV color space.

In [2]:

```
# Load in color image for face detection
image = cv2.imread('images/test_image_1.jpg')

# Convert the image to RGB colorspace
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Plot our image using subplots to specify a size and title
fig = plt.figure(figsize = (8,8))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])

ax1.set_title('Original Image')
ax1.imshow(image)
```

Out[2]:

```
<matplotlib.image.AxesImage at 0x21317c77940>
```



There are a lot of people - and faces - in this picture. 13 faces to be exact! In the next code cell, we demonstrate how to use a Haar Cascade classifier to detect all the faces in this test image.

This face detector uses information about patterns of intensity in an image to reliably detect faces under varying light conditions. So, to use this face detector, we'll first convert the image from color to grayscale.

Then, we load in the fully trained architecture of the face detector -- found in the file *haarcascade_frontalface_default.xml* - and use it on our image to find faces!

To learn more about the parameters of the detector see this post.

In [3]:

```
# Convert the RGB  image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

# Extract the pre-trained face detector from an xml file
face_cascade =
cv2.CascadeClassifier('detector_architectures/haarcascade_frontalface_default.xm
l')

# Detect the faces in image
faces = face_cascade.detectMultiScale(gray, 4, 6)

# Print the number of faces detected in the image
print('Number of faces detected:', len(faces))

# Make a copy of the orginal image to draw face detections on
image_with_detections = np.copy(image)
```

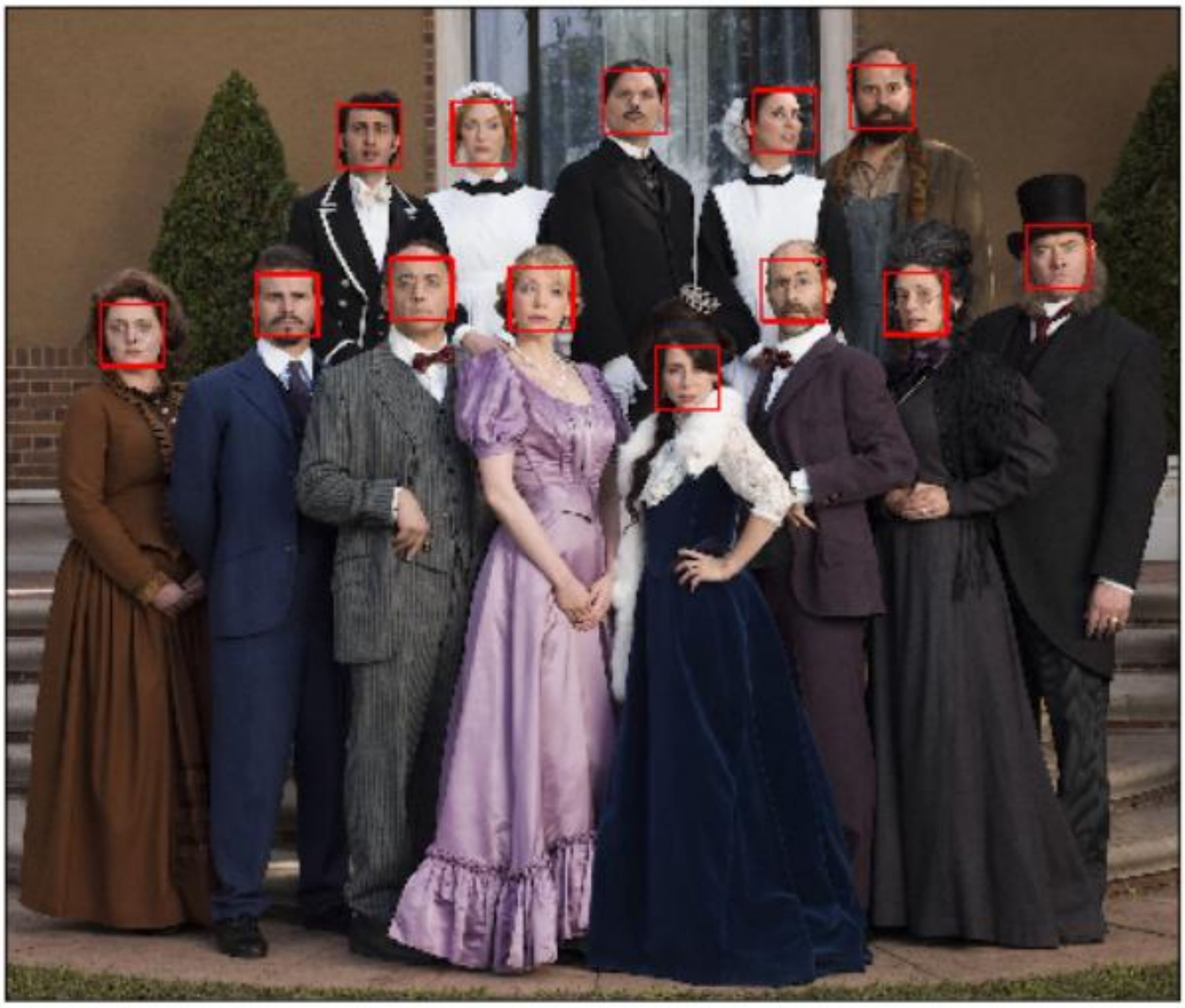
```
# Get the bounding box for each detected face
for (x,y,w,h) in faces:
    # Add a red bounding box to the detections image
    cv2.rectangle(image_with_detections, (x,y), (x+w,y+h), (255,0,0), 3)


# Display the image with the detections
fig = plt.figure(figsize = (8,8))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])

ax1.set_title('Image with Face Detections')
ax1.imshow(image_with_detections)
```

```
Number of faces detected: 13
```

Out[3]:

```
<matplotlib.image.AxesImage at 0x21318492c50>
```

Image with Face Detections



In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

Step 1: Add Eye Detections

There are other pre-trained detectors available that use a Haar Cascade Classifier - including full human body detectors, license plate detectors, and more. A full list of the pre-trained architectures can be found here.
To test your eye detector, we'll first read in a new test image with just a single face.

In [4]:

```
# Load in color image for face detection
image = cv2.imread('images/james.jpg')

# Convert the image to RGB colorspace
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Plot the RGB image
fig = plt.figure(figsize = (6,6))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])

ax1.set_title('Original Image')
ax1.imshow(image)
```

Out[4]:

```
<matplotlib.image.AxesImage at 0x21317474a20>
```

Notice that even though the image is a black and white image, we have read it in as a color image and so it will still need to be converted to grayscale in order to perform the most accurate face detection.

So, the next steps will be to convert this image to grayscale, then load OpenCV's face detector and run it with parameters that detect this face accurately.

In [5]:

```
# Convert the RGB  image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

# Extract the pre-trained face detector from an xml file
face_cascade =
cv2.CascadeClassifier('detector_architectures/haarcascade_frontalface_default.xm
l')

# Detect the faces in image
faces = face_cascade.detectMultiScale(gray, 1.25, 6)

# Print the number of faces detected in the image
print('Number of faces detected:', len(faces))

# Make a copy of the orginal image to draw face detections on
image_with_detections = np.copy(image)
```

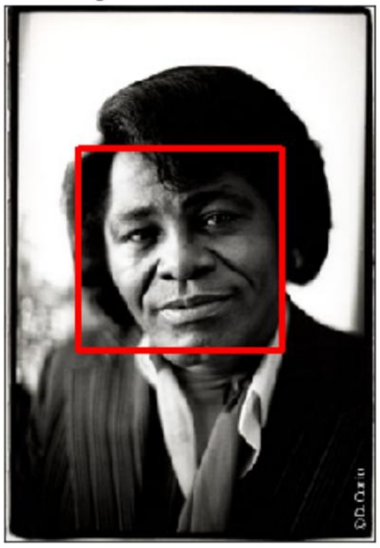
```
# Get the bounding box for each detected face
for (x,y,w,h) in faces:
    # Add a red bounding box to the detections image
    cv2.rectangle(image_with_detections, (x,y), (x+w,y+h), (255,0,0), 3)


# Display the image with the detections
fig = plt.figure(figsize = (6,6))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])

ax1.set_title('Image with Face Detection')
ax1.imshow(image_with_detections)
```

```
Number of faces detected: 1
```

Out[5]:

```
<matplotlib.image.AxesImage at 0x213177e0160>
```

Image with Face Detection



(IMPLEMENTATION) Add an eye detector to the current face detection setup.

A Haar-cascade eye detector can be included in the same way that the face detector was and, in this first task, it will be your job to do just this.
To set up an eye detector, use the stored parameters of the eye cascade detector, called `haarcascade_eye.xml`, located in the `detector_architectures` subdirectory. In the next code cell, create your eye detector and store its detections.

A few notes before you get started:

First, make sure to give your loaded eye detector the variable name

```
eye_cascade
```

and give the list of eye regions you detect the variable name

```
eyes
```

Second, since we've already run the face detector over this image, you should only search for eyes *within the rectangular face regions detected in `faces`*. This will minimize false detections.

Lastly, once you've run your eye detector over the facial detection region, you should display the RGB image with both the face detection boxes (in red) and your eye detections (in green) to verify that everything works as expected.

In [6]:

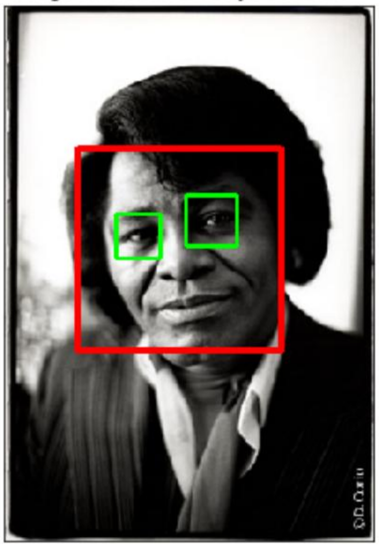
```
# Make a copy of the original image to plot rectangle detections
image_with_detections = np.copy(image)


# Loop over the detections and draw their corresponding face detection boxes
```

```
for (x,y,w,h) in faces:
    cv2.rectangle(image_with_detections, (x,y), (x+w,y+h),(255,0,0), 3)

# Do not change the code above this comment!


## TODO: Add eye detection, using haarcascade_eye.xml, to the current face
detector algorithm
## TODO: Loop over the eye detections and draw their corresponding boxes in
green on image_with_detections

# Extract the pre-trained eye detector from an xml file
eye_cascade =
cv2.CascadeClassifier('detector_architectures/haarcascade_eye.xml')

# Detect the eyes in image
eyes = eye_cascade.detectMultiScale(gray, 1.25, 6)

for (x,y,w,h) in faces:
    cv2.rectangle(image_with_detections,(x,y),(x+w,y+h),(255,0,0),2)
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = image_with_detections[y:y+h, x:x+w]
    eyes = eye_cascade.detectMultiScale(roi_gray)
    for (x1,y1,w1,h1) in eyes:
        cv2.rectangle(roi_color,(x1,y1),(x1+w1,y1+h1),(0,255,0),2)


# Plot the image with both faces and eyes detected
fig = plt.figure(figsize = (6,6))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])

ax1.set_title('Image with Face and Eye Detection')
ax1.imshow(image_with_detections)
```

Out[6]:

```
<matplotlib.image.AxesImage at 0x21317b49be0>
```

Image with Face and Eye Detection



(Optional) Add face and eye detection to your laptop camera

It's time to kick it up a notch, and add face and eye detection to your laptop's camera! Afterwards, you'll be able to show off your creation like in the gif shown below - made with a completed version of the code!

Notice that not all of the detections here are perfect - and your result need not be perfect either. You should spend a small amount of time tuning the parameters of your detectors to get reasonable results, but don't hold out for perfection. If we wanted perfection we'd need to spend a ton of time tuning the parameters of each detector, cleaning up the input image frames, etc. You can think of this as more of a rapid prototype.

The next cell contains code for a wrapper function called `laptop_camera_face_eye_detector` that, when called, will activate your laptop's camera. You will place the relevant face and eye detection code in this wrapper function to implement face/eye detection and mark those detections on each image frame that your camera captures.

Before adding anything to the function, you can run it to get an idea of how it works - a small window should pop up showing you the live feed from your camera; you can press any key to close this window.

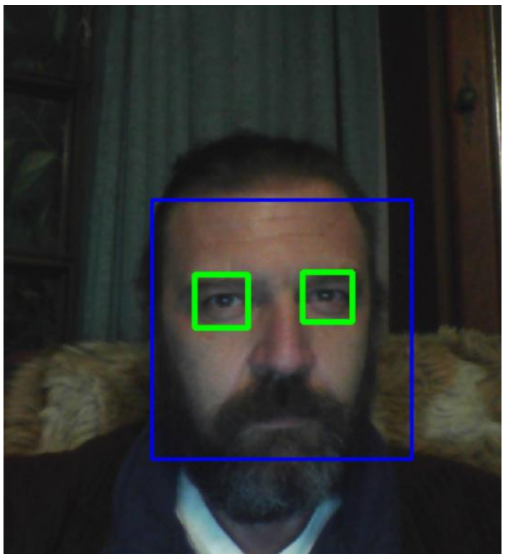
Note: Mac users may find that activating this function kills the kernel of their notebook every once in a while. If this happens to you, just restart your notebook's kernel, activate cell(s) containing any crucial import statements, and you'll be good to go!

In [7]:

```
### Add face and eye detection to this laptop camera function
# Make sure to draw out all faces/eyes found in each frame on the shown video
feed

import cv2
import time

# wrapper function for face/eye detection with your laptop camera
def laptop_camera_go():
    # Create instance of video capturer
    cv2.namedWindow("face detection activated")
    vc = cv2.VideoCapture(0)

    face_cascade =
cv2.CascadeClassifier("detector_architectures/haarcascade_frontalface_default.xm
l")
    eye_cascade =
cv2.CascadeClassifier("detector_architectures/haarcascade_eye.xml")

    # Try to get the first frame
    if vc.isOpened():
        rval, frame = vc.read()
    else:
        rval = False
    # Keep the video stream open
    while rval:

        gray = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
        faces = face_cascade.detectMultiScale(gray, 1.25, 6)

        for (x, y, w, h) in faces:
            cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)
        for (x, y, w, h) in faces:
            face = gray[y:y+h, x:x+w]
            eyes = eye_cascade.detectMultiScale(face)
            for (x1, y1, w1, h1) in eyes:
                cv2.rectangle(frame, (x1 + x, y1 + y), (x1+w1+x, y1+h1+y),
(0,255,0), 3)

        # Plot the image from camera with all the face and eye detections marked
        cv2.imshow("face detection activated", frame)

        # Exit functionality - press any key to exit laptop video
        key = cv2.waitKey(20)
        if key > 0: # Exit by pressing any key
            # Destroy windows
            cv2.destroyAllWindows()

            # Make sure window closes on OSx
            for i in range (1,5):
                cv2.waitKey(1)
            return

        # Read next frame
        time.sleep(0.05)
# control framerate for computation - default 20 frames per sec
        rval, frame = vc.read()
```

In [8]:

```
# Call the laptop camera face/eye detector function above
laptop_camera_go()
```



Step 2: De-noise an Image for Better Face Detection

Image quality is an important aspect of any computer vision task. Typically, when creating a set of images to train a deep learning network, significant care is taken to ensure that training images are free of visual noise or artifacts that hinder object detection. While computer vision algorithms - like a face detector - are typically trained on 'nice' data such as this, new test data doesn't always look so nice!

When applying a trained computer vision algorithm to a new piece of test data one often cleans it up first before feeding it in. This sort of cleaning - referred to as *pre-processing* - can include a number of cleaning phases like blurring, de-noising, color transformations, etc., and many of these tasks can be accomplished using OpenCV.

In this short subsection we explore OpenCV's noise-removal functionality to see how we can clean up a noisy image, which we then feed into our trained face detector.

Create a noisy image to work with

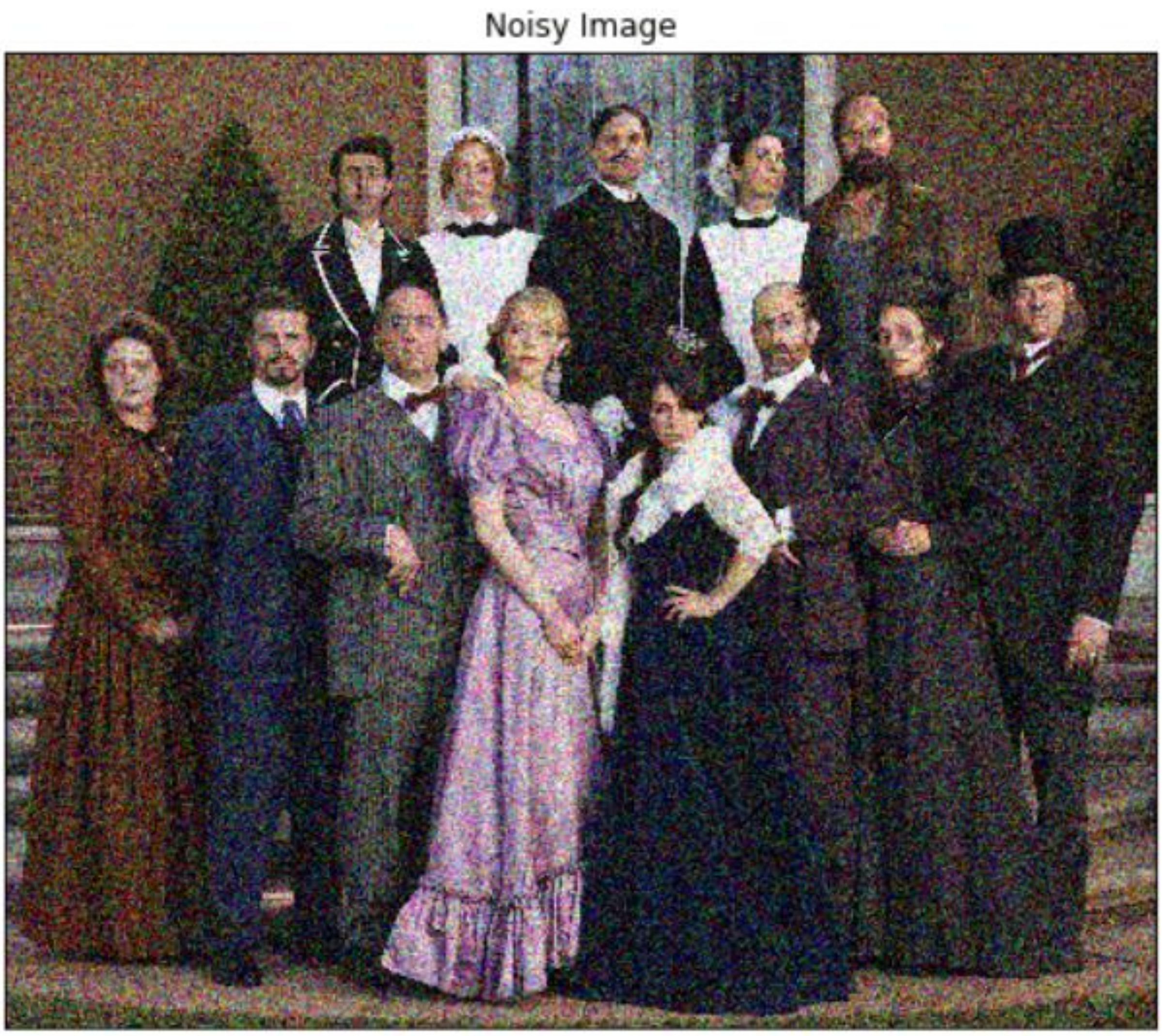
In the next cell, we create an artificial noisy version of the previous multi-face image. This is a little exaggerated - we don't typically get images that are this noisy - but image noise, or 'grainy-ness' in a digitial image - is a fairly common phenomenon.

In [9]:

```
# Load in the multi-face test image again
image = cv2.imread('images/test_image_1.jpg')

# Convert the image copy to RGB colorspace
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Make an array copy of this image
image_with_noise = np.asarray(image)

# Create noise - here we add noise sampled randomly from a Gaussian
distribution: a common model for noise
noise_level = 40
noise =
np.random.randn(image.shape[0],image.shape[1],image.shape[2])*noise_level

# Add this noise to the array image copy
image_with_noise = image_with_noise + noise

# Convert back to uint8 format
image_with_noise = np.asarray([np.uint8(np.clip(i,0,255)) for i in
image_with_noise])

# Plot our noisy image!
fig = plt.figure(figsize = (8,8))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])

ax1.set_title('Noisy Image')
ax1.imshow(image_with_noise)
```

Out[9]:

```
<matplotlib.image.AxesImage at 0x213188732b0>
```



In the context of face detection, the problem with an image like this is that - due to noise - we may miss some faces or get false detections.

In the next cell we apply the same trained OpenCV detector with the same settings as before, to see what sort of detections we get.

In [10]:

```
# Convert the RGB  image to grayscale
gray_noise = cv2.cvtColor(image_with_noise, cv2.COLOR_RGB2GRAY)

# Extract the pre-trained face detector from an xml file
face_cascade =
cv2.CascadeClassifier('detector_architectures/haarcascade_frontalface_default.xm
l')

# Detect the faces in image
faces = face_cascade.detectMultiScale(gray_noise, 4, 6)

# Print the number of faces detected in the image
print('Number of faces detected:', len(faces))

# Make a copy of the orginal image to draw face detections on
image_with_detections = np.copy(image_with_noise)

# Get the bounding box for each detected face
for (x,y,w,h) in faces:
    # Add a red bounding box to the detections image
    cv2.rectangle(image_with_detections, (x,y), (x+w,y+h), (255,0,0), 3)
```

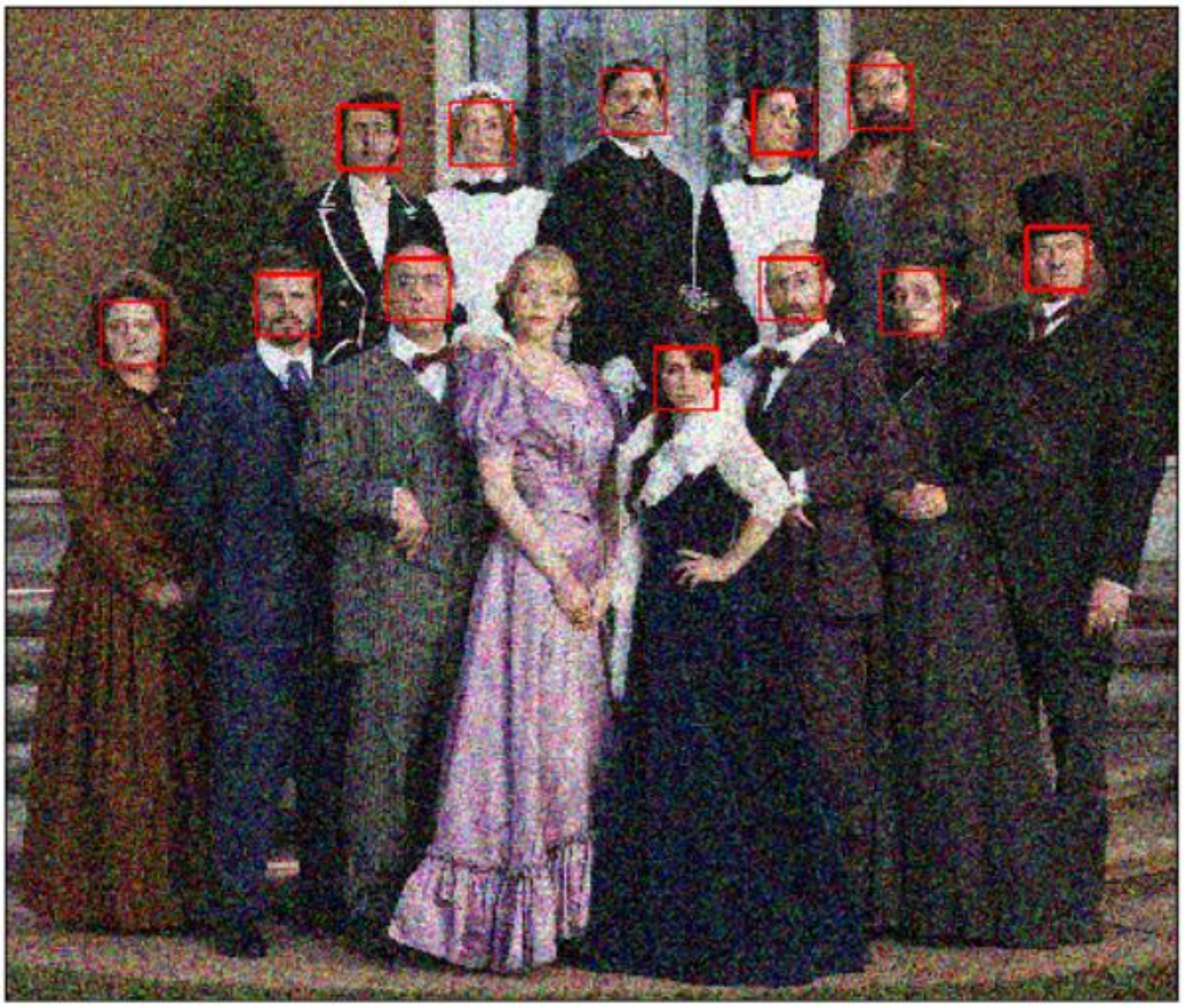
```
# Display the image with the detections
fig = plt.figure(figsize = (8,8))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])

ax1.set_title('Noisy Image with Face Detections')
ax1.imshow(image_with_detections)
```

```
Number of faces detected: 12
```

Out[10]:

```
<matplotlib.image.AxesImage at 0x21317768b38>
```

Noisy Image with Face Detections



With this added noise we now miss one of the faces! Even two of them :))

(IMPLEMENTATION) De-noise this image for better face detection

Time to get your hands dirty: using OpenCV's built in color image de-noising functionality called `fastNlMeansDenoisingColored` - de-noise this image enough so that all the faces in the image are properly detected. Once you have cleaned the image in the next cell, use the cell that follows to run our trained face detector over the cleaned image to check out its detections.

You can find its official documentation here and a useful example here.

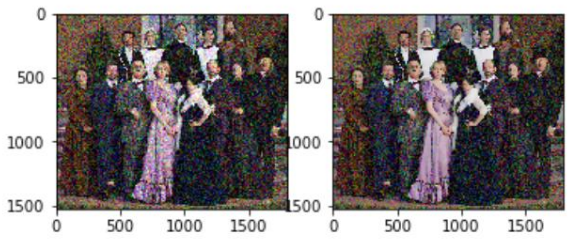
Note: you can keep all parameters *except* `photo_render` fixed as shown in the second link above. Play around with the value of this parameter - see how it affects the resulting cleaned image.

In [11]:

```
## TODO: Use OpenCV's built in color image de-noising function to clean up our
noisy image!

denoised_image =
cv2.fastNlMeansDenoisingColored(image_with_noise,None,13,10,7,21)
```

In [12]:

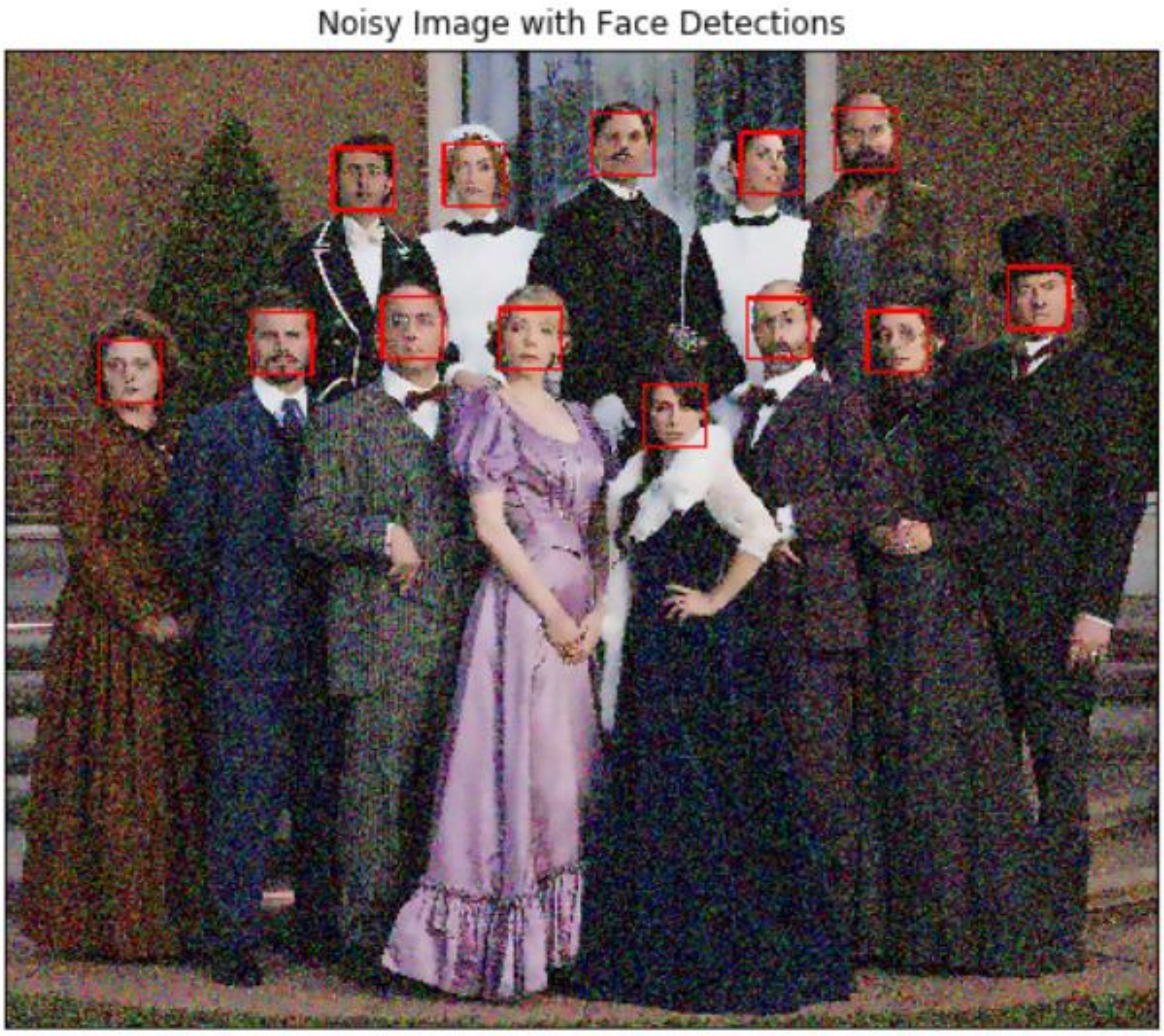
```
## TODO: Run the face detector on the de-noised image to improve your detections
and display the result
plt.subplot(121),plt.imshow(image_with_noise)
plt.subplot(122),plt.imshow(denoised_image)
plt.show()

# Convert the RGB  image to grayscale
gray_noise = cv2.cvtColor(denoised_image, cv2.COLOR_RGB2GRAY)

# Extract the pre-trained face detector from an xml file
face_cascade =
cv2.CascadeClassifier('detector_architectures/haarcascade_frontalface_default.xm
l')

# Detect the faces in image
faces = face_cascade.detectMultiScale(gray_noise, 4, 4)

# Print the number of faces detected in the image
print('Number of faces detected:', len(faces))

# Make a copy of the orginal image to draw face detections on
image_with_detections = np.copy(denoised_image)

# Get the bounding box for each detected face
for (x,y,w,h) in faces:
    # Add a red bounding box to the detections image
    cv2.rectangle(image_with_detections, (x,y), (x+w,y+h), (255,0,0), 3)


# Display the image with the detections
fig = plt.figure(figsize = (8,8))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])

ax1.set_title('Noisy Image with Face Detections')
ax1.imshow(image_with_detections)
```



```
Number of faces detected: 13
```

Out[12]:

```
<matplotlib.image.AxesImage at 0x2131c867da0>
```



Step 3: Blur an Image and Perform Edge Detection

Now that we have developed a simple pipeline for detecting faces using OpenCV - let's start playing around with a few fun things we can do with all those detected faces!

Importance of Blur in Edge Detection

Edge detection is a concept that pops up almost everywhere in computer vision applications, as edge-based features (as well as features built on top of edges) are often some of the best features for e.g., object detection and recognition problems.

Edge detection is a dimension reduction technique - by keeping only the edges of an image we get to throw away a lot of non-discriminating information. And typically the most useful kind of edge-detection is one that preserves only the important, global structures (ignoring local structures that aren't very discriminative). So removing local structures / retaining global structures is a crucial pre-processing step to performing edge detection in an image, and blurring can do just that.

Below is an animated gif showing the result of an edge-detected cat taken from Wikipedia, where the image is gradually blurred more and more prior to edge detection. When the animation begins you can't quite make out what it's a picture of, but as the animation evolves and local structures are removed via blurring the cat becomes visible in the edge-detected image.

Edge detection is a **convolution** performed on the image itself, and you can read about Canny edge detection on this OpenCV documentation page.

Canny edge detection

In the cell below we load in a test image, then apply *Canny edge detection* on it. The original image is shown on the left panel of the figure, while the edge-detected version of the image is shown on the right. Notice how the result looks very busy - there are too many little details preserved in the image before it is sent to the edge detector. When applied in computer vision applications, edge detection should preserve *global* structure; doing away with local structures that don't help describe what objects are in the image.

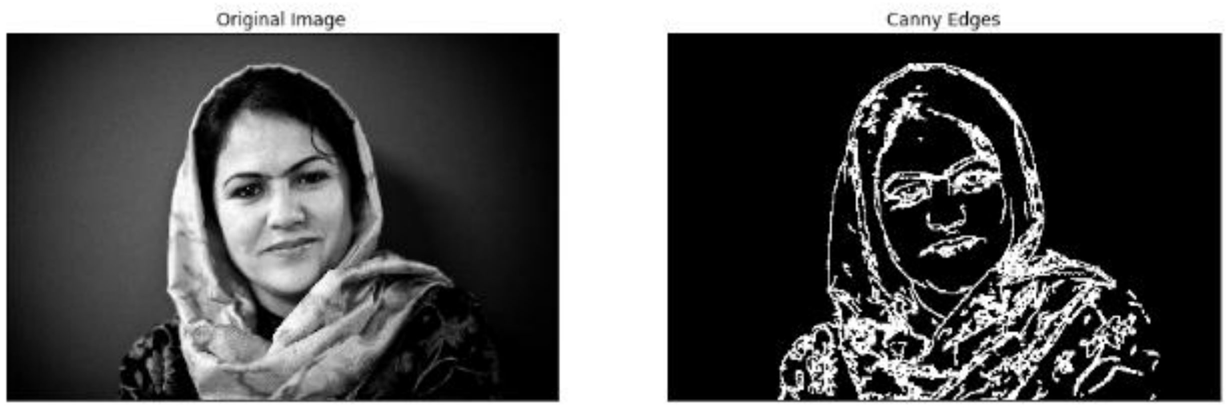
In [13]:

```
# Load in the image
image = cv2.imread('images/fawzia.jpg')
```

```
# Convert to RGB colorspace
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Convert to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

# Perform Canny edge detection
edges = cv2.Canny(gray,100,200)

# Dilate the image to amplify edges
edges = cv2.dilate(edges, None)

# Plot the RGB and edge-detected image
fig = plt.figure(figsize = (15,15))
ax1 = fig.add_subplot(121)
ax1.set_xticks([])
ax1.set_yticks([])

ax1.set_title('Original Image')
ax1.imshow(image)

ax2 = fig.add_subplot(122)
ax2.set_xticks([])
ax2.set_yticks([])

ax2.set_title('Canny Edges')
ax2.imshow(edges, cmap='gray')
```

Out[13]:

```
<matplotlib.image.AxesImage at 0x2131c260ba8>
```



Without first blurring the image, and removing small, local structures, a lot of irrelevant edge content gets picked up and amplified by the detector (as shown in the right panel above).

(IMPLEMENTATION) Blur the image *then* perform edge detection

In the next cell, you will repeat this experiment - blurring the image first to remove these local structures, so that only the important boudnary details remain in the edge-detected image.

Blur the image by using OpenCV's `filter2d` functionality - which is discussed in this documentation page - and use an *averaging kernel* of width equal to 4.

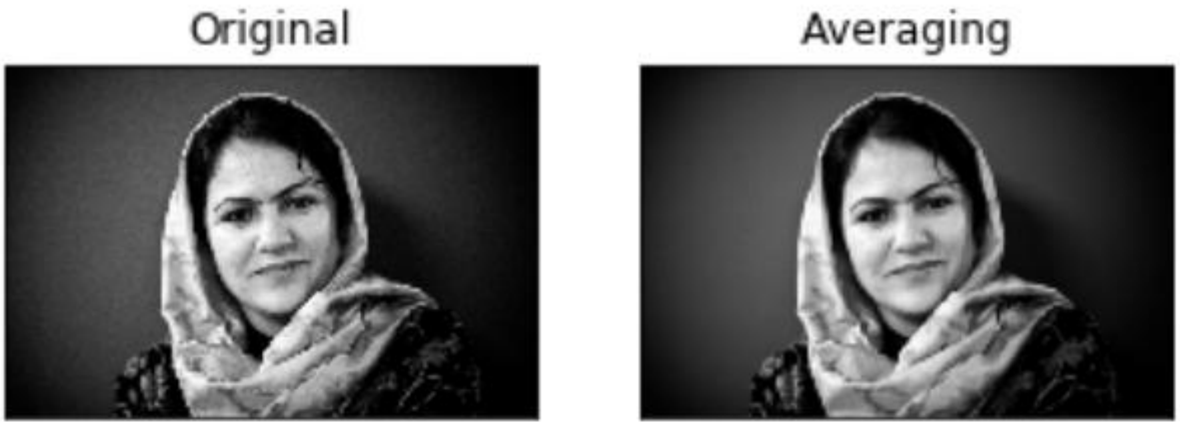
In [14]:

```
### TODO: Blur the test imageusing OpenCV's filter2d functionality,
# Use an averaging kernel, and a kernel width equal to 4
kernel = np.ones((4,4),np.float32)/16
```

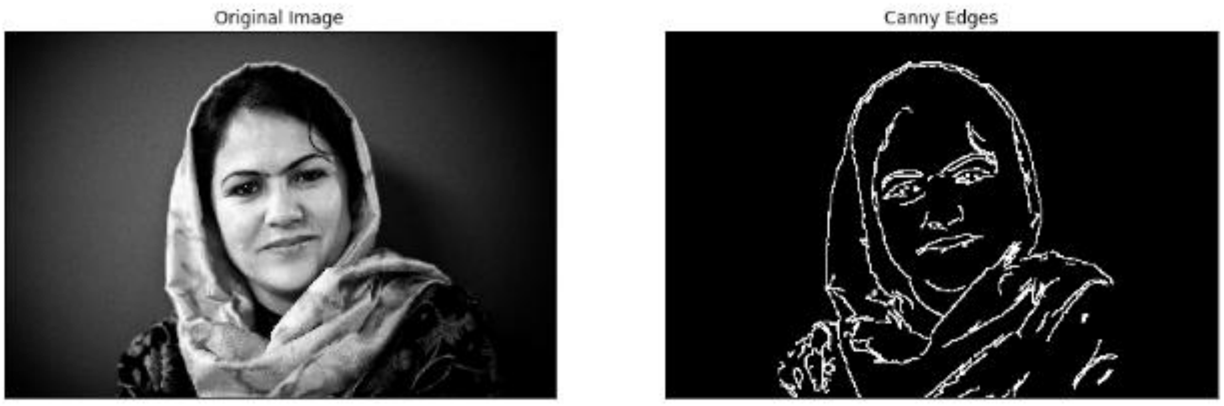
```
## TODO: Then perform Canny edge detection and display the output

dst = cv2.filter2D(image,-1,kernel)
plt.subplot(121),plt.imshow(image),plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow(dst),plt.title('Averaging')
plt.xticks([]), plt.yticks([])
plt.show()
## TODO: Then perform Canny edge detection and display the output
# Perform Canny edge detection
edges = cv2.Canny(dst,100,200)

# Dilate the image to amplify edges
edges = cv2.dilate(edges, None)

# Plot the RGB and edge-detected image
fig = plt.figure(figsize = (15,15))
ax1 = fig.add_subplot(121)
ax1.set_xticks([])
ax1.set_yticks([])

ax1.set_title('Original Image')
ax1.imshow(image)

ax2 = fig.add_subplot(122)
ax2.set_xticks([])
ax2.set_yticks([])

ax2.set_title('Canny Edges')
ax2.imshow(edges, cmap='gray')
```

Out[14]:

```
<matplotlib.image.AxesImage at 0x2131c6f4160>
```



```
<matplotlib.image.AxesImage at 0x2131c6f4160>
```



Step 4: Automatically Hide the Identity of an Individual

If you film something like a documentary or reality TV, you must get permission from every individual shown on film before you can show their face, otherwise you need to blur it out - by blurring the face a lot (so much so that even the global structures are obscured)! This is also true for projects like Google's StreetView maps - an enormous collection of mapping images taken from a fleet of Google vehicles. Because it would be impossible for Google to get the permission of every single person accidentally captured in one of these images they blur out everyone's faces, the detected images must automatically blur the identity of detected people. Here's a few examples of folks caught in the camera of a Google street view vehicle.

Read in an image to perform identity detection

Let's try this out for ourselves. Use the face detection pipeline built above and what you know about using the `filter2D` to blur and image, and use these in tandem to hide the identity of the person in the following image - loaded in and printed in the next cell.

In [15]:

```
# Load in the image
image = cv2.imread('images/gus.jpg')

# Convert the image to RGB colorspace
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Display the image
fig = plt.figure(figsize = (6,6))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])
ax1.set_title('Original Image')
ax1.imshow(image)
```

Out[15]:

```
<matplotlib.image.AxesImage at 0x2131c04f748>
```

(IMPLEMENTATION) Use blurring to hide the identity of an individual in an image

The idea here is to 1) automatically detect the face in this image, and then 2) blur it out! Make sure to adjust the parameters of the *averaging* blur filter to completely obscure this person's identity.

In [16]:

```
## TODO: Implement face detection
# Convert the RGB  image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

# Extract the pre-trained face detector from an xml file
face_cascade =
cv2.CascadeClassifier('detector_architectures/haarcascade_frontalface_default.xm
l')

# Detect the faces in image
faces = face_cascade.detectMultiScale(gray, 1.2, 5)

# Print the number of faces detected in the image
print('Number of faces detected:', len(faces))

# Make a copy of the orginal image to draw face detections on
image_with_detections = np.copy(image)

# Get the bounding box for each detected face
for (x,y,w,h) in faces:
    # Add a red bounding box to the detections image
    cv2.rectangle(image_with_detections, (x,y), (x+w,y+h), (255,0,0), 3)

## TODO: Blur the bounding box around each detected face using an averaging
filter and display the result

# Make a copy of the orginal image to draw face detections on
image_with_blur = np.copy(image)

for (x, y, w, h) in faces:
    roi = image_with_blur[y:y+h, x:x+w]
    roi = cv2.GaussianBlur(roi, (111, 111),50,50)
    image_with_blur[y:y+h, x:x+w] = roi
```

```
# Display the image with the detections
fig = plt.figure(figsize = (6,6))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])

ax1.set_title('Image with Face Detection')
ax1.imshow(image_with_detections)

# Display the image with the blur
fig = plt.figure(figsize = (6,6))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])

ax1.set_title('Image with Face Blurred')
ax1.imshow(image_with_blur)
```

```
Number of faces detected: 1
```

Out[16]:

```
<matplotlib.image.AxesImage at 0x2131c0f3940>
```

Image with Face Detection

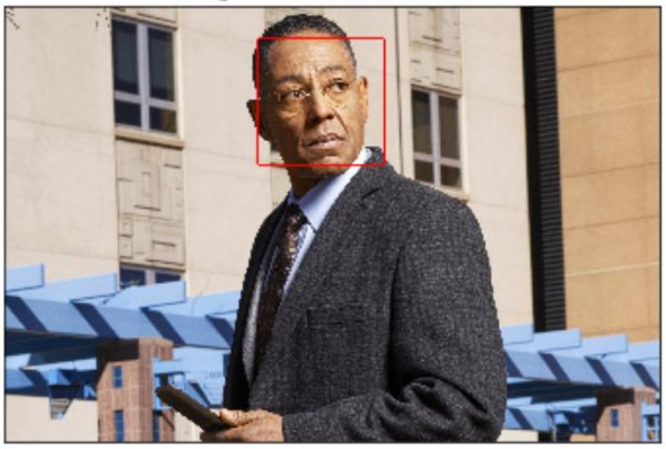
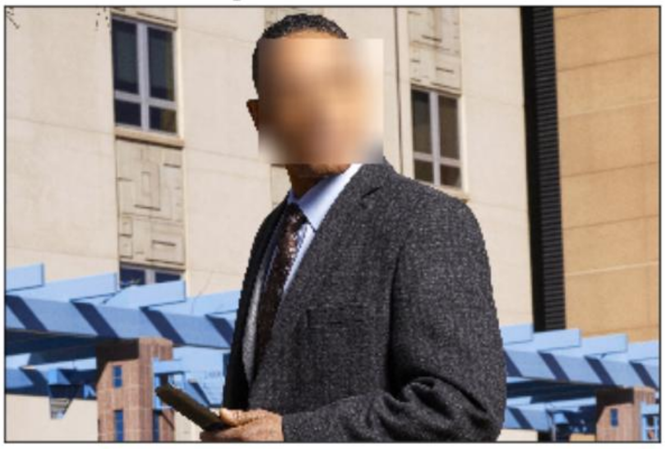


Image with Face Blurred



(Optional) Build identity protection into your laptop camera

In this optional task you can add identity protection to your laptop camera, using the previously completed code where you added face detection to your laptop camera - and the task above. You should be able to get reasonable results with little parameter tuning - like the one shown in the gif below.

As with the previous video task, to make this perfect would require significant effort - so don't strive for perfection here, strive for reasonable quality.

The next cell contains code a wrapper function called `laptop_camera_identity_hider` that - when called - will activate your laptop's camera. You need to place the relevant face detection and blurring code developed above in this function in order to blur faces entering your laptop camera's field of view.

Before adding anything to the function you can call it to get a hang of how it works - a small window will pop up showing you the live feed from your camera, you can press any key to close this window.

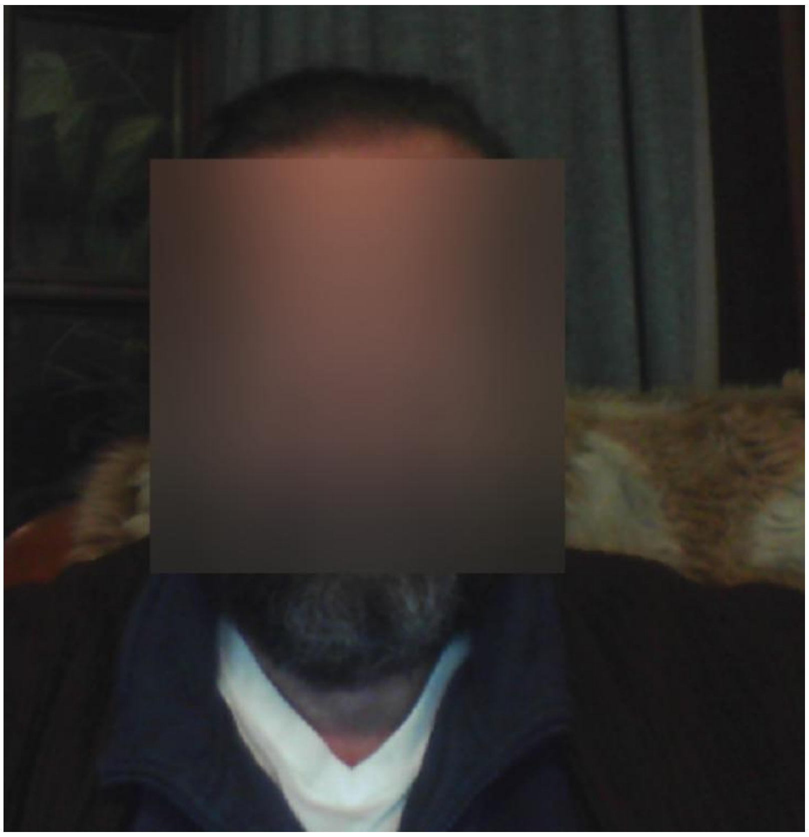
Note: Mac users may find that activating this function kills the kernel of their notebook every once in a while. If this happens to you, just restart your notebook's kernel, activate cell(s) containing any crucial import statements, and you'll be good to go!

In [17]:

```
### Insert face detection and blurring code into the wrapper below to create an
identity protector on your laptop!
import cv2
import time

def laptop_camera_go():
    # Create instance of video capturer
    cv2.namedWindow("face detection activated")
    vc = cv2.VideoCapture(0)
    face_cascade =
cv2.CascadeClassifier("detector_architectures/haarcascade_frontalface_default.xm
l")

    # Try to get the first frame
    if vc.isOpened():
        rval, frame = vc.read()
    else:
        rval = False

    # Keep video stream open
    while rval:
        # Plot image from camera with detections marked
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(gray, 1.25, 6)
        for (x, y, w, h) in faces:
            roi = frame[y:y+h, x:x+w]
            roi = cv2.GaussianBlur(roi, (111, 111), 50, 50)
            frame[y:y+h, x:x+w] = roi

        cv2.imshow("face detection activated", frame)

        # Exit functionality - press any key to exit laptop video
        key = cv2.waitKey(20)
        if key > 0: # Exit by pressing any key
            # Destroy windows
            cv2.destroyAllWindows()

            for i in range (1,5):
                cv2.waitKey(1)
            return

        # Read next frame
        time.sleep(0.05)             # control framerate for computation -
default 20 frames per sec
        rval, frame = vc.read()
```

In [18]:

```python
# Run laptop identity hider
laptop_camera_go()
```



Step 5: Create a CNN to Recognize Facial Keypoints

OpenCV is often used in practice with other machine learning and deep learning libraries to produce interesting results. In this stage of the project you will create your own end-to-end pipeline - employing convolutional networks in keras along with OpenCV - to apply a "selfie" filter to streaming video and images.

You will start by creating and then training a convolutional network that can detect facial keypoints in a small dataset of cropped images of human faces. We then guide you towards OpenCV to expanding your detection algorithm to more general images. What are facial keypoints? Let's take a look at some examples.

Facial keypoints (also called facial landmarks) are the small blue-green dots shown on each of the faces in the image above - there are 15 keypoints marked in each image. They mark important areas of the face - the eyes, corners of the mouth, the nose, etc. Facial keypoints can be used in a variety of machine learning applications from face and emotion recognition to commercial applications like the image filters popularized by Snapchat.

Below we illustrate a filter that, using the results of this section, automatically places sunglasses on people in images (using the facial keypoints to place the glasses correctly on each face). Here, the facial keypoints have been colored lime green for visualization purposes.

Make a facial keypoint detector

But first things first: how can we make a facial keypoint detector? Well, at a high level, notice that facial keypoint detection is a *regression problem*. A single face corresponds to a set of 15 facial keypoints (a set of 15 corresponding (x,y)(x,y) coordinates, i.e., an output point). Because our input data are images, we can employ a *convolutional neural network* to recognize patterns in our images and learn how to identify these keypoint given sets of labeled data.
In order to train a regressor, we need a training set - a set of facial image / facial keypoint pairs to train on. For this we will be using [this dataset from Kaggle](). We've already downloaded this data and placed it in the `data` directory. Make sure that you have both the *training* and *test* data files. The training dataset contains several thousand $96\times96$96×96 grayscale images of cropped human faces, along with each face's 15 corresponding facial keypoints (also called landmarks) that have been placed by hand, and recorded in (x,y)(x,y) coordinates. This wonderful resource also has a substantial testing set, which we will use in tinkering with our convolutional network.

To load in this data, run the Python cell below - notice we will load in both the training and testing sets.

The `load_data` function is in the included `utils.py` file.

In [19]:

```python
from utils import *

# Load training set
X_train, y_train = load_data()
print("X_train.shape == {}".format(X_train.shape))
print("y_train.shape == {}; y_train.min == {:.3f}; y_train.max ==
{:.3f}".format(
    y_train.shape, y_train.min(), y_train.max()))

# Load testing set
X_test, _ = load_data(test=True)
print("X_test.shape == {}".format(X_test.shape))
```

```
Using TensorFlow backend.
X_train.shape == (2140, 96, 96, 1)
y_train.shape == (2140, 30); y_train.min == -0.920; y_train.max == 0.996
X_test.shape == (1783, 96, 96, 1)
```

The `load_data` function in `utils.py` originates from this excellent blog post, which you are *strongly* encouraged to read. Please take the time now to review this function. Note how the output values - that is, the coordinates of each set of facial landmarks - have been normalized to take on values in the range $[-1,1]$[−1,1], while the pixel values of each input point (a facial image) have been normalized to the range $[0,1]$[0,1].

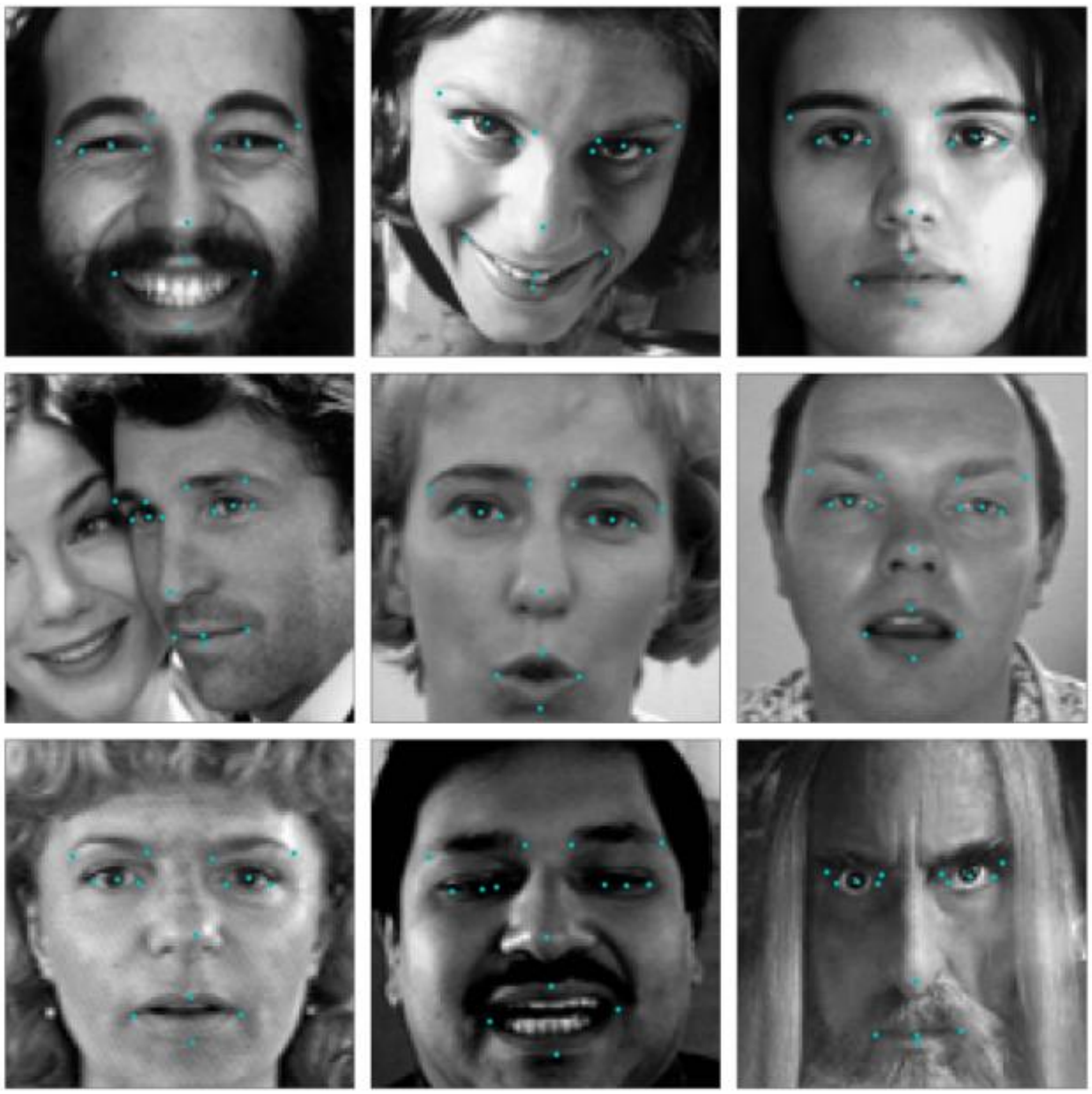
Note: the original Kaggle dataset contains some images with several missing keypoints. For simplicity, the `load_data` function removes those images with missing labels from the dataset. As an ***optional*** extension, you are welcome to amend the `load_data` function to include the incomplete data points.

Visualize the Training Data

Execute the code cell below to visualize a subset of the training data.

In [20]:

```
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(20,20))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)
for i in range(9):
    ax = fig.add_subplot(3, 3, i + 1, xticks=[], yticks=[])
    plot_data(X_train[i], y_train[i], ax)
```



For each training image, there are two landmarks per eyebrow (**four** total), three per eye (**six** total), **four** for the mouth, and **one** for the tip of the nose.

Review the `plot_data` function in `utils.py` to understand how the 30-dimensional training labels in `y_train` are mapped to facial locations, as this function will prove useful for your pipeline.

(IMPLEMENTATION) Specify the CNN Architecture

In this section, you will specify a neural network for predicting the locations of facial keypoints. Use the code cell below to specify the architecture of your neural network. We have imported some layers that you may find useful for this task, but if you need to use more Keras layers, feel free to import them in the cell.
Your network should accept a 96×9696×96 grayscale image as input, and it should output a vector with 30 entries, corresponding to the predicted (horizontal and vertical) locations of 15 facial keypoints. If you are not sure where to start, you can find some useful starting architectures in [this blog](), but you are not permitted to copy any of the architectures that you find online.

In [21]:

```
# Import deep learning resources from Keras
# Import deep learning resources from Keras
from keras.models import Sequential
from keras.layers import Convolution2D, MaxPooling2D, Dropout,
GlobalAveragePooling2D
from keras.layers import Flatten, Dense, Activation
from keras.layers.normalization import BatchNormalization
from keras import regularizers

## TODO: Specify a CNN architecture
# Your model should accept 96x96 pixel graysale images in
# It should have a fully-connected output layer with 30 values (2 for each
facial keypoint)

input_shape=(96,96,1)

model_final = Sequential()
model_final.add(Convolution2D(48, kernel_size=2, padding="same",

kernel_regularizer=regularizers.l2(0.01),input_shape=input_shape))
model_final.add(BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001,
                                   center=True, scale=True,
beta_initializer='zeros',
                                   gamma_initializer='ones',
moving_mean_initializer='zeros',
                                   moving_variance_initializer='ones',
                                   beta_regularizer=None,
gamma_regularizer=None,
                                   beta_constraint=None, gamma_constraint=None))
model_final.add(Activation('tanh'))
model_final.add(MaxPooling2D(pool_size=2))
model_final.add(Dropout(0.2))
model_final.add(Convolution2D(96, kernel_size=5, padding="same"))
model_final.add(MaxPooling2D(pool_size=2))
model_final.add(Dropout(0.2))
model_final.add(Convolution2D(192, kernel_size=3, padding="same"))
model_final.add(MaxPooling2D(pool_size=2))

model_final.add(Flatten())
model_final.add(Dropout(0.2))
model_final.add(Dense(300))
model_final.add(Dropout(0.2))

model_final.add(Dense(30))

model_final.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 96, 96, 48)        240
_________________________________________________________________
batch_normalization_1 (Batch (None, 96, 96, 48)        192
_________________________________________________________________
activation_1 (Activation)    (None, 96, 96, 48)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 48, 48, 48)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 48, 48, 48)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 48, 48, 96)        115296
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 24, 24, 96)        0
_________________________________________________________________
dropout_2 (Dropout)          (None, 24, 24, 96)        0

_________________________________________________________________
conv2d_3 (Conv2D)            (None, 24, 24, 192)       166080

_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 12, 12, 192)       0

_________________________________________________________________
flatten_1 (Flatten)          (None, 27648)             0

_________________________________________________________________
dropout_3 (Dropout)          (None, 27648)             0

_________________________________________________________________
dense_1 (Dense)              (None, 300)               8294700

_________________________________________________________________
dropout_4 (Dropout)          (None, 300)               0

_________________________________________________________________
dense_2 (Dense)              (None, 30)                9030
=================================================================
Total params: 8,585,538.0
Trainable params: 8,585,442.0
Non-trainable params: 96.0

_________________________________________________________________
```

Step 6: Compile and Train the Model

After specifying your architecture, you'll need to compile and train the model to detect facial keypoints'

(IMPLEMENTATION) Compile and Train the Model

Use the `compile` method to configure the learning process. Experiment with your choice of optimizer; you may have some ideas about which will work best (`SGD` vs. `RMSprop`, etc), but take the time to empirically verify your theories.

Use the `fit` method to train the model. Break off a validation set by setting `validation_split=0.2`. Save the returned `History` object in the `history` variable.

Your model is required to attain a validation loss (measured as mean squared error) of at least **XYZ**. When you have finished training, save your model as an HDF5 file with file path `my_model.h5`.

In [22]:

```
from keras.models import load_model

model_final_cont_rmsprop =
load_model('./model_start_48_ker_253_dr_2222_fl_300_rmsprop_1regtanh_pre1tanh_conv_batchnorm_1_m99_batchsize_16_l1_0001_d_000001_75.h5')
model_final_cont_rmsprop.load_weights('./model_start_48_ker_253_dr_2222_fl_300_rmsprop_1regtanh_pre1tanh_conv_batchnorm_1_m99_batchsize_16_l1_0001_d_000001_75.h5')
```

I spent all AWS credit and have to continue with my notebook, but it is too difficult for it. So I have to calculate no more then 20-30 epochs per time and then reload the model and weights and continue...Too tedious, but my comp just freezes and I have to restart to continue with the result obtained without starting the next cell again.

In [ ]:

```
from keras.optimizers import SGD, RMSprop, Adagrad, Adadelta, Adam, Adamax,
Nadam
from keras.callbacks import CSVLogger
import pandas as pd

## TODO: Compile the model
from keras.callbacks import ModelCheckpoint
```

```
checkpointer =
ModelCheckpoint(filepath='model_start_48_ker_253_dr_2222_fl_300_rmsprop_1regtanh
_pre1tanh_conv_batchnorm_1_m99_batchsize_16_l1_0001_d_000001_75+25.h5',
                               verbose=1, save_best_only=True)

rmsprop = RMSprop(lr=0.0001, rho=0.9, epsilon=1e-08, decay=0.000001)
model_final_cont_rmsprop.compile(optimizer=rmsprop, loss="mean_squared_error",
metrics=['accuracy'])

#csv_logger = CSVLogger('model_as_start_rmsprop_1reg_conv_butchsize_96.csv',
append=True,separator=';')

## TODO: Train the model
history = model_final_cont_rmsprop.fit(X_train, y_train, batch_size=16,
epochs=25, verbose=1,
                 validation_split=0.2, callbacks=[checkpointer])

pd.DataFrame(history.history).to_csv('model_start_48_ker_253_dr_2222_fl_300_rmsp
rop_1regtanh_pre1tanh_conv_batchnorm_1_m99_batchsize_16_l1_0001_d_000001_75+25.c
sv')

## TODO: Save the model as model.h5
model_final_cont_rmsprop.save('model_start_48_ker_253_dr_2222_fl_300_rmsprop_1re
gtanh_pre1tanh_conv_batchnorm_1_m99_batchsize_16_l1_0001_d_000001_75+25.h5')
```

```
Train on 1712 samples, validate on 428 samples
Epoch 1/25
1696/1712 [============================>.] - ETA: 1s - loss: 0.0018 - acc:
0.8084
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
Epoch 00000: val_loss improved from inf to 0.00186, saving model to
model_start_48_ker_253_dr_2222_fl_300_rmsprop_1regtanh_pre1tanh_conv_batchnorm_1
_m99_batchsize_16_l1_0001_d_000001_75+25.h5
1712/1712 [==============================] - 174s - loss: 0.0018 - acc: 0.8078 -
val_loss: 0.0019 - val_acc: 0.7804
Epoch 2/25
1696/1712 [============================>.] - ETA: 1s - loss: 0.0017 - acc:
0.8078
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
Epoch 00001: val_loss improved from 0.00186 to 0.00182, saving model to
model_start_48_ker_253_dr_2222_fl_300_rmsprop_1regtanh_pre1tanh_conv_batchnorm_1
_m99_batchsize_16_l1_0001_d_000001_75+25.h5
1712/1712 [==============================] - 179s - loss: 0.0017 - acc: 0.8084 -
val_loss: 0.0018 - val_acc: 0.8084
Epoch 3/25
1696/1712 [============================>.] - ETA: 1s - loss: 0.0016 - acc:
0.8019
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
Epoch 00002: val_loss improved from 0.00182 to 0.00179, saving model to
model_start_48_ker_253_dr_2222_fl_300_rmsprop_1regtanh_pre1tanh_conv_batchnorm_1
_m99_batchsize_16_l1_0001_d_000001_75+25.h5
```

```
1712/1712 [==============================] - 181s - loss: 0.0016 - acc: 0.8020 -
val_loss: 0.0018 - val_acc: 0.8107
Epoch 4/25
1696/1712 [============================>.] - ETA: 1s - loss: 0.0016 - acc:
0.8042
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
Epoch 00003: val_loss did not improve
1712/1712 [==============================] - 182s - loss: 0.0016 - acc: 0.8049 -
val_loss: 0.0018 - val_acc: 0.8037
Epoch 5/25
1696/1712 [============================>.] - ETA: 1s - loss: 0.0016 - acc:
0.8243
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
Epoch 00004: val_loss improved from 0.00179 to 0.00179, saving model to
model_start_48_ker_253_dr_2222_fl_300_rmsprop_1regtanh_pre1tanh_conv_batchnorm_1
_m99_batchsize_16_l1_0001_d_000001_75+25.h5
1712/1712 [==============================] - 185s - loss: 0.0016 - acc: 0.8236 -
val_loss: 0.0018 - val_acc: 0.7897
Epoch 6/25
1696/1712 [============================>.] - ETA: 1s - loss: 0.0015 - acc:
0.8149
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
Epoch 00005: val_loss did not improve
1712/1712 [==============================] - 185s - loss: 0.0015 - acc: 0.8143 -
val_loss: 0.0020 - val_acc: 0.7734
Epoch 7/25
1696/1712 [============================>.] - ETA: 1s - loss: 0.0015 - acc:
0.8325
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
Epoch 00006: val_loss improved from 0.00179 to 0.00168, saving model to
model_start_48_ker_253_dr_2222_fl_300_rmsprop_1regtanh_pre1tanh_conv_batchnorm_1
_m99_batchsize_16_l1_0001_d_000001_75+25.h5
1712/1712 [==============================] - 185s - loss: 0.0015 - acc: 0.8318 -
val_loss: 0.0017 - val_acc: 0.7734
Epoch 8/25
1696/1712 [============================>.] - ETA: 1s - loss: 0.0014 - acc:
0.8296
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
Epoch 00007: val_loss improved from 0.00168 to 0.00156, saving model to
model_start_48_ker_253_dr_2222_fl_300_rmsprop_1regtanh_pre1tanh_conv_batchnorm_1
_m99_batchsize_16_l1_0001_d_000001_75+25.h5
1712/1712 [==============================] - 185s - loss: 0.0014 - acc: 0.8306 -
val_loss: 0.0016 - val_acc: 0.7967
Epoch 9/25
1696/1712 [============================>.] - ETA: 1s - loss: 0.0014 - acc:
0.8261
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
Epoch 00024: val_loss did not improve
1712/1712 [==============================] - 188s - loss: 9.1771e-04 - acc:
0.8394 - val_loss: 0.0013 - val_acc: 0.8014
```

Step 7: Visualize the Loss and Test Predictions

(IMPLEMENTATION) Answer a few questions and visualize the loss

Question 1: Outline the steps you took to get to your final neural network architecture and your reasoning at each step.

Answer: I decided to start with the CNN architecture from our lesson **CNNs for Image Classification with Keras**. Here we created the model which start up with the three pairs Conv2D and MaxPool2D layers. I've made some changes:

1. doubled the number of filters for each convolutional layer from 32 through 128, as our input shape is three times bigger than in the lesson.
2. To make the start model more simple I remove the activation function at all, as values are normalized between -1 and 1, while ReLU has limits (0,1). I decided to check the influence of the presence of an activation function later.

But left the kerenel size 2, because it was noticed in this blog, that "2x2 filters, is again a pretty good regularizer by itself".

Besides, I noticed, that all our points of interest are situated enough far from the image borders, that is why I left **padding='same'** as well as **pool_size=2** for the MaxPool2D layers as in the lesson.

I continued with the Flatten layer and the two fully-connected layers with 512 nodes for the first one and 30 for the last according to the task.

Finally, after dozens of attempts to improve my model (no joke - I spent almost 90 USD for the AWS instance, that is why have to finish the project with my own laptop), I added four Dropout layers with the dropout rate from 0.2 through 0.5 and started with 50 epochs.

This model turned out to be strongly overfitted with more than 9M trainable params. The most amount was given by the Dense layer. Then I've decided to change my Flatten layer by the GlobalAveragePooling2D, which decreased the number of trainable params up to 15 times. And I even had to increase the number of nodes in the Convolutional layers up to 48 for the first one and doubled for the each next. The GAP decreased the accuracy significantly and I returned flatten layer back but left the number of nodes.

As it is noticed about filter shapes in this tutorial, the trick is thus to find the right level of “granularity” (i.e. filter shapes) in order to create abstractions at the proper scale, given a particular dataset. In my case I've received best result with kernel sizes 2, 5 and 3 for the first through the third Convolutional layers.

Question 2: Defend your choice of optimizer. Which optimizers did you test, and how did you determine which worked best?

Answer: As I've read in this blog, each optimizer has its own pros and cons and it was merely interesting to compare the same model behavior with different optimizers. So, I trained my start model with 50 epochs using all of them sequentially. Results you can see below.

Use the code cell below to plot the training and validation loss of your neural network. You may find this resource useful.

In [23]:

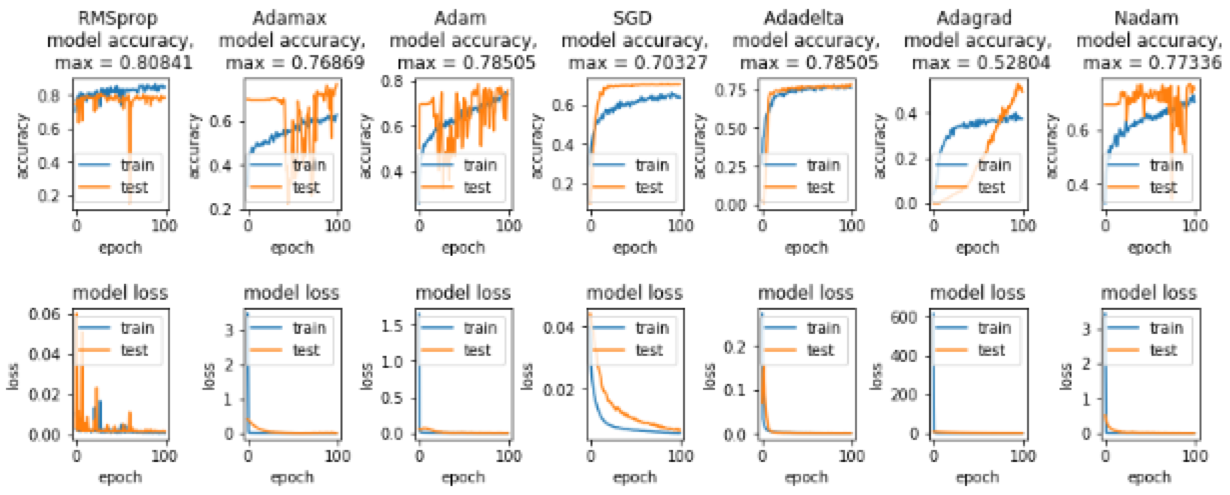
```
# Just methods for the data plotting
def plot_optim_acc(data,optim):
    # summarize history for accuracy
    plt.plot(data['acc'])
    plt.plot(data['val_acc'])
    plt.title('{} \n model accuracy, \n max =
{}'.format(optim,round(max(data['val_acc']),5)))

    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='lower right')

def plot_optim_loss(data, optim):
    # summarize history for loss
    plt.plot(data['loss'])
    plt.plot(data['val_loss'])
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper right')
```

In [24]:

```
from keras.optimizers import SGD, RMSprop, Adagrad, Adadelta, Adam, Adamax,
Nadam
from keras.callbacks import CSVLogger, ModelCheckpoint
import pandas as pd

data_rmsprop =
pd.read_csv('./opt_models/_no_no_f_500_1_dr_222_rmsprop_50_history.csv')
```

```
data_adamax =
pd.read_csv('./opt_models/model_start_adamax_32_noreg_ker222_dr_222_f_500_nobutn
orm_adamax_100.csv')
data_adam =
pd.read_csv('./opt_models/model_start_adam_32_noreg_ker222_dr_222_f_500_nobutnor
m_adamax_100.csv')
data_sgd =
pd.read_csv('./opt_models/model_start_sgd_32_noreg_ker222_dr_222_f_500_nobutnorm
_sgd_100.csv')
data_adadelta =
pd.read_csv('./opt_models/model_start_adadelta_32_noreg_ker222_dr_222_f_500_nobu
tnorm_adadelta_100.csv')
data_adagrad =
pd.read_csv('./opt_models/model_start_adagrad_32_noreg_ker222_dr_222_f_500_nobut
norm_adagrad_100.csv')
data_nadam =
pd.read_csv('./opt_models/model_start_nadam_32_noreg_ker222_dr_222_f_500_nobutno
rm_nadam_100.csv')

%matplotlib inline
fig1 = plt.figure()

ax1 = fig1.add_subplot(2,7,1)
plot_optim_acc(data_rmsprop,'RMSprop')

ax2 = fig1.add_subplot(2,7,2)
plot_optim_acc(data_adamax,'Adamax')

ax3 = fig1.add_subplot(278)
plot_optim_loss(data_rmsprop,'RMSprop')

ax4 = fig1.add_subplot(279)
plot_optim_loss(data_adamax,'Adamax')

ax5 = fig1.add_subplot(2,7,3)
plot_optim_acc(data_adam,'Adam')

ax6 = fig1.add_subplot(2,7,4)
plot_optim_acc(data_sgd,'SGD')

ax7 = fig1.add_subplot(2,7,10)
plot_optim_loss(data_adam,'Adam')

ax8 = fig1.add_subplot(2,7,11)
plot_optim_loss(data_sgd,'SGD')

ax9 = fig1.add_subplot(2,7,5)
plot_optim_acc(data_adadelta,'Adadelta')

ax10 = fig1.add_subplot(2,7,6)
plot_optim_acc(data_adagrad,'Adagrad')
```

```
ax11 = fig1.add_subplot(2,7,12)
plot_optim_loss(data_adadelta,'Adadelta')

ax12 = fig1.add_subplot(2,7,13)
plot_optim_loss(data_adagrad,'Adagrad')

ax13 = fig1.add_subplot(2,7,7)
plot_optim_acc(data_nadam,'Nadam')

ax14 = fig1.add_subplot(2,7,14)
plot_optim_loss(data_nadam,'Nadam')

plt.subplots_adjust(top=0.92, bottom=0.08, left=0.20, right=1.95, hspace=0.75,
wspace=0.75)

plt.show()
```



So, the best results, as well as the best graphs, were obtained with RMSprop and Adadelta optimizers.

SGD also gave good graph and I beleive that playing with hyperparameters could give us the better accuracy. A simple accumulation of a pulse already gives a good result, but Nesterov goes further and applies the well-known idea in computational mathematics: looking ahead for the update vector. But looking ahead can play a cruel joke with us if too large **gamma** and **eta**: we look so far that we miss past the areas with the opposite gradient sign.

The idea of **Adagrad** is to use *something* that would reduce updates for the items we are already updating. Nobody forces us to use this particular formula, therefore **Adagrad** is sometimes called a *family of algorithms*. The drawback of **Adagrad** is that the sum of the squares of updates can increase as much as necessary, which after a while leads to too small updates and paralysis of the algorithm. It showed the worst result.

If we'll modify the idea of **Adagrad**: still update less weight, which is too often updated, but instead of the full amount of updates, use averaged over the square gradient, then we'll get **RMSprop**. And **Adadelta** differs from it by the stabilizing term we add to the calculation of theta. Keras documentation recommends to leave the parameters of this optimizer at their default values, otherwise the latter can lead to paralysis of the algorithm, and can cause deliberately "greedy" behavior when the algorithm first updates the neurons that encode the best signs.

Adam is an *adaptive moment* estimation, another optimization algorithm. It combines both the idea of accumulation of motion and the idea of a weaker update of weights for typical features. But in my case the accuracy obtained with it is exactly equal to the one with **Adadelta** whenever the difference of prediction demanded more thin tuning.

Nadam and **Adamax** just subclasses of **Adam** played with momentum, as I understand, and in my case gave more worse result then **Adam** itself.

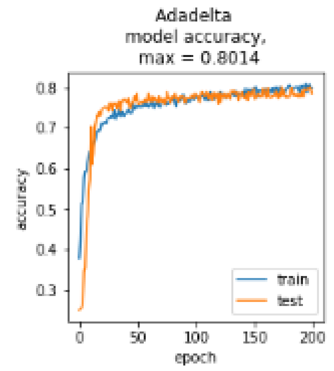
So, I chose two optimizers to research: **RMSprop** and **Adadelta**. Although Keras documentation does recommend to leave the parameters of this optimizer at their default values, I played with decay and learning rate and finally got not bad results for both of them. But when I obtained the final model I checked again both of them and the model with **RMSprop** won additional 3.5 %!.

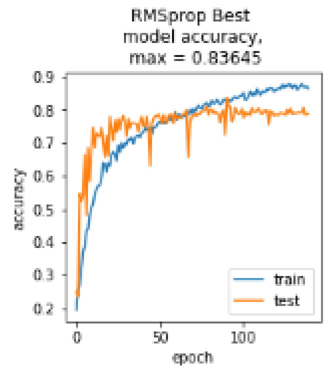
Once again, because of the troubles with calculating I have to conjunct few .cvs file into one to bild completed graph!!!

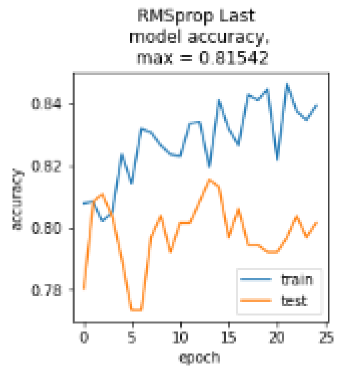
In [25]:

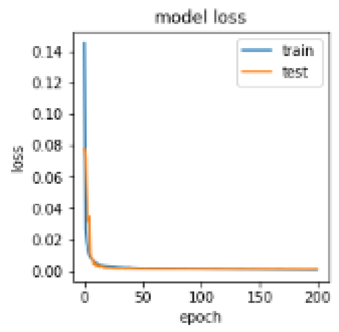
```
## TODO: Visualize the training and validation loss of your neural network
data_adadelta1 =
pd.read_csv('./model_cont_adadelta_new_32_noreg_ker222_dr_222_f_500_nobutnorm_20
0.csv')
```

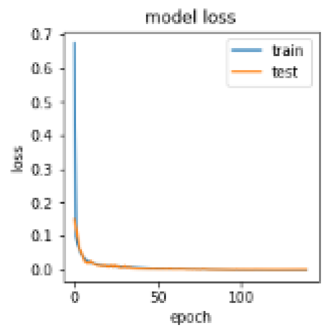
```
data_adadelta2 =
pd.read_csv('./model_cont_adadelta_new_32_noreg_ker222_dr_222_f_500_nobutnorm_20
0_n.csv')
data_adadelta3 = np.asarray(pd.concat(dict(df1 = data_adadelta1, df2 =
data_adadelta2),axis=0))
data_adadelta3=pd.DataFrame(data_adadelta3)
data_adadelta3=data_adadelta3.rename(columns = {1:'acc'})
data_adadelta3=data_adadelta3.rename(columns = {2:'loss'})
data_adadelta3=data_adadelta3.rename(columns = {3:'val_acc'})
data_adadelta3=data_adadelta3.rename(columns = {4:'val_loss'})

data_rmsprop1 =
pd.read_csv('./model_start_48_300_rmsprop_1regtanh_pre1tanh_conv_butchnorm_1_m99
_butchsize_16_l1_0001_d_000001_30.csv')
data_rmsprop2 = pd.read_csv('./model_cont_rms.csv')
data_rmsprop3 = pd.read_csv('./model_cont_rms1.csv')
data_rmsprop_best = np.asarray(pd.concat(dict(df1 = data_rmsprop1, df2 =
data_rmsprop2, df3 = data_rmsprop3),axis=0))
data_rmsprop_best = pd.DataFrame(data_rmsprop_best)
data_rmsprop_best = data_rmsprop_best.rename(columns = {1:'acc'})
data_rmsprop_best = data_rmsprop_best.rename(columns = {2:'loss'})
data_rmsprop_best = data_rmsprop_best.rename(columns = {3:'val_acc'})
data_rmsprop_best = data_rmsprop_best.rename(columns = {4:'val_loss'})

data_rmsprop_last =
pd.read_csv('./model_start_48_ker_253_dr_2222_fl_300_rmsprop_1regtanh_pre1tanh_c
onv_batchnorm_1_m99_batchsize_16_l1_0001_d_000001_75+25.csv')

%matplotlib inline
fig2 = plt.figure(figsize=(15,8))

ax1 = fig2.add_subplot(2,3,1)
plot_optim_acc(data_adadelta3,'Adadelta')

ax2 = fig2.add_subplot(2,3,4)
plot_optim_loss(data_adadelta3,'Loss')

ax3 = fig2.add_subplot(2,3,2)
plot_optim_acc(data_rmsprop_best,'RMSprop Best')

ax4 = fig2.add_subplot(2,3,5)
plot_optim_loss(data_rmsprop_best,'Loss')

ax5 = fig2.add_subplot(2,3,3)
plot_optim_acc(data_rmsprop_last,'RMSprop Last')

ax6 = fig2.add_subplot(2,3,6)
plot_optim_loss(data_rmsprop_last,'Loss')

plt.subplots_adjust(top=0.92, bottom=0.08, left=0.20, right=0.95, hspace=0.75,
wspace=0.75)
plt.show()
```

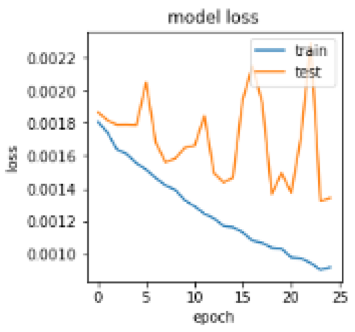












Question 3: Do you notice any evidence of overfitting or underfitting in the above plot? If so, what steps have you taken to improve your model? Note that slight overfitting or underfitting will not hurt your chances of a successful submission, as long as you have attempted some solutions towards improving your model (such as *regularization, dropout, increased/decreased number of layers, etc*).

Answer:

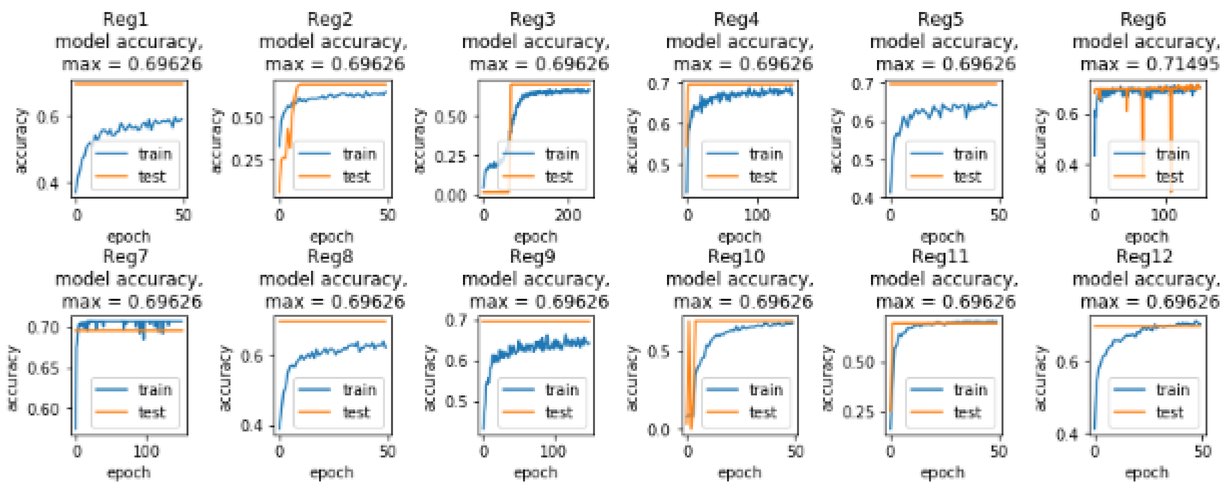
First, of course, I checked the L1 and L2 regularizations. I'd been awaiting that accuracy should decrease a little bit, but I was surprised by the results: (1) the accuracy decreased much more than I was waiting for and (2) when I used regularization for each Convolutional layer, all other things had no matter absolutely! **activity_regularizer** gave the worst result. But for the best solution I left the kernel_regularizer for the first Convolutional layer and it works pretty well. The samples of results with different models you can see below:

In [26]:

```
data_reg1 = pd.read_csv('./reg/17_312_no_no_gap_256_1_dr_224_lr_0001_d_01_rmsprop_50_history.csv')
data_reg2 = pd.read_csv('./reg/18_312_no_no_gap_512_1_dr_224_lr_0001_d_01_rmsprop_50_history.csv')
data_reg3 = pd.read_csv('./reg/32_batch128_312_no_t_gap_128_512_1_dr_225_lr_0001_d_01_rmsprop_250_history.csv')
data_reg4 = pd.read_csv('./reg/34_312_no_no_gap_40_300_1_dr_222_ker_753_lr_0001_d_01_rmsprop_150_history.csv')
data_reg5 = pd.read_csv('./reg/19_312_no_no_gap_64_512_1_dr_224_lr_0001_d_01_rmsprop_50_history.csv')
data_reg6 = pd.read_csv('./reg/36_312_no_no_gap_40_300_1_dr_222_ker_753_noreg_rmsprop_150_history.csv')
data_reg7 = pd.read_csv('./reg/38_312_t_t_gap_48_512_1_dr_222_ker_333_reg_no_rmsprop_250_history.csv')
data_reg8 = pd.read_csv('./reg/20_412_no_no_gap_64_512_2_dr_2244_lr_0001_d_01_rmsprop_50_history.csv')
```

```
data_reg9 =
pd.read_csv('./reg/74_reg_01_butch_64_3l2_gap_128_n_t_512_1_dr_2345_ker_953_rmsp
rop_l1_0001_d_01_150_history.csv')
data_reg10 =
pd.read_csv('./reg/75_reg_01_butch_64_3l2_f_256_n_t_512_1_dr_2345_ker_333_rmspro
p_l1_001_d_01_50_history.csv')
data_reg11 =
pd.read_csv('./reg/83_reg_01_butch_64_3l2_gap_256_t_t_512_1_bnorm_9befor_dr_2225
_ker_222_rmsprop_l1_001_d_01_50_history.csv')
data_reg12 =
pd.read_csv('./reg/model_adadelta_48_3l2reg_ker953_dr_2345_gap_512_nobutnorm_ada
delta_50.csv')

%matplotlib inline
fig1 = plt.figure()

ax1 = fig1.add_subplot(2,6,1)
plot_optim_acc(data_reg1,'Reg1')

ax2 = fig1.add_subplot(2,6,2)
plot_optim_acc(data_reg2,'Reg2')

ax3 = fig1.add_subplot(2,6,3)
plot_optim_acc(data_reg3,'Reg3')

ax4 = fig1.add_subplot(2,6,4)
plot_optim_acc(data_reg4,'Reg4')

ax5 = fig1.add_subplot(2,6,5)
plot_optim_acc(data_reg5,'Reg5')

ax6 = fig1.add_subplot(2,6,6)
plot_optim_acc(data_reg6,'Reg6')

ax7 = fig1.add_subplot(2,6,7)
plot_optim_acc(data_reg7,'Reg7')

ax8 = fig1.add_subplot(2,6,8)
plot_optim_acc(data_reg8,'Reg8')

ax9 = fig1.add_subplot(2,6,9)
plot_optim_acc(data_reg9,'Reg9')

ax10 = fig1.add_subplot(2,6,10)
plot_optim_acc(data_reg10,'Reg10')

ax11 = fig1.add_subplot(2,6,11)
plot_optim_acc(data_reg11,'Reg11')

ax12 = fig1.add_subplot(2,6,12)
plot_optim_acc(data_reg12,'Reg12')
```

```
plt.subplots_adjust(top=0.92, bottom=0.1, left=0.20, right=1.95, hspace=1.0,
wspace=0.75)

plt.show()
```



Answer (cont): Besides regularization, I've checked the **Batchnormalization** as well. The interesting thing here, that mostly they recommend to use it **after** activation layer, but I received the best result using it again only once and **before** the activation layer for the first Convolution. Then I found information on StackOverflow , that there is no final decision on this question yet. I've got the best result with standart configuration but increased momentum up to 0.99.

Also, I tried different **batch_size** in the **fit()** method and have got the best result with it equals 16.

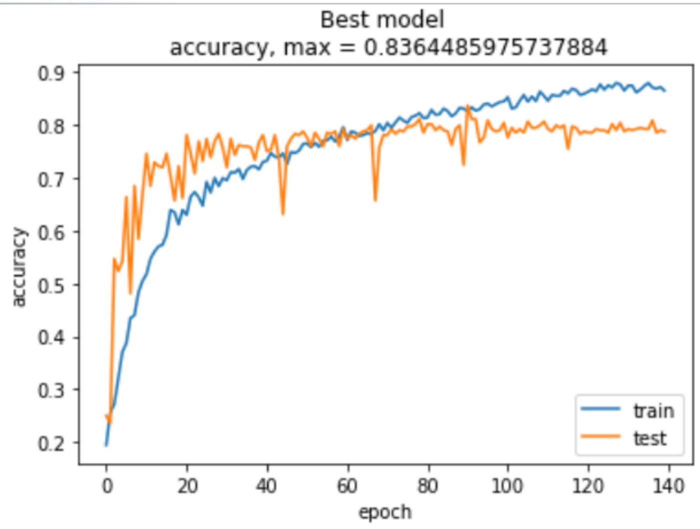
Additional layers did not improve final result, but the filters dimension had the large influence. And I was surprised when got the best result with configuration 2-5-3 when there is good practice to decrease slowly their dimension through layers.

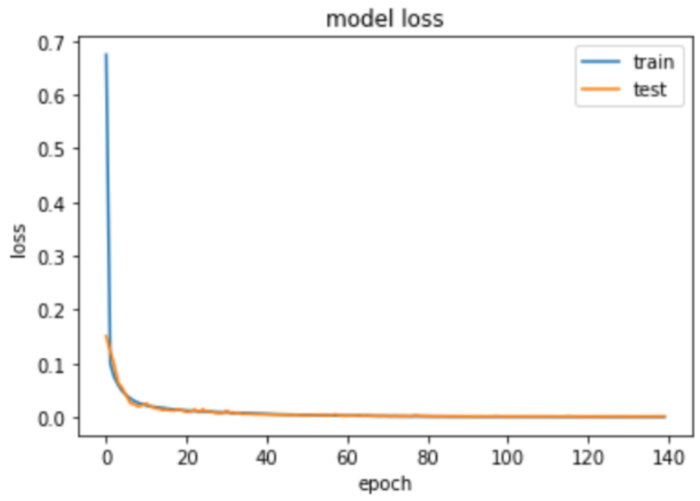
And of course, **dropout** helped significantly. The interesting thing here is the constant value 20%: any changes decreased result.

The graph of my best model training is below:

In [27]:

```
import pandas as pd

# summarize history for accuracy
plt.plot(data_rmsprop_best['acc'])
plt.plot(data_rmsprop_best['val_acc'])
plt.title('Best model \n accuracy, max =
{}'.format(max(data_rmsprop_best['val_acc'])))

plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='lower right')
plt.show()
# summarize history for loss
plt.plot(data_rmsprop_best['loss'])
plt.plot(data_rmsprop_best['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.show()
```



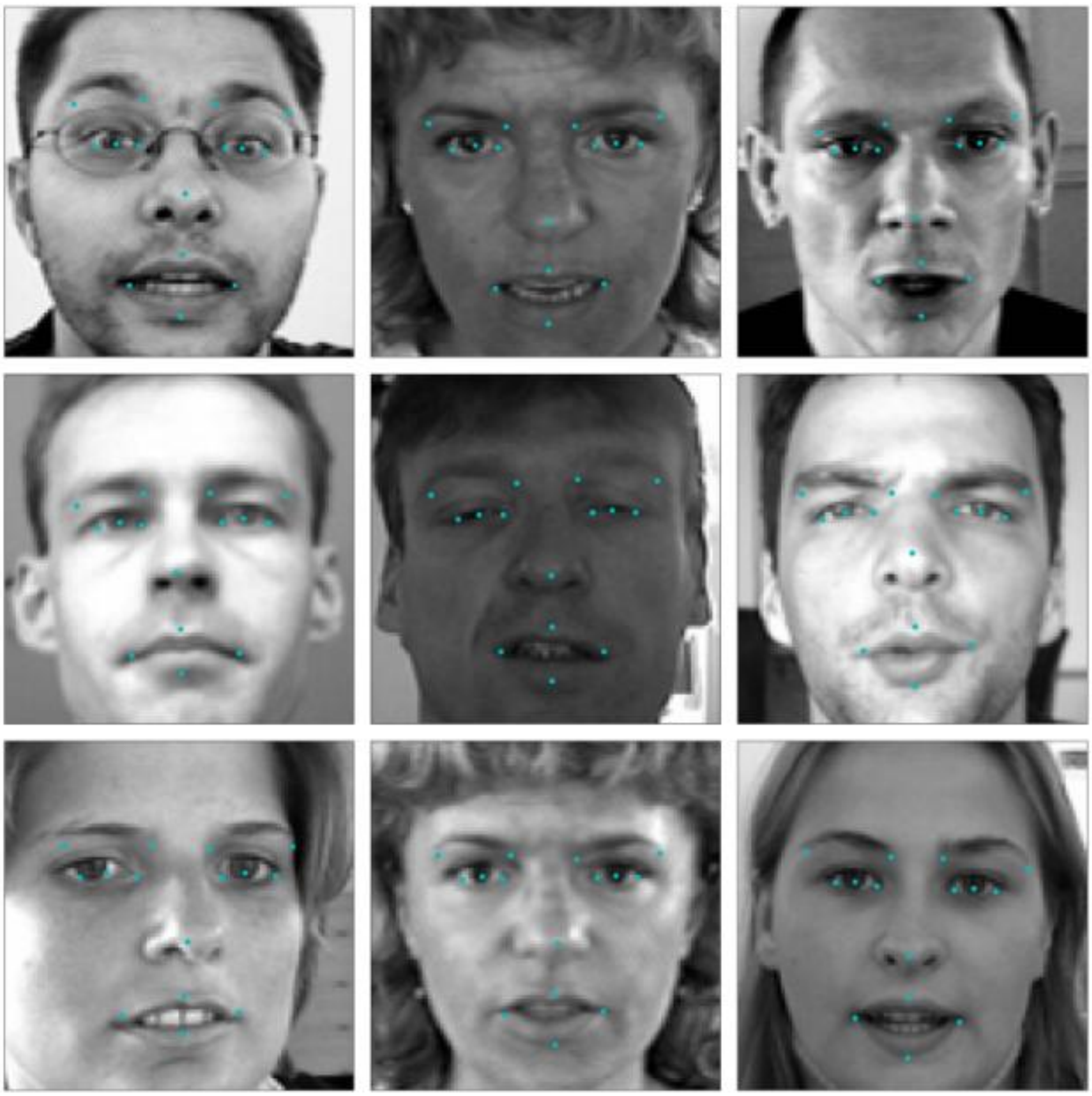


Visualize a Subset of the Test Predictions

Execute the code cell below to visualize your model's predicted keypoints on a subset of the testing images.

In [28]:

```
model = load_model('./model_cont_rms1.h5')
model.load_weights('./model_cont_rms1.h5')

y_test = model.predict(X_test)
fig = plt.figure(figsize=(20,20))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)
for i in range(9):
    ax = fig.add_subplot(3, 3, i + 1, xticks=[], yticks=[])
    plot_data(X_test[i], y_test[i], ax)
```



Step 8: Complete the pipeline

With the work you did in Sections 1 and 2 of this notebook, along with your freshly trained facial keypoint detector, you can now complete the full pipeline. That is given a color image containing a person or persons you can now

- Detect the faces in this image automatically using OpenCV
- Predict the facial keypoints in each face detected in the image
- Paint predicted keypoints on each face detected

In this Subsection you will do just this!

(IMPLEMENTATION) Facial Keypoints Detector

Use the OpenCV face detection functionality you built in previous Sections to expand the functionality of your keypoints detector to color images with arbitrary size. Your function should perform the following steps

1. Accept a color image.
2. Convert the image to grayscale.
3. Detect and crop the face contained in the image.
4. Locate the facial keypoints in the cropped image.
5. Overlay the facial keypoints in the original (color, uncropped) image.

Note: step 4 can be the trickiest because remember your convolutional network is only trained to detect facial keypoints in $96\times96$96×96 grayscale images where each pixel was normalized to lie in the interval $[0,1]$[0,1], and remember that each facial keypoint was normalized during training to the interval $[-1,1]$[−1,1]. This means - practically speaking - to paint detected keypoints onto a test face you need to perform this same pre-processing to your candidate face - that is after detecting it you should resize it to $96\times96$96×96 and normalize its values before feeding it into your facial keypoint detector. To be shown correctly on the original image the output keypoints from your detector then need to be shifted and re-normalized from the interval $[-1,1]$[−1,1] to the width and height of your detected face.

When complete you should be able to produce example images like the one below

In [29]:

```
# Load in color image for face detection
image = cv2.imread('images/obamas4.jpg')


# Convert the image to RGB colorspace
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
image_copy = np.copy(image)

# plot our image
fig = plt.figure(figsize = (9,9))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])
ax1.set_title('image copy')
ax1.imshow(image_copy)
```

Out[29]:

```
<matplotlib.image.AxesImage at 0x213016419b0>
```

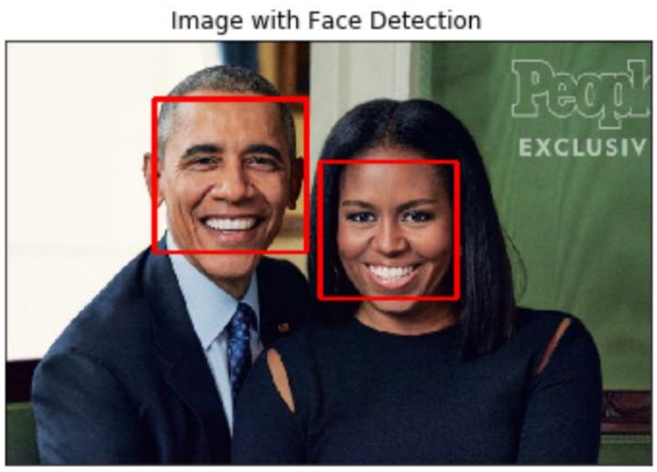
image copy

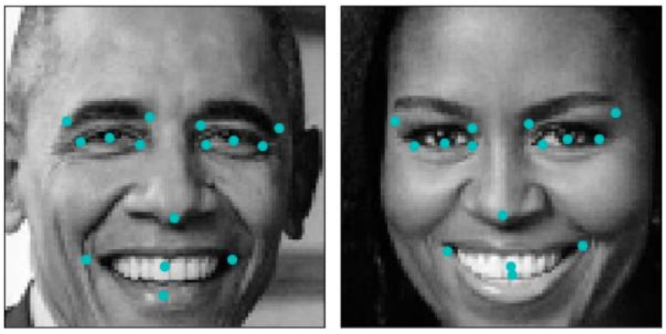
In [30]:

```
### TODO: Use the face detection code we saw in Section 1 with your trained
conv-net
# Convert the RGB  image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

# Extract the pre-trained face detector from an xml file
face_cascade =
cv2.CascadeClassifier('detector_architectures/haarcascade_frontalface_default.xm
l')

# Detect the faces in image
faces = face_cascade.detectMultiScale(gray, 1.2, 5)

# Print the number of faces detected in the image
print('Number of faces detected:', len(faces))

# Make a copy of the orginal image to draw face detections on
image_with_detections = np.copy(image_copy)
```

```
# Get the bounding box for each detected face
for (x,y,w,h) in faces:
    # Add a red bounding box to the detections image
    cv2.rectangle(image_with_detections, (x,y), (x+w,y+h), (255,0,0), 3)

# Display the image with the detections
fig = plt.figure(figsize = (6,6))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])

ax1.set_title('Image with Face Detection')
ax1.imshow(image_with_detections)

# Crop the face contained in the image.
faces_with_keypoints = np.ndarray(shape=(len(faces),96,96,1), dtype=float)
face_number = 0

for (x,y,w,h) in faces:
    # Extract a face
    face = gray[y:y+h,x:x+w]
    # Resize it to the size (96x96) and normalize it to be in [0,1]
    face = cv2.resize(face, (96,96)) / 255
    # Add it to the list of faces
    faces_with_keypoints[face_number,:,:,0] = face
    face_number = face_number + 1

# Predict facial keypoints
facial_keypoints = model.predict(faces_with_keypoints)

# Locate the facial keypoints in the cropped image.
fig = plt.figure(figsize=(10,10))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)
for i in range(len(faces)):
    ax = fig.add_subplot(3, 3, i + 1, xticks=[], yticks=[])
    plot_data(faces_with_keypoints[i], facial_keypoints[i], ax)
```

```
Number of faces detected: 2
```





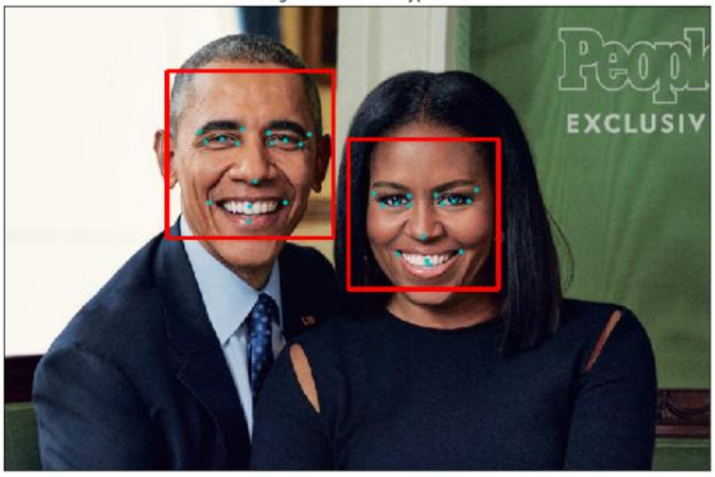
In [31]:

```
## TODO : Paint the predicted keypoints on the test image
## TODO : Paint the predicted keypoints on the test image
fig = plt.figure(figsize = (10,10))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])

faces_with_keypoints = np.ndarray(shape=(len(faces),96,96,1), dtype=float)

for (x,y,w,h) in faces:
    # Detect a face
    face = gray[y:y+h, x:x+w]
    # Resize it to the size (96x96) and normalize it to be in [0,1]
    face_with_keypoints = cv2.resize(face, (96,96)) / 255
    # Input
    faces_with_keypoints[0,:,:,0] = face_with_keypoints
    # Predict facial keypoints
    facial_keypoints = model.predict(faces_with_keypoints)[0]
    # Overlay the facial keypoints in the original (color, uncropped) image
    ax1.scatter(facial_keypoints[0::2] * (w / 2) + (w / 2) + x,
                facial_keypoints[1::2] * (h / 2) + (h / 2) + y,
                marker='o', c='c', s=8)

# Display the image
ax1.set_title('Image with Facial Keypoints')
ax1.imshow(image_with_detections)
```

Out[31]:

```
<matplotlib.image.AxesImage at 0x213009a5588>
```

Image with Facial Keypoints



(Optional) Further Directions - add a filter using facial keypoints to your laptop camera

Now you can add facial keypoint detection to your laptop camera - as illustrated in the gif below.

The next Python cell contains the basic laptop video camera function used in the previous optional video exercises. Combine it with the functionality you developed for keypoint detection and marking in the previous exercise and you should be good to go!

In [32]:

```
import cv2
import time
from keras.models import load_model
def laptop_camera_go():
    # Create instance of video capturer
    cv2.namedWindow("face detection activated")
    vc = cv2.VideoCapture(0)

    # Try to get the first frame
    if vc.isOpened():
        rval, frame = vc.read()
    else:
        rval = False

    face_cascade =
cv2.CascadeClassifier('detector_architectures/haarcascade_frontalface_default.xm
l')
    faces_with_keypoints = np.ndarray(shape=(1,96,96,1), dtype=float)

    # keep video stream open
    while rval:
        gray = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
        faces = face_cascade.detectMultiScale(gray, 1.25, 6)

        for (x,y,w,h) in faces:
            # Detect a face
            face = gray[y:y+h, x:x+w]
            # Resize it to the size (96x96) and normalize it to be in [0,1]
            face_with_keypoints = cv2.resize(face, (96,96)) / 255
            # Input
            faces_with_keypoints[0,:,:,0] = face_with_keypoints
            # Predict facial keypoints
```

```
                facial_keypoints = model.predict(faces_with_keypoints)[0]
                # Display each face rectangle in the color image
                cv2.rectangle(frame, (x,y), (x+w,y+h), (0,0,255), 3)
                facial_keypoints[0::2] = facial_keypoints[0::2] * (w / 2) + (w / 2)
+ x
                facial_keypoints[1::2] = facial_keypoints[1::2] * (h / 2) + (h / 2)
+ y
                for (x_point,y_point) in zip(facial_keypoints[0::2],
facial_keypoints[1::2]):
                    cv2.circle(frame, (x_point, y_point), 3, (0,255,0), -1)

        # plot image from camera with detections marked
        cv2.imshow("face detection activated", frame)

        # exit functionality - press any key to exit laptop video
        key = cv2.waitKey(20)
        if key > 0: # exit by pressing any key
            # destroy windows
            cv2.destroyAllWindows()

            # hack from stack overflow for making sure window closes on osx -->
https://stackoverflow.com/questions/6116564/destroywindow-does-not-close-window-
on-mac-using-python-and-opencv
            for i in range (1,5):
                cv2.waitKey(1)
            return

        # read next frame
        time.sleep(0.05)             # control framerate for computation -
default 20 frames per sec
        rval, frame = vc.read()
```

In [33]:

```
# Run your keypoint face painter
laptop_camera_go()
```