https://towardsdatascience.com/detecting-facial-features-using-deep-learning-2e23c8660a7a

Peter Skvarenina

Detecting facial features using Deep Learning

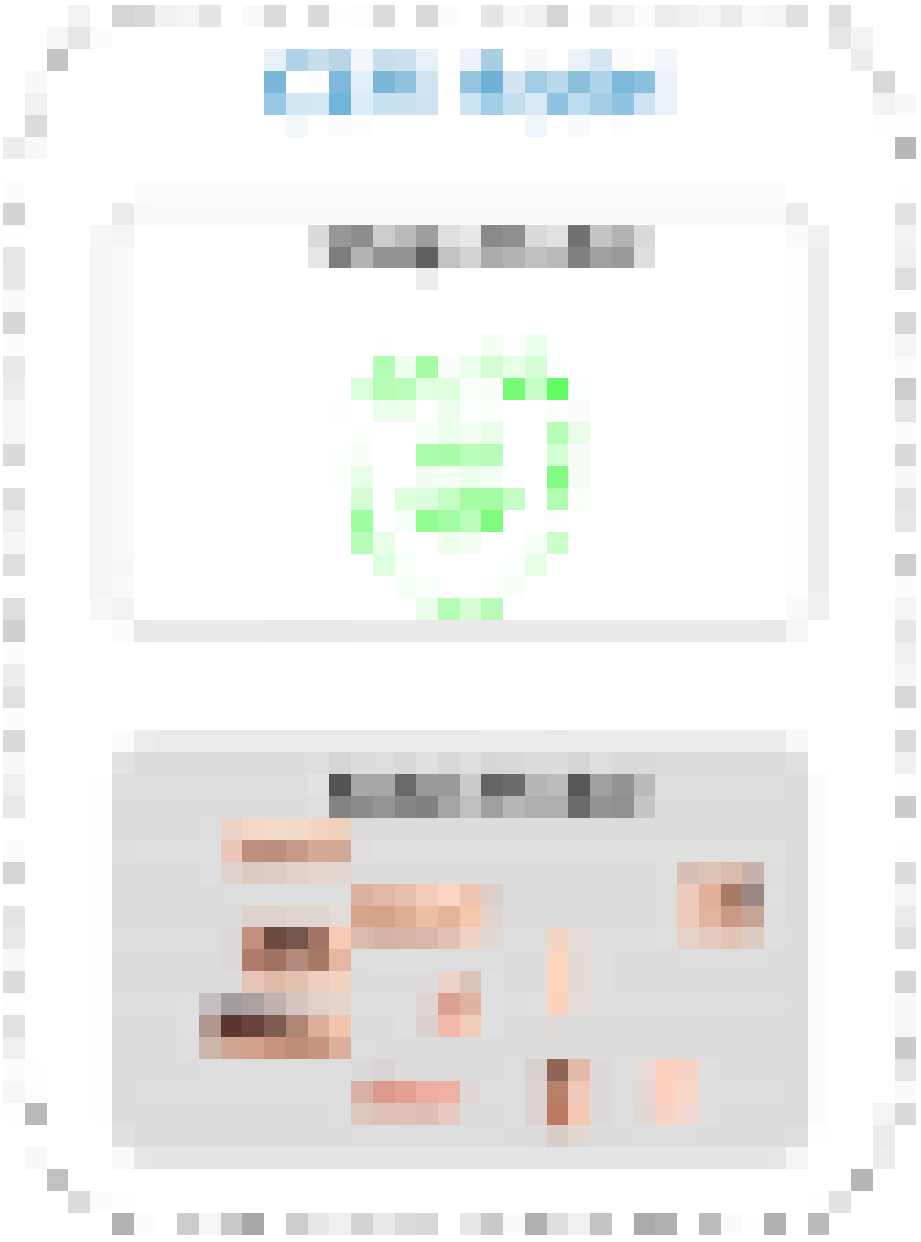
Peter Skvarenina
Follow

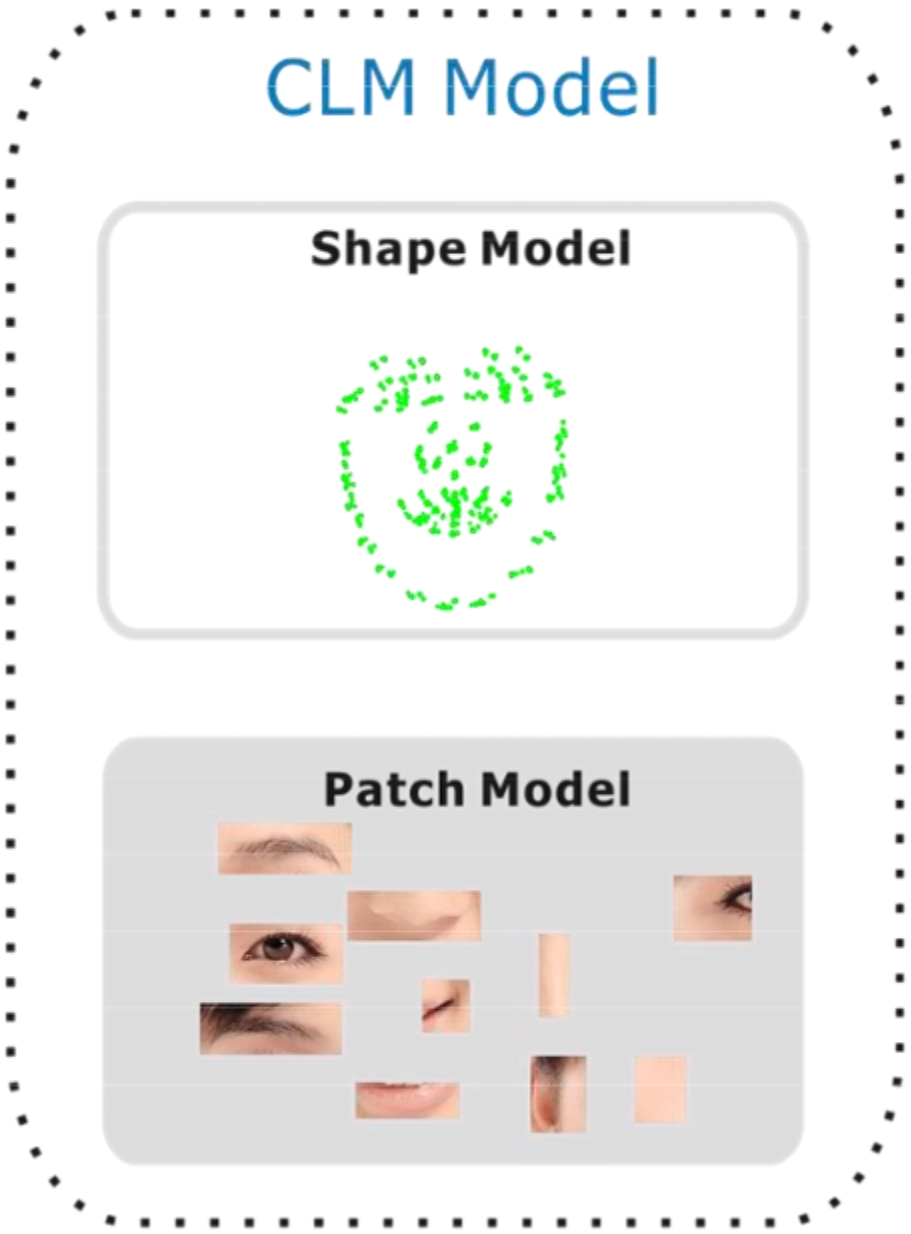
Aug 2, 2017 · 6 min read

Maybe you were wondering how you can place funny objects on faces in real-time video chats or detect emotions? I'll show you one possible approach here utilizing deep learning as well as skim over one older approach.

A challenging task in the past was detection of faces and their features like eyes, nose, mouth and even deriving emotions from their shapes. This task can be now “magically” solved by deep learning and any talented teenager can do it in a few hours. I will show you such an approach in this post.

“Classical” method (CLM)

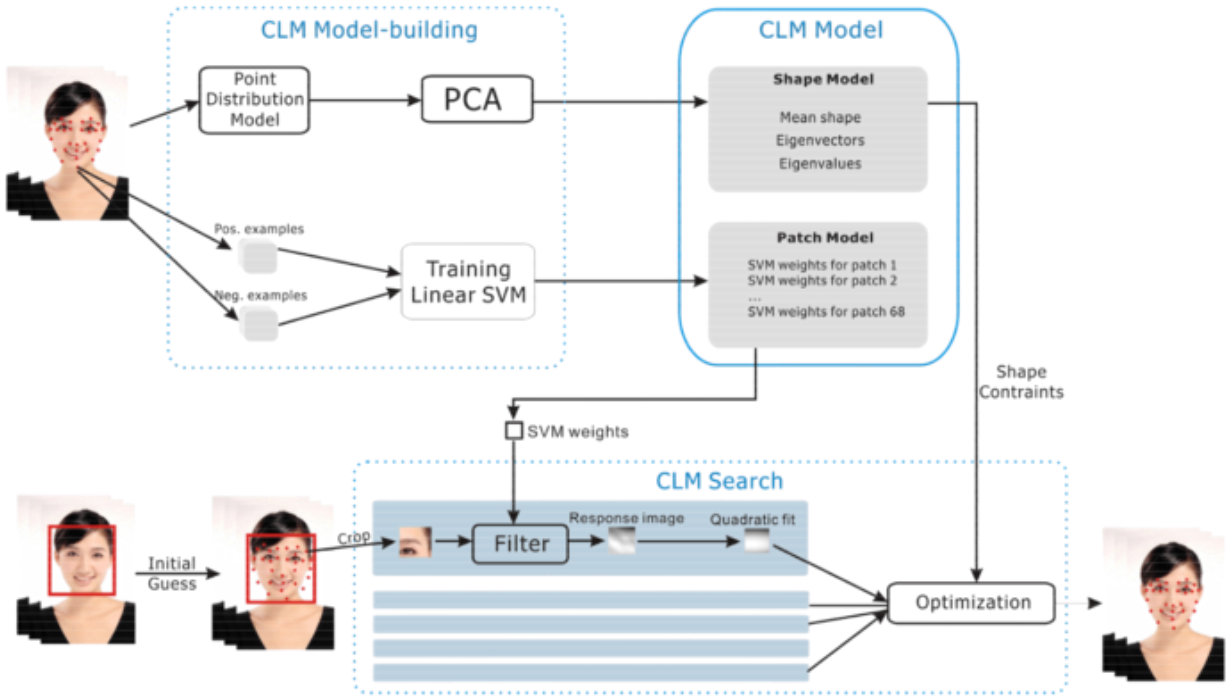




If you were like me and had a need to perform face tracking (in my case to transfer one's gestures from a web camera to an animated character), you probably found out that one of the best performing algorithms used to be Constrained Local Model (CLM), as implemented by e.g. Cambridge Face Trackeror its newer OpenFaceincarnation. This is based on splitting the detection tasks into detecting shape vector features (ASM) and

patch image templates (AAM), and refining the detection using pre-trained linear SVM.

It works by roughly estimating key-point positions first, then applying SVM with pre-trained images containing parts of face and adjusting key-point positions. This is repeated until error is sufficiently low for our purposes. Also, worth mentioning that it assumes the position of face on the image was estimated already, e.g. by using Viola-Jones detector (Haar cascades). The CLM process is however pretty involved and non-trivial, and definitely won't be implemented by a high-school wizard. You can see the overall architecture here:



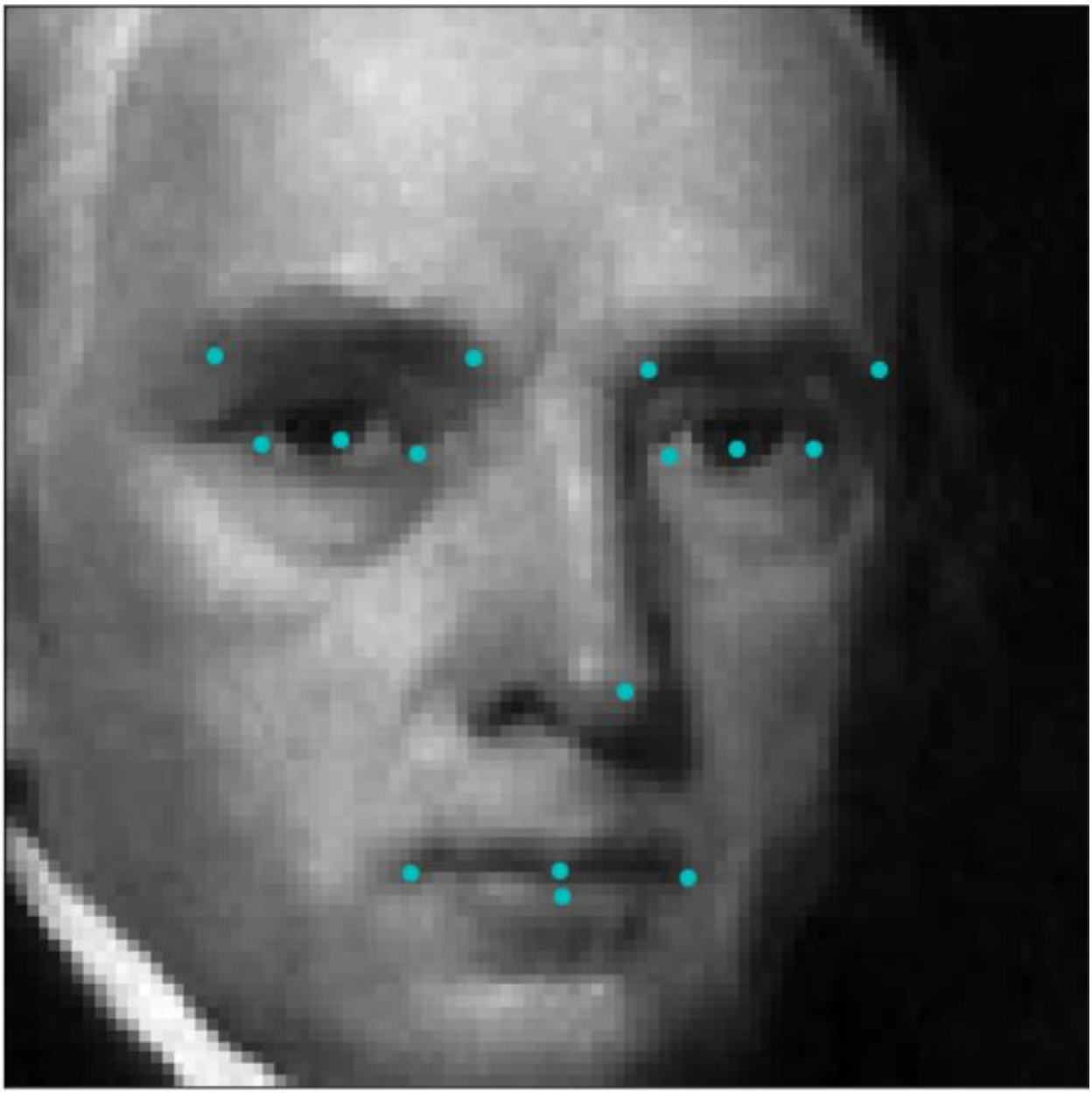
Well, fairly complicated, right?

Deep Learning

Instead, we can use a very simple convolutional neural network (CNN) and perform detection of key-points on parts of images we expect to contain faces. For that we need to have a training dataset though; we can use the one provided by Kaggle for their facial key-points detection challenge, containing 15 key-points, or a more complex MUCT dataset with 76 key-points (Yay!).

Obviously, having quality training datasets is essential here and we all should commemorate poor undergrad students that had to sacrifice their time and effort to annotate bunch of faces to be allowed to graduate, so that we can perform these magical tricks easily.

Here is how a sample baroque face and its key-points in the Kaggle dataset would look like:



James Madison Jr.

The dataset contains greyscale images with 96x96 resolution and 15 key-points, 5 per each eye and 5 for mouth/nose positions.

For an arbitrary image, we first need to detect where in the image the faces are; aforementioned Viola-Jones detector based on Haar cascades can be used (and if you look at how it works, it resembles CNNs a bit). Or if you are more adventurous, you can use Fully

Convolutional Networks (FCN) as well and perform image segmentation with depth estimation.

OpenCV does the trick

Anyway, this is a piece of cake using OpenCV:

```
Grayscale_image = cv2.cvtColor(image,
cv2.COLOR_RGB2GRAY)face_cascade =
cv2.CascadeClassifier('haarcascade_frontalface_default.xml')boundin
g_boxes = face_cascade.detectMultiScale(grayscale_image, 1.25, 6)
```

This chunk of code returns all possible face bounding boxes on an image.

Next for each bounding box returned by Viola-Jones we extract the corresponding sub-images, convert them to greyscale and resize them to 96x96. And they will become input to our finished CNN for inference.

The CNN architecture is super trivial; a bunch of 5x5 convolutional layers (3 in fact, with 24, 36 and 48 filters each), then 2 more 3x3 convolutional layers (64 filters each) and 3 fully connected layers (with 500, 90 and 30 nodes) in the end. Some max pooling to prevent overfitting and global average pooling to reduce number of flatten parameters. The output will be 30 floating point numbers denoting a sequence of x, y coordinates for each of 15 key-points.

Here is the implementation in Keras:

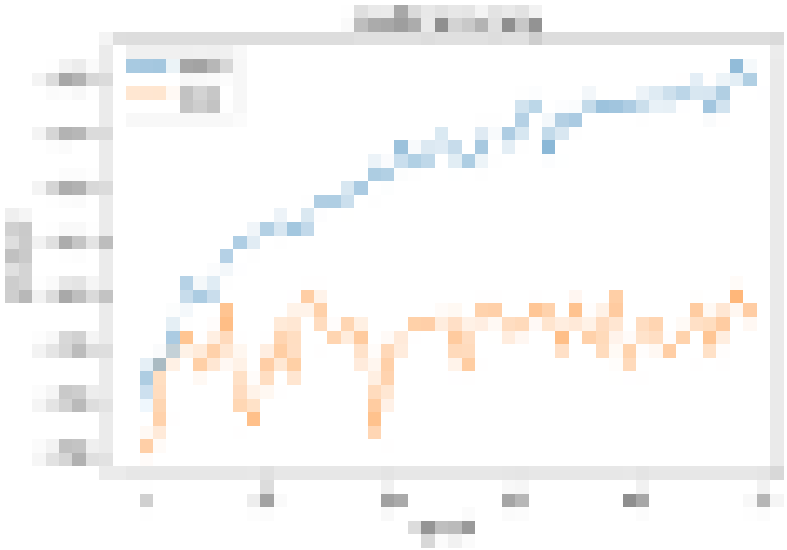
```
model = Sequential()model.add(BatchNormalization(input_shape=(96,
96, 1)))model.add(Convolution2D(24, 5, 5, border_mode="same",
init='he_normal', input_shape=(96, 96, 1), dim_ordering="tf"))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2),
border_mode="valid"))model.add(Convolution2D(36, 5, 5))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2),
border_mode="valid"))model.add(Convolution2D(48, 5, 5))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2),
border_mode="valid"))model.add(Convolution2D(64, 3, 3))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2),
border_mode="valid"))model.add(Convolution2D(64, 3, 3))
```

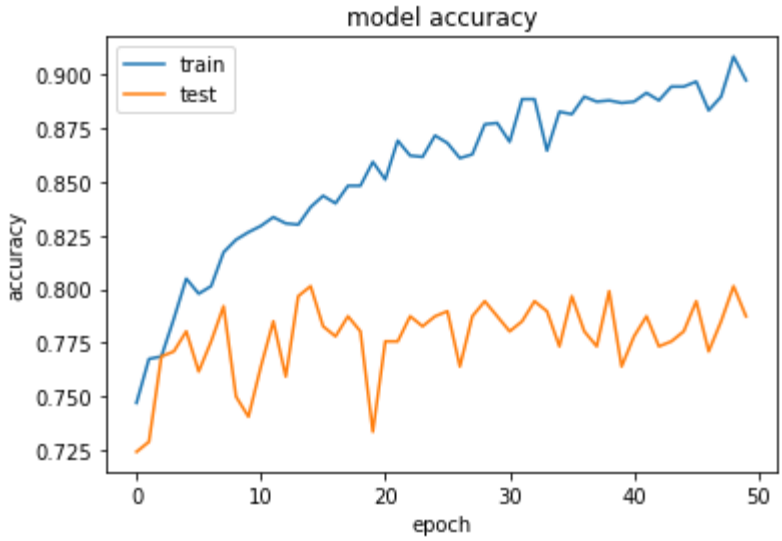
```
model.add(Activation("relu"))model.add(GlobalAveragePooling2D());mo
del.add(Dense(500, activation="relu"))
model.add(Dense(90, activation="relu"))
model.add(Dense(30))
```

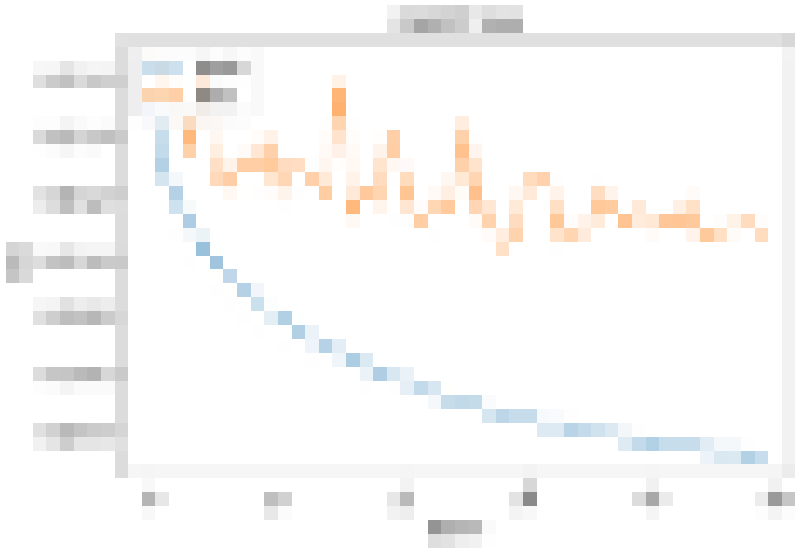
You might want to choose Root Mean Square Propagation (rmsprop) optimizer and Mean Squared Error (MSE) as your loss function and accuracy metrics.

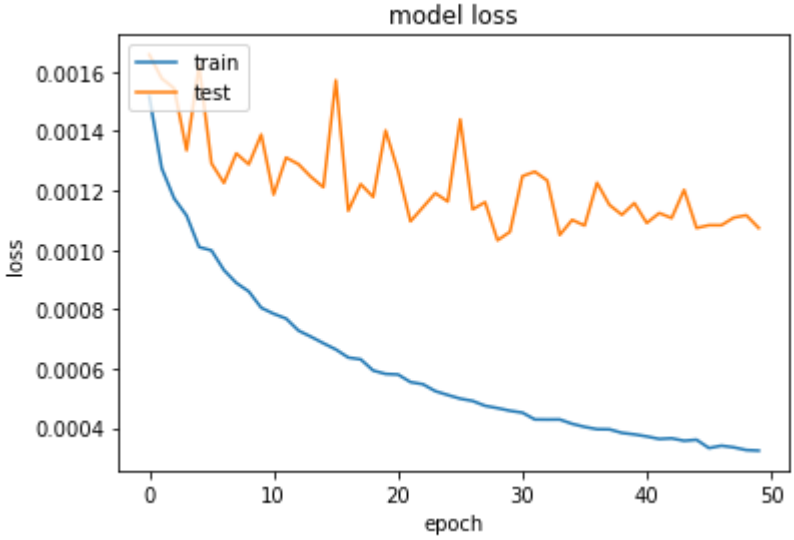
Just by some trivial tricks like batch normalization on input images, global average pooling as well as HE normal weight initialization you can get 80–90% validation accuracy and loss < 0.001 at around 30 training epochs:

```
model.compile(optimizer='rmsprop', loss='mse',
metrics=['accuracy'])checkpointer =
ModelCheckpoint(filepath='face_model.h5', verbose=1,
save_best_only=True)epochs = 30hist = model.fit(X_train, y_train,
validation_split=0.2, shuffle=True, epochs=epochs, batch_size=20,
callbacks=[checkpointer], verbose=1)
```




model accuracy
train
test
accuracy
epoch
0.900
0.875
0.850
0.825
0.800
0.775
0.750
0.725
0
10
20
30
40
50



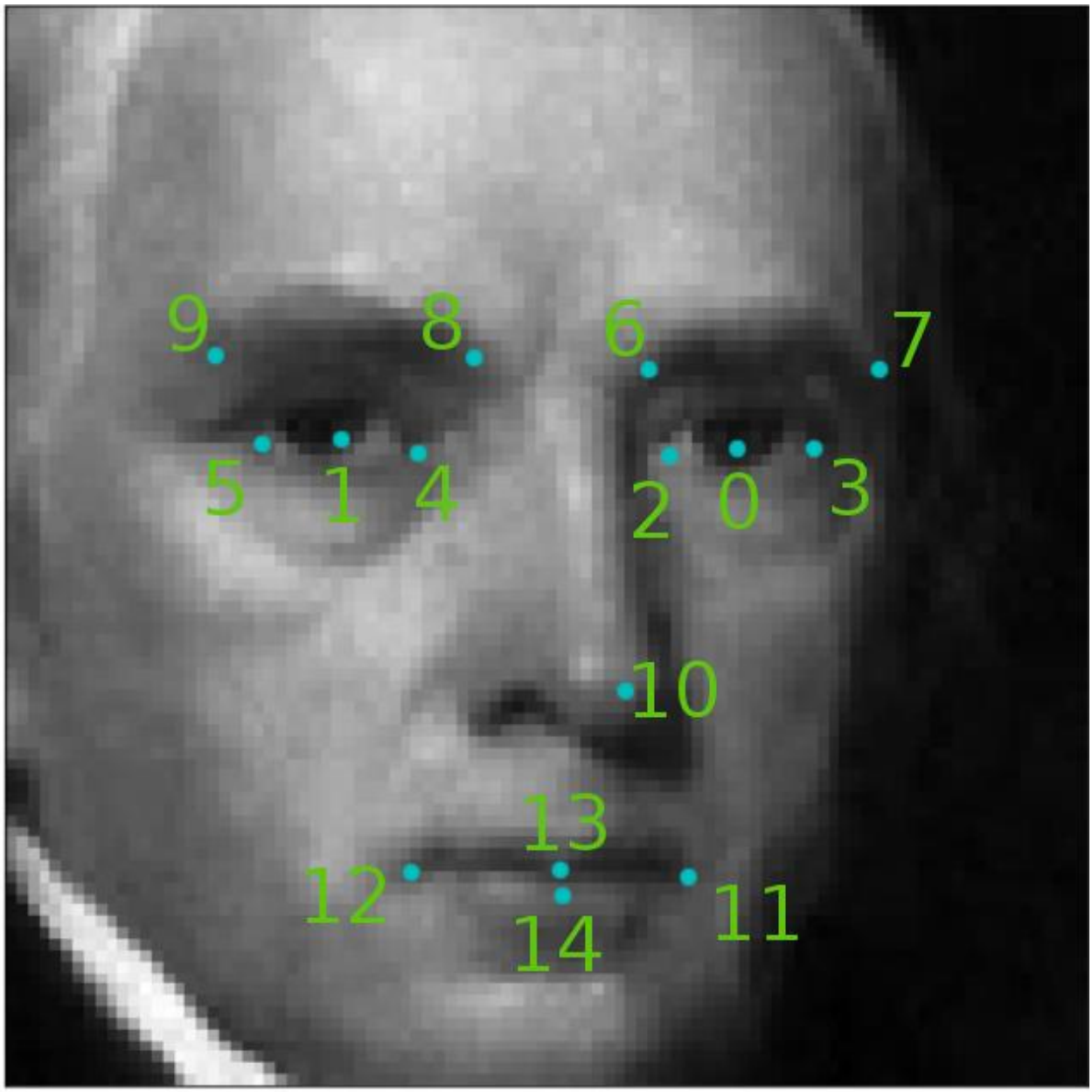

model loss
train
test
loss
epoch
0.0016
0.0014
0.0012
0.0010
0.0008
0.0006
0.0004
0
10
20
30
40
50

Then to predict key-points location simply run:

```
features = model.predict(region, batch_size=1)
```

aaand that's it! You have now mastered the art of detecting facial key-points!

Mind you, the result of your prediction will be 15 pairs of x, y coordinates for each key-point, in the order as depicted on the following picture:



Saving you some time…

If you want to do better, you might want to do some additional homework:

- experiment how you can reduce number of convolutional layers and filter sizes while retaining accuracy and improving inference speed

- replace convolutional part with transfer learning (Xception is my favorite)
- use a more detailed dataset
- do some advanced image augmentation to improve robustness

You might find all of this way too easy; if you want a challenge, move to 3D and look at how Facebook and NVidia are tracking faces.

Obviously, you can use this newly learned magic to perform some non-trivial things you might have always wanted to do but didn't know how:

- place annoying objects on faces during video chat such as sunglasses, weird caps, mustaches etc.
- swap faces between friends, enemies, animals and objects
- contribute to vanity of human population by allowing testing new hair-styles, jewelry or make-up on selfie real-time videos
- detect if your employee is too drunk (or insufficiently drunk) to perform assigned tasks
- identify prevailing emotion in people in your video feed if you want to automate emotional processing for whatever weird reason
- use real-time face-to-cartoon conversion using GANs and bend the cartoon faces according to your own face on a web-cam to mimic your movement, talk and emotions

So now you know how to make your own awesome video chat filter! How about writing one right now?