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 (https://en.wikipedia.org/wiki/OpenCV#History) 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 (http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github (https://github.com/opencv/opencv/tree/master/data/haarcascades). 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 (http://docs.opencv.org/3.2.0/df/d9d/tutorial_py_colorspaces.html). 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 0x26925bd3d68>

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 (https://stackoverflow.com/questions/20801015/recommended-values-for-opencv-detectmultiscale-parameters).

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_frontalf

# 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 0x26925c86b38>
```

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 (https://github.com/opencv/opencv/tree/master/data/haarcascades).

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 0x26925ceb908>

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_frontalf

# 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 0x26925cfc668>
```

(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`

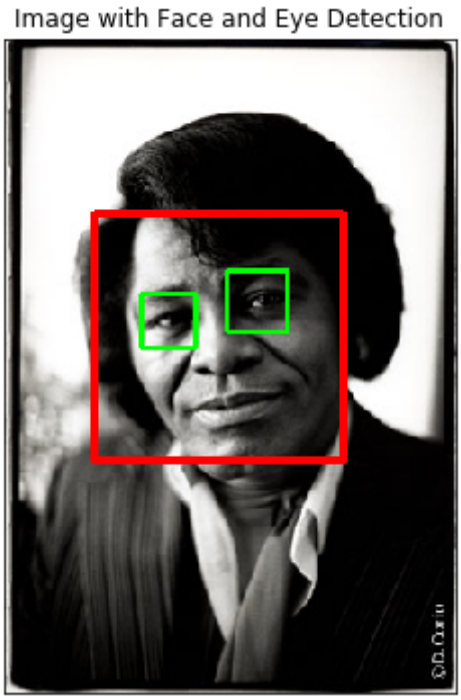
Second, since we've already run the face detector over this image, you should only search for eyes *within the rectangular face regions detected in* `faces`. This will minimize false detections.

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 detect

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

gray_img = cv2.cvtColor (image, cv2.COLOR_RGB2GRAY)

## Iterate eye detection over face regions detected previously
for (x,y,w,h) in faces:
    fgray = gray_img[y:y+h, x:x+w]
    fcolor = image_with_detections[y:y+h, x:x+w]
    eyes = eye_cascade.detectMultiScale(fgray, 1.10, 6)
    print('Number of eyes detected in this face:', len(eyes))

    ## include a green rectangle around each eye.
    for (ex,ey,ew,eh) in eyes:
        cv2.rectangle(fcolor,(ex,ey),(ex+ew,ey+eh),(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)
```

```
Number of eyes detected in this face: 2
```

Out[6]:
```
<matplotlib.image.AxesImage at 0x26925dcdc18>
```



(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 fee

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)

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

    # Keep the video stream open
    while rval:

        ## include code for detecting faces & eyes and markers
        # Convert the RGB  image to grayscale
        gray = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)

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

        # Detect the faces in image
        ## Resizing and minimum neighbors parameters may need tuning depending ca
        faces = face_cascade.detectMultiScale(gray, 1.10, 6)

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

        for (x,y,w,h) in faces:
            fgray = gray[y:y+h, x:x+w]
            fcolor = frame[y:y+h, x:x+w]

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

            ## Resizing and minimum neighbors parameters may need tuning dependin
            eyes = eye_cascade.detectMultiScale(fgray, 1.10, 6)
            #print('Number of eyes detected in this face:', len(eyes))

            ## box the eye rectangles
            for (ex,ey,ew,eh) in eyes:
                cv2.rectangle(fcolor,(ex,ey),(ex+ew,ey+eh),(0,255,0),2)

            ## box the face rectangle
            cv2.rectangle(frame, (x,y), (x+w,y+h),(255,0,0), 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 - defaul
        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 (https://digital-photography-school.com/how-to-avoid-and-reduce-noise-in-your-images/), 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:
noise_level = 40  # 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_noi

# 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 0x269271ca320>

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_frontalf

# 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: 11
```

Out[10]:
```
<matplotlib.image.AxesImage at 0x269269bfba8>
```

With this added noise we now miss one of the faces!

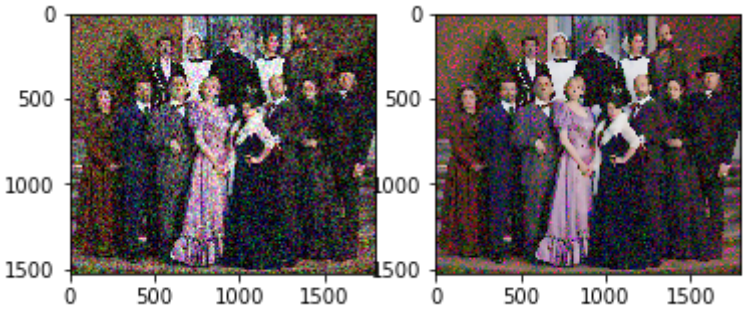
(IMPLEMENTATION) De-noise this image for better face detection

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 (http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_photo/py_non_local_means/py_non_local_means.ht

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 no


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

plt.subplot(121),plt.imshow(image_with_noise)
plt.subplot(122),plt.imshow(denoised_image)
plt.show()
```



In [12]:
```
## TODO: Run the face detector on the de-noised image to improve your detections

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

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

# Detect the faces in image,  the scale factor and minimum adjacent lines impact
faces = face_cascade.detectMultiScale(gray, 1.20, 9)

# 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 = (6,6))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])

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

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

Out[12]:
```
<matplotlib.image.AxesImage at 0x269293c7710>
```

Step 3: Blur an Image and Perform Edge Detection

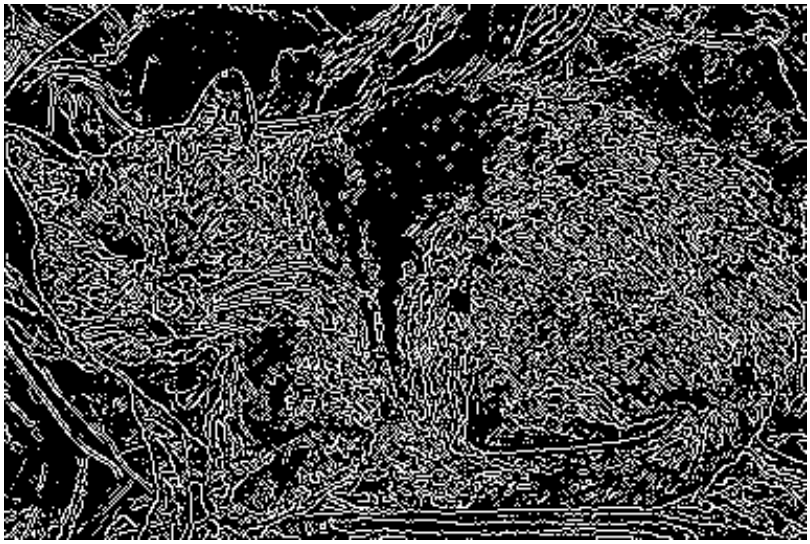
Now that we have developed a simple pipeline for detecting faces using OpenCV - let's start playing around with a few fun things we can do with all those detected faces!

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 (https://en.wikipedia.org/wiki/Gaussian_blur#Common_uses), 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 (http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html).

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 0x26929439d30>

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).

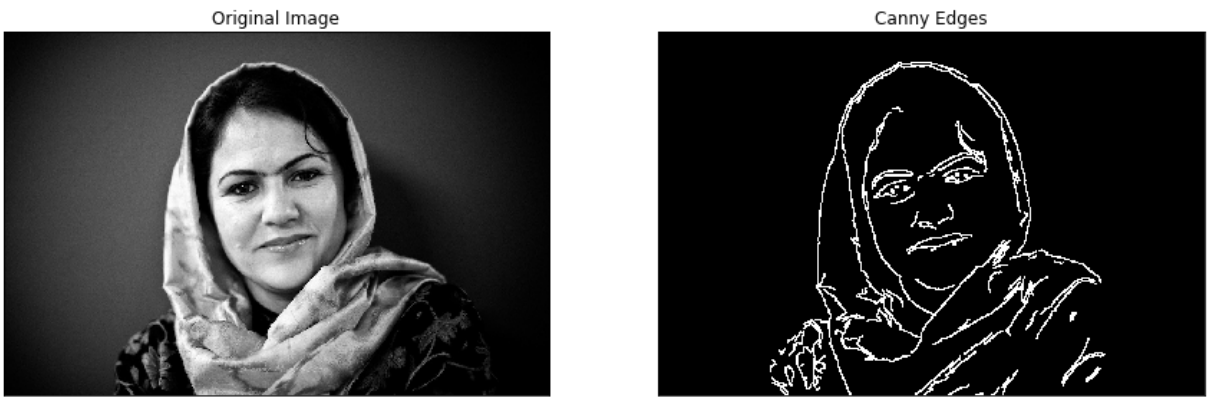
(IMPLEMENTATION) Blur the image *then* perform edge detection

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 (http://docs.opencv.org/3.1.0/d4/d13/tutorial_py_filtering.html) - and use an *averaging kernel* of width equal to 4.

In [14]:
```
# 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)

### 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
gray = cv2.filter2D(gray,-1,kernel)

## TODO: Then perform Canny edge detection and display the output
# 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[14]: <matplotlib.image.AxesImage at 0x2692936af28>



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 (https://www.google.com/streetview/) - 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/obamas4.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 0x269294bb400>

(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_frontalf

# Detect the faces in image
faces = face_cascade.detectMultiScale(gray, 1.3, 10)

# 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 f
    face_rgn = image_with_detections[y:y+h, x:x+w]

    ### TODO: Blur the test imageusing OpenCV's filter2d functionality,
    # Use an averaging kernel, and a kernel width shall be large enough to mask.
    kernel = np.ones((31,31),np.float32)/961
    face_rgn = cv2.filter2D(face_rgn,-1,kernel)

    ##  write blurred region into image to be displayed
    image_with_detections [y:y+face_rgn.shape[0], x:x+face_rgn.shape[1]] = face_r

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

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

Out[16]:
```
<matplotlib.image.AxesImage at 0x26926821eb8>
```

(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 i
import cv2
import time

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

    # Keep video stream open
    while rval:
        # Plot image from camera with detections marked

        ## the face masking logic starts here. ##############################

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

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

        # Detect the faces in image
        faces = face_cascade.detectMultiScale(gray, 1.09, 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(frame)

        # Get the bounding box for each detected face
        for (x,y,w,h) in faces:

            ## TODO: Blur the bounding box around each detected face using an ave
            face_rgn = frame [y:y+h, x:x+w]

            ### TODO: Blur the test imageusing OpenCV's filter2d functionality,
            # Use an averaging kernel, and a kernel width shall be large enough t
            kernel = np.ones((31,31),np.float32)/961
            face_rgn = cv2.filter2D(face_rgn,-1,kernel)

            ##  write blurred region into image to be displayed
            frame [y:y+face_rgn.shape[0], x:x+face_rgn.shape[1]] = face_rgn

        ## the face masking logic ends here.#####################################

        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 - defaul
        rval, frame = vc.read()
```

In [18]:
```
# Run laptop identity hider
laptop_camera_go()
```

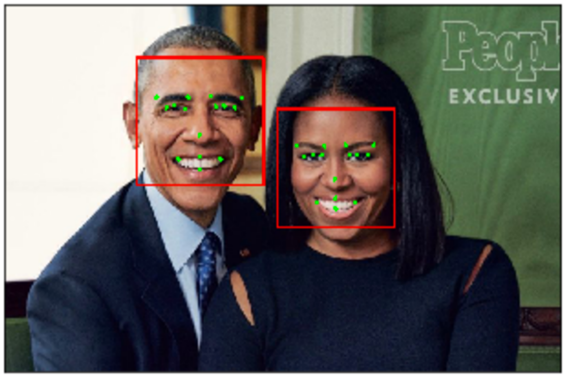
Step 5: Create a CNN to Recognize Facial Keypoints

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) 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 (https://www.kaggle.com/c/facial-keypoints-detection/data). 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 96×96 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) 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]:
```
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 (http://danielnouri.org/notes/2014/12/17/using-convolutional-neural-nets-to-detect-facial-keypoints-tutorial/), 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 $[-1, 1]$, while the pixel values of each input point (a facial image) have been normalized to the range $[0, 1]$.

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 96×96 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 (http://danielnouri.org/notes/2014/12/17/using-convolutional-neural-nets-to-detect-facial-keypoints-tutorial/), but you are not permitted to copy any of the architectures that you find online.

In [21]:
```
# Import deep learning resources from Keras
import keras
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

#from keras.layers import Flat  ## giving error
## 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

### TODO: Define your architecture.
krnl_sz = 3                ## kernel size for convolutions - smaller better for b
pl_sz = 2                  ## pool size for max pooling
classes= 30                ## output classes - nornalized  x, y positions of each
model = Sequential()

## FIRST LAYER SET
##  Define the first 2D convolutional layer with proper input shape and 32 filter
model.add(Conv2D(filters=32, kernel_size= krnl_sz, padding='same', activation='re
                         input_shape=(96, 96, 1)))
## Add pooling layer
model.add(MaxPooling2D(pool_size= pl_sz))

## Add minimal drop out in early stage.
model.add(Dropout(0.10))    ## dropout reduces the risk of overfitting.


#### SECOND LAYER SET
## Define the second 2D with increased number of filters and relu activation.
model.add(Conv2D(filters=64, kernel_size=krnl_sz, padding='same', activation='rel

## Add pooling layer
model.add(MaxPooling2D(pool_size=pl_sz))

## Add moderate drop out to reduce overfitting.
model.add(Dropout(0.20))

####  THIRD LAYER SET
## Define the third 2D with increased number of filters and relu activation.
model.add(Conv2D(filters=128, kernel_size=krnl_sz, padding='same', activation='re

## Add pooling layer
model.add(MaxPooling2D(pool_size=pl_sz))

## Add typical drop out to reduce overfitting.
model.add(Dropout(0.3))

# Flatten => RELU layers
model.add(Flatten())

###  Add Fully connected dense layer with 500 nodes
model.add(Dense(500, activation='relu'))

## Add strong drop out rate at this later stage.
model.add(Dropout(0.40))
```

```
### ensure default linear output without 'relu' or 'softmax'
model.add(Dense(classes)) #no softmax i.e. no non-linearity


# Summarize the model
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 96, 96, 32)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 48, 48, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 48, 48, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 48, 48, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 24, 24, 64)        0
_________________________________________________________________
dropout_2 (Dropout)          (None, 24, 24, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 24, 24, 128)       73856
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 12, 12, 128)       0
_________________________________________________________________
dropout_3 (Dropout)          (None, 12, 12, 128)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 18432)             0
_________________________________________________________________
dense_1 (Dense)              (None, 500)               9216500
_________________________________________________________________
dropout_4 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_2 (Dense)              (None, 30)                15030
=================================================================
Total params: 9,324,202.0
Trainable params: 9,324,202.0
Non-trainable params: 0.0
_________________________________________________________________
```

DO NOT COMPILE & Train, i.e. Skip Step 6 if you want to use pre-trained weights, available.

I used AWS (GPU) equipped platform for training.

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 (https://keras.io/models/sequential/#sequential-model-methods) to configure the learning process. Experiment with your choice of optimizer (https://keras.io/optimizers/); 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 (https://keras.io/models/sequential/#sequential-model-methods) 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 (https://keras.io/getting-started/faq/#how-can-i-save-a-keras-model) as an HDF5 file with file path `my_model.h5`.

In [63]:
```
from keras.optimizers import SGD, RMSprop, Adagrad, Adadelta, Adam, Adamax, Nadam
from keras.callbacks import ModelCheckpoint

## TODO: Compile the model
# TODO: Compile the model using a loss function and an optimizer.
model.compile(loss = 'mean_squared_error', optimizer='rmsprop', metrics=['accurac


## TODO: Train the model
checkpointer = ModelCheckpoint(filepath='weights.best.rmsprop.hdf5',
                               verbose=1, save_best_only=True)

# TODO: Run the model. Feel free to experiment with different batch sizes and num
hist = model.fit(X_train, y_train, epochs=50, batch_size=64, callbacks=[checkpoin

## TODO: Save the model as model.h5
model.save('my_model.h5')
#save model
model_json = model.to_json()
open('my_model_architecture.json', 'w').write(model_json)
```

```
Train on 1712 samples, validate on 428 samples
Epoch 1/50
1664/1712 [============================>.] - ETA: 0s - loss: 0.7727 - acc: 0.
2987

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
poch 00000: val_loss improved from inf to 0.03698, saving model to weights.be
st.rmsprop.hdf5
1712/1712 [==============================] - 30s - loss: 0.7516 - acc: 0.3032
- val_loss: 0.0370 - val_acc: 0.6963
Epoch 2/50
1664/1712 [============================>.] - ETA: 0s - loss: 0.0223 - acc: 0.
4189

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
poch 00001: val_loss improved from 0.03698 to 0.03622, saving model to weight
s.best.rmsprop.hdf5
1712/1712 [==============================] - 30s - loss: 0.0221 - acc: 0.4194
- val_loss: 0.0362 - val_acc: 0.6963
Epoch 3/50
```

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 followed the following steps to realize the final CNN network architecture. Step 0: I defined small kernel size and pool size parameters to preserve local properties reasonably well. I decided to use typical activation function 'relu' at all stages, except last stage.

Step 1: I started with a 2D CONV layer, which accepts the input data with specified shape (,96,96,1). I started with typical number of (32) filters, as the premitives extracted would be rudimentary. I added a max pooling layer with specified pool size. I added drop out to minimize the risk of overfitting with mimimal amount at this early stage.

Step 2: I added another 2D CNN layer, now with 64 filters, followed by max pooling to extract summary features and added moderate dropout to reduce overfitting.

Step 3: I added another 2D CNN layer, now with 128 filters, followed by max pooling to extract summary features and added reasonable dropout to reduce overfitting.

Step 4: I flattened and added the dense layer with 500 nodes, with strong dropout to avoid overfitting.

Step 5: I defined the output layer with 30 classes (normalized x, y coordinates of each feature point of 15 feature points). I did not use softmax to avoid the loss of negative values of normalized coordinates. If used softmax, I never was able to optimize the loss values during training. I left it default linear activation, without specifying.

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

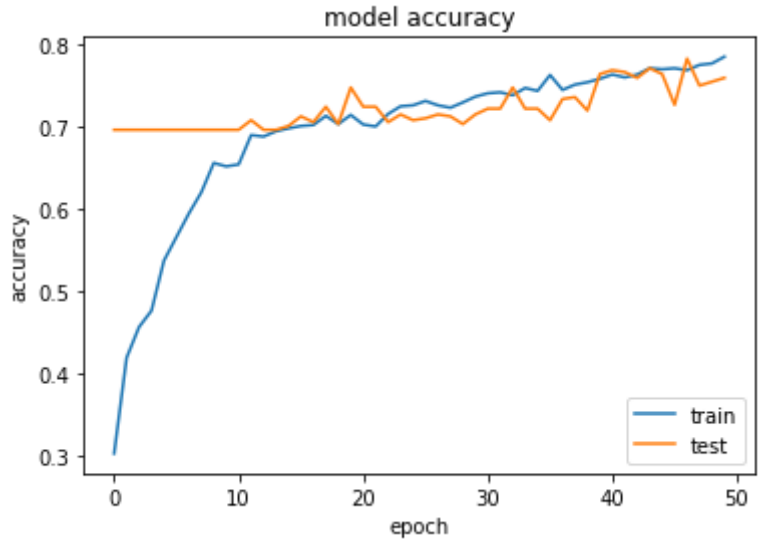
Answer: Based on the review of literature and observations of ['adam', 'rmsprop', 'nadam' and 'sgd'] optimizers, I selected 'rmsprop', as it has better validation loss optimizatio as well as better validation accuracy. The optimizer 'rmsprop' converged faster as well. The optimizer 'sgd' fared worst both in terms of validation loss and validation accuracy.

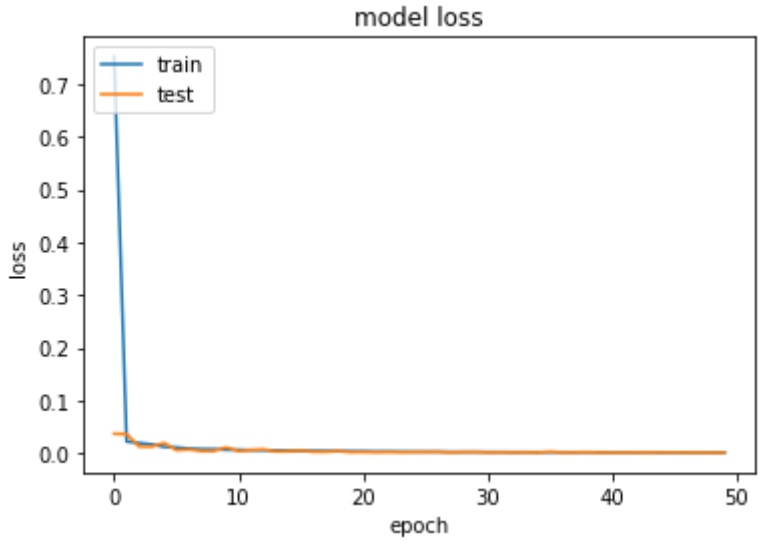
Further I provided additional information in 'optimizers.pdf' regardig compare and contrast of these optimizers with visualized training data and validation loss graphs.

Use the code cell below to plot the training and validation loss of your neural network. You may find this resource (http://machinelearningmastery.com/display-deep-learning-model-training-history-in-keras/) useful.

In [65]:
```
## TODO: Visualize the training and validation loss of your neural network
# list all data in history
print(hist.history.keys())
# summarize history for accuracy
plt.plot(hist.history['acc'])
plt.plot(hist.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='lower right')
plt.show()
# summarize history for loss
plt.plot(hist.history['loss'])
plt.plot(hist.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
```





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: I used dropouts in a disciplined manner (progressively increasing manner) to mitigate the overfitting effect.

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 [22]:
```
## load the best weights for the model
model.load_weights('weights.best.rmsprop.hdf5')
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!

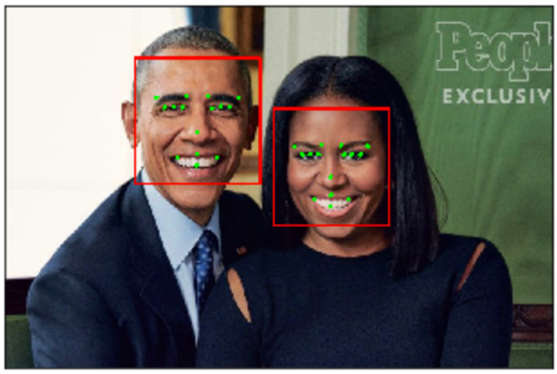
(IMPLEMENTATION) Facial Keypoints Detector

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.

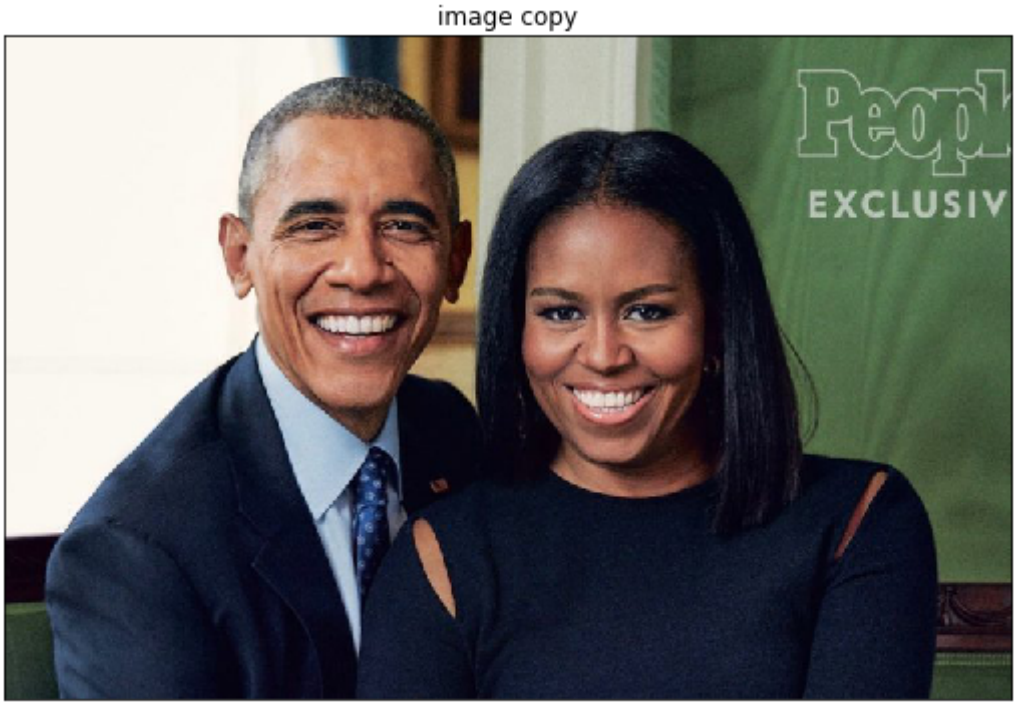
Note: step 4 can be the trickiest because remember your convolutional network is only trained to detect facial keypoints in 96×96 grayscale images where each pixel was normalized to lie in the interval $[0,1]$, and remember that each facial keypoint was normalized during training to the interval $[-1,1]$. 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 96×96 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 $[-1,1]$ to the width and height of your detected face.

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



In [23]:
```
# 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)

## copy the image
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[23]: <matplotlib.image.AxesImage at 0x26938339dd8>

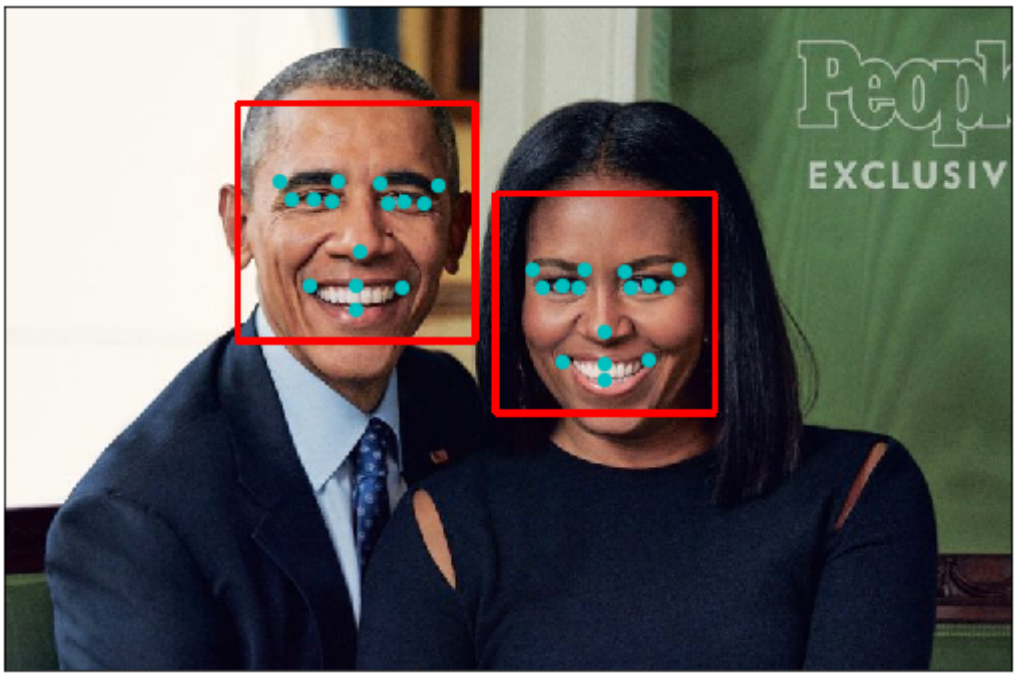


In [24]:
```
### TODO: Use the face detection code we saw in Section 1 with your trained conv-

### TODO: Use the face detection code we saw in Section 1 with your trained conv-
## TODO : Paint the predicted keypoints on the test image
### TODO: Use the face detection code we saw in Section 1 with your trained conv-


def feature_point_detect_plot (img, model, face_cascade):
    # imag - RGB image
    # model - loaded conv net model with trained weights
    # face_cascade - initialized


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

    ##3. Detect and crop the face contained in the image.
    # Detect the faces in image
    faces = face_cascade.detectMultiScale(gray, 1.02, 10)

    # 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(img)

    fig = plt.figure(figsize=(9,9))
    ax = fig.add_subplot(1, 1, 1, xticks=[], yticks=[])

    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)
        ax.imshow(image_with_detections)

    # 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)

        ## extract face region from gray image to feed to the model
        fgray = gray[y:y+h, x:x+w]

        ## resize and convert normalize the pixel values to [0 .. 1]
        resize_gray_crop = cv2.resize(fgray, (96, 96)) / 255.0

        ## Reshape or expand the dimensions
        model_input = np.expand_dims(np.expand_dims(resize_gray_crop, axis=-1), a

        ## Locate the facial keypoints in the cropped image.
        feature_points = model.predict(model_input)

        ##  squeeze dimensions
        feature_points = np.squeeze(feature_points)

        orig_crop_shape = (image_with_detections [y:y+h, x:x+w]).shape
```

```
        ax.scatter(((feature_points[0::2] * 48 + 48)*orig_crop_shape[0]/96)+x,
                   ((feature_points[1::2] * 48 + 48)*orig_crop_shape[1]/96)+y,
                   marker='o', c='c', s=40)

    plt.show()
```

In [25]:
```
# Extract the pre-trained face detector from an xml file
face_cascade = cv2.CascadeClassifier('detector_architectures/haarcascade_frontalf

#invoke the feature detector and plotter
feature_point_detect_plot(image_copy, model, face_cascade)
```

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



(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 [26]:
```
def detect_display_features (frame, model, face_cascade):
    # frame - a frame from camera video.
    # model - loaded conv net model with trained weights
    # face_cascade - initialized
    # returns modified frame

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

    ##3. Detect and crop the face contained in the image.
    # Detect the faces in image
    num_faces = face_cascade.detectMultiScale(gray, 1.03, 10)

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

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


    for (x,y,w,h) in num_faces:
        # Add a red bounding box to the detections image
        cv2.rectangle(frame_with_detections, (x,y), (x+w,y+h), (255,0,0), 3)


    # Get the bounding box for each detected face
    for (x,y,w,h) in num_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)

        ## extract face region from gray image to feed to the model
        fgray = gray[y:y+h, x:x+w]

        ## resize and convert normalize the pixel values to [0 .. 1]
        resize_gray_crop = cv2.resize(fgray, (96, 96)) / 255.0
        # print ("resized gray crop shape {}".format(resize_gray_crop.shape))

        ## Reshape or expand the dimensions
        model_input = np.expand_dims(np.expand_dims(resize_gray_crop, axis=-1), a
        # print ("model_input shape {}".format(model_input.shape))

        ## Locate the facial keypoints in the cropped image.
        feature_points = np.squeeze(model.predict(model_input))
        # print ("feature_points shape {}".format(feature_points.shape))

        #capture cropped original image shape needed for de-normalizing the featu
        orig_crop_shape = (frame_with_detections [y:y+h, x:x+w]).shape
        # print ("orig_crop shape {}".format(orig_crop_shape))


        x_indx = 0
        y_indx = 1
        #de normalize and apply translation on normalized values of feature point
        for i in range (15):
                x_cord = int (((feature_points [x_indx] * 48 + 48) * orig_crop_sh
                y_cord = int (((feature_points [y_indx] * 48 + 48) * orig_crop_sh
```

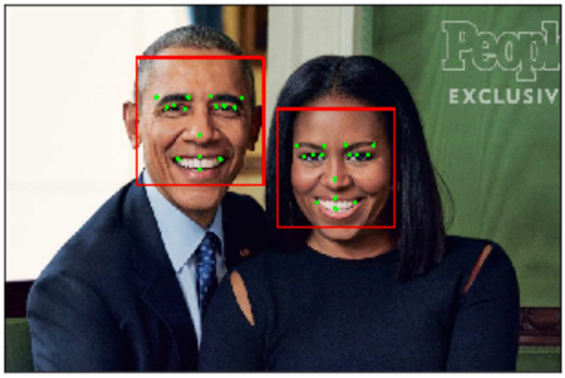
```
                frame_with_detections = cv2.circle(frame_with_detections, (x_cord

                x_indx = x_indx + 2
                y_indx = y_indx + 2

    return frame_with_detections
```

In [27]:
```
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

    # keep video stream open
    while rval:
        # plot image from camera with detections marked


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

        #invoke the feature detector and plotter
        frame = detect_display_features (frame, model, face_cascade)
        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 --> h
            for i in range (1,5):
                cv2.waitKey(1)
            return

        # read next frame
        time.sleep(0.05)             # control framerate for computation - defaul
        rval, frame = vc.read()
```

In [28]:
```
# Run your keypoint face painter
laptop_camera_go()
```

(Optional) Further Directions - add a filter using facial keypoints

Using your freshly minted facial keypoint detector pipeline you can now do things like add fun filters to a person's face automatically. In this optional exercise you can play around with adding sunglasses automatically to each individual's face in an image as shown in a demonstration image below.



To produce this effect an image of a pair of sunglasses shown in the Python cell below.

In [92]:
```
# Load in sunglasses image - note the usage of the special option
# cv2.IMREAD_UNCHANGED, this option is used because the sunglasses
# image has a 4th channel that allows us to control how transparent each pixel in
sunglasses = cv2.imread("images/sunglasses_4.png", cv2.IMREAD_UNCHANGED)

# Plot the image
fig = plt.figure(figsize = (6,6))
ax1 = fig.add_subplot(111)
ax1.set_xticks([])
ax1.set_yticks([])
ax1.imshow(sunglasses)
ax1.axis('off');
```

In [ ]:
```

```

This image is placed over each individual's face using the detected eye points to determine the location of the sunglasses, and eyebrow points to determine the size that the sunglasses should be for each person (one could also use the nose point to determine this).

Notice that this image actually has *4 channels*, not just 3.

In [93]:
```
# Print out the shape of the sunglasses image
print ('The sunglasses image has shape: ' + str(np.shape(sunglasses)))
```

```
The sunglasses image has shape: (1123, 3064, 4)
```

It has the usual red, blue, and green channels any color image has, with the 4th channel representing the transparency level of each pixel in the image. Here's how the transparency channel works: the lower the value, the more transparent the pixel will become. The lower bound (completely transparent) is zero here, so any pixels set to 0 will not be seen.

This is how we can place this image of sunglasses on someone's face and still see the area around of their face where the sunglasses lie - because these pixels in the sunglasses image have been made completely transparent.

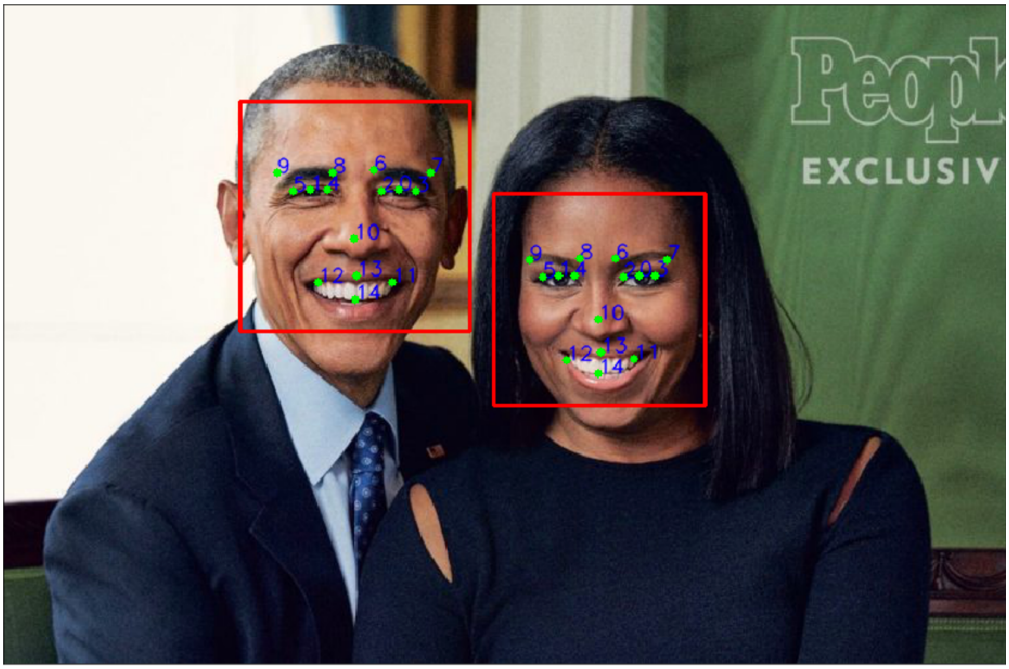
Lets check out the alpha channel of our sunglasses image in the next Python cell. Note because many of the pixels near the boundary are transparent we'll need to explicitly print out non-zero values if we want to see them.

In [94]:
```
# Print out the sunglasses transparency (alpha) channel
alpha_channel = sunglasses[:,:,3]
print ('the alpha channel here looks like')
print (alpha_channel)

# Just to double check that there are indeed non-zero values
# Let's find and print out every value greater than zero
values = np.where(alpha_channel != 0)
print ('\n the non-zero values of the alpha channel look like')
print (values)
```

```
the alpha channel here looks like
[[0 0 0 ..., 0 0 0]
 [0 0 0 ..., 0 0 0]
 [0 0 0 ..., 0 0 0]
 ...,
 [0 0 0 ..., 0 0 0]
 [0 0 0 ..., 0 0 0]
 [0 0 0 ..., 0 0 0]]

 the non-zero values of the alpha channel look like
(array([  17,   17,   17, ..., 1109, 1109, 1109], dtype=int64), array([ 687,  6
88,  689, ..., 2376, 2377, 2378], dtype=int64))
```

This means that when we place this sunglasses image on top of another image, we can use the transparency channel as a filter to tell us which pixels to overlay on a new image (only the non-transparent ones with values greater than zero).

One last thing: it's helpful to understand which keypoint belongs to the eyes, mouth, etc. So, in the image below, we also display the index of each facial keypoint directly on the image so that you can tell which keypoints are for the eyes, eyebrows, etc.



With this information, you're well on your way to completing this filtering task! See if you can place the sunglasses automatically on the individuals in the image loaded in / shown in the next Python cell.

In [95]:
```
# 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)


# Plot the image
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[95]: <matplotlib.image.AxesImage at 0x1139c24bcc0>

In [ ]:
```
## (Optional) TODO: Use the face detection code we saw in Section 1 with your tra
## sunglasses on the individuals in our test image
```

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

Now you can add the sunglasses filter 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 adding sunglasses to someone's face in the previous optional exercise and you should be good to go!

In [ ]:
```
import cv2
import time
from keras.models import load_model
import numpy as np

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

    # Keep video stream open
    while rval:
        # 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()

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

        # Read next frame
        time.sleep(0.05)             # control framerate for computation - defaul
        rval, frame = vc.read()
```

In [ ]:
```
# Load facial landmark detector model
model = load_model('my_model.h5')

# Run sunglasses painter
laptop_camera_go()
```