Julien VIEILLEVIGNE

DAI – Promo 2024

Filters

I – Interpretation of Filters

1) The filter visualization algorithm through activation maximization is an analytical technique primarily used in CNN models, which are in turn mostly used for computer vision. The goal is to interpret the functioning of our CNN to make it more transparent and interpretable by visualizing the features that our model has learned to recognize.

   To do this, we must maximize the activations to obtain information about the patterns or features to which each neuron is sensitive. For reference, activations in ML are the outputs of neurons in a neural network.

   The information obtained can include edges/corners, textures, patterns, parts of objects, or entire objects. The deeper you go into the layers, the more complex the patterns detected become.

   In the link shared in the project statement, the first step was to define the dimension of the image and the convolutional layer we wish to observe.

   Then we retrieve our model pre-trained on the ImageNet dataset with weights that are suitable for image classification. It's our starting point because it's a model that has already learned to recognize certain patterns and features that we want to visualize.

   The next step is to process the gradient ascent process by defining several functions like the compute_loss function, which aims to maximize the loss by doing the mean of the activation of a specific filter in our target layer.
   Then the gradient ascent function will calculate the gradients of the loss above regarding the input image and update the image to move it toward a state that will activate the target filter more strongly. It iteratively updated the image to maximize filter activation.

   The next step is to set up the end-to-end filter visualization loop by starting with an image close to all grays. We apply the previously defined gradient ascent function and convert the resulting input image back to a displayable form, by normalizing it, center-cropping it, and restricting it to the [0, 255] range.

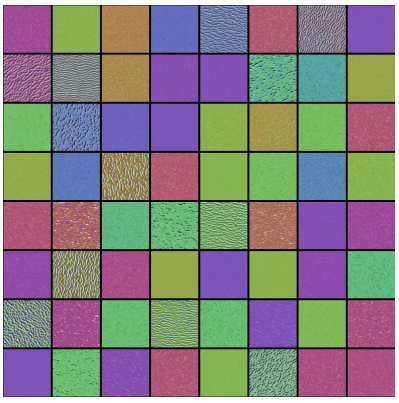
   This will ultimately allow us to visualize an 8x8 grid of the first 64 filters in the target layers to get a feel for the range of different visual patterns that the model has learned.

2) Using VGG16 from the Keras-VGGFace library, we were able to analyze the structure of the model we used to visualize the filters of the different layers.

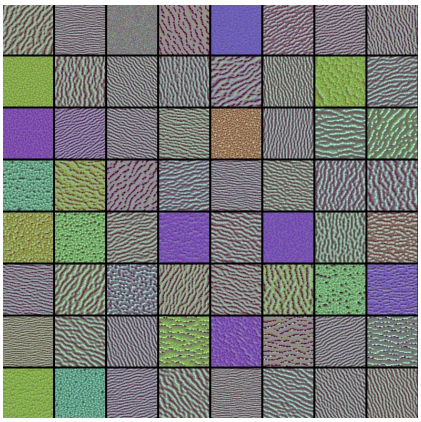
```
Model: "vggface_vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         (None, None, None, 3)     0
_________________________________________________________________
conv1_1 (Conv2D)             (None, None, None, 64)    1792
_________________________________________________________________
conv1_2 (Conv2D)             (None, None, None, 64)    36928
_________________________________________________________________
pool1 (MaxPooling2D)         (None, None, None, 64)    0
_________________________________________________________________
conv2_1 (Conv2D)             (None, None, None, 128)   73856
_________________________________________________________________
conv2_2 (Conv2D)             (None, None, None, 128)   147584
_________________________________________________________________
pool2 (MaxPooling2D)         (None, None, None, 128)   0
_________________________________________________________________
conv3_1 (Conv2D)             (None, None, None, 256)   295168
_________________________________________________________________
conv3_2 (Conv2D)             (None, None, None, 256)   590080
_________________________________________________________________
conv3_3 (Conv2D)             (None, None, None, 256)   590080
_________________________________________________________________
pool3 (MaxPooling2D)         (None, None, None, 256)   0
_________________________________________________________________
conv4_1 (Conv2D)             (None, None, None, 512)   1180160
_________________________________________________________________
conv4_2 (Conv2D)             (None, None, None, 512)   2359808
_________________________________________________________________
conv4_3 (Conv2D)             (None, None, None, 512)   2359808
_________________________________________________________________
pool4 (MaxPooling2D)         (None, None, None, 512)   0
_________________________________________________________________
conv5_1 (Conv2D)             (None, None, None, 512)   2359808
_________________________________________________________________
conv5_2 (Conv2D)             (None, None, None, 512)   2359808
_________________________________________________________________
conv5_3 (Conv2D)             (None, None, None, 512)   2359808
_________________________________________________________________
pool5 (MaxPooling2D)         (None, None, None, 512)   0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

Here we can see that there are 5 convolutional blocks. In each block, there are two to three convolutional layers followed by a MaxPooling2D layer that reduces the dimensionality of the data and extracts the most important features. The convolutional layers aim to find patterns and features (extract the important characteristics) that will allow us to have the most performant model possible.
We visualized the last convolutional layer of each block.

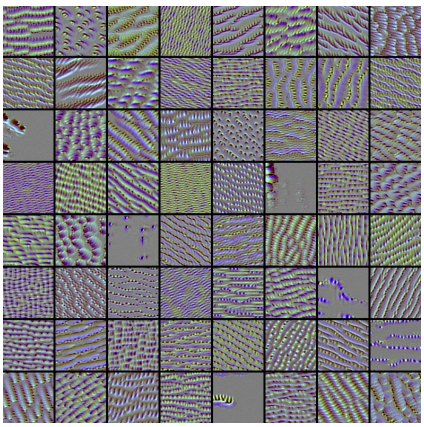
Conv1_2:



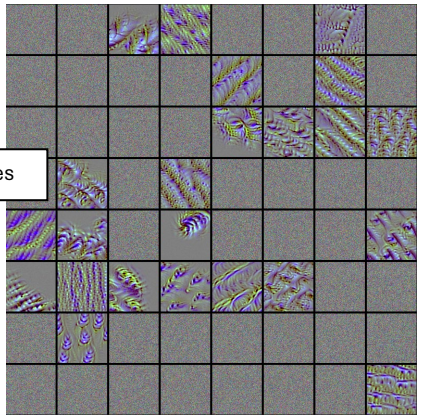
Conv2_2:



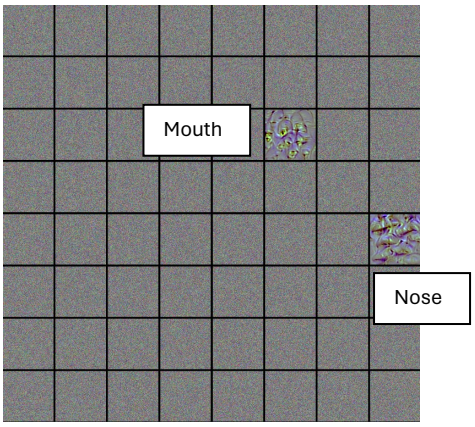
Conv3_3:



Conv4_3:



Conv5_3:



As we progress through the layers, we visualize increasingly complex patterns. In the first observed layer, our model detects simple things such as edges, colors, and textures.

In the second and third layers, we see simple features combined in ways that more closely resemble facial patterns as we might imagine them.

Finally, in the fourth and fifth layers, we capture much more complex patterns. These patterns/representations are complex and can be abstract, but in our case, we can identify some elements. Looking more closely, we see mouths, eyes, and noses, and other patterns characteristic of the human face.

We can therefore conclude that our model is well trained and of good quality, as it identifies important facial features such as the eyes, nose, and mouth. As our image progresses through the convolutional layers, we can see increasingly precise and complex patterns representing the human face.

You will find screens of the adapted code below:

```
img_width = 180
img_height = 180
# Our target layer: we will visualize the filters from this layer.
# See `model.summary()` for list of layer names, if you want to change this.
layer_name = "conv4_3"

layer = model.get_layer(name=layer_name)
feature_extractor = keras.Model(inputs=model.inputs, outputs=layer.output)
```

✓ 0.0s

```
def compute_loss(input_image, filter_index):
    activation = feature_extractor(input_image)
    # We avoid border artifacts by only involving non-border pixels in the loss.
    filter_activation = activation[:, 2:-2, 2:-2, filter_index]
    return tf.reduce_mean(filter_activation)

@tf.function
def gradient_ascent_step(img, filter_index, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(img)
        loss = compute_loss(img, filter_index)
    # Compute gradients.
    grads = tape.gradient(loss, img)
    # Normalize gradients.
    grads = tf.math.l2_normalize(grads)
    img += learning_rate * grads
    return loss, img

def initialize_image():
    # We start from a gray image with some random noise
    img = tf.random.uniform((1, img_width, img_height, 3))
    # ResNet50V2 expects inputs in the range [-1, +1].
    # Here we scale our random inputs to [-0.125, +0.125]
    return (img - 0.5) * 0.25

def visualize_filter(filter_index):
    # We run gradient ascent for 20 steps
    iterations = 30
    learning_rate = 10.0
    img = initialize_image()
    for iteration in range(iterations):
        loss, img = gradient_ascent_step(img, filter_index, learning_rate)

    # Decode the resulting input image
    img = deprocess_image(img[0].numpy())
    return loss, img
```

```python
def deprocess_image(img):
    # Normalize array: center on 0., ensure variance is 0.15
    img -= img.mean()
    img /= img.std() + 1e-5
    img *= 0.15

    # Center crop
    img = img[25:-25, 25:-25, :]

    # Clip to [0, 1]
    img += 0.5
    img = np.clip(img, 0, 1)

    # Convert to RGB array
    img *= 255
    img = np.clip(img, 0, 255).astype("uint8")
    return img
```

✓ 0.4s

```python
# Compute image inputs that maximize per-filter activations
# for the first 64 filters of our target layer
all_imgs = []
for filter_index in range(64):
    #print("Processing filter %d" % (filter_index,))
    loss, img = visualize_filter(filter_index)
    all_imgs.append(img)

# Build a black picture with enough space for
# our 8 x 8 filters of size 128 x 128, with a 5px margin in between
margin = 5
n = 8
cropped_width = img_width - 25 * 2
cropped_height = img_height - 25 * 2
width = n * cropped_width + (n - 1) * margin
height = n * cropped_height + (n - 1) * margin
stitched_filters = np.zeros((width, height, 3))

# Fill the picture with our saved filters
for i in range(n):
    for j in range(n):
        img = all_imgs[i * n + j]
        stitched_filters[
            (cropped_width + margin) * i : (cropped_width + margin) * i + cropped_width,
            (cropped_height + margin) * j : (cropped_height + margin) * j
            + cropped_height,
            :,
        ] = img
save_img("stiched_filters_4_3.png", stitched_filters)


display(Image("stiched_filters_4_3.png"))
```

✓ 35m 50.9s

II – Grad-CAM and Occluding Parts of the Image

1) Grad-CAM stands for Gradient-weighted Class Activation Mapping, and it is an algorithm for interpreting Convolutional Neural Network (CNN) models in computer vision. Its objective is to highlight the part of the image that influences the model's final decision. Thus, it's a crucial algorithm for explaining the predictions of our model.
Grad-CAM is a qualitative algorithm that allows for the visualization of CNN decision-making processes in computer vision.

   The algorithm uses the gradient to produce a heatmap on the image, localizing and highlighting the important regions of the image that contributed to the prediction.

   Here are the steps of the algorithm:

   The first step is to configure parameters such as our CNN model, the dimension of our image, our image, and the last convolutional layer.

   Next, we proceed to the Grad-CAM algorithm with the creation of the function `get_img_array`, which takes an image, changes its dimension, and the function `make_gradcam_heatmap`.
This function creates a model that maps the input image to the activations of the last conv layer as well as the output prediction.

   Then, we compute the gradient of the top predicted class for our input image with respect to the activations of the last conv layer.

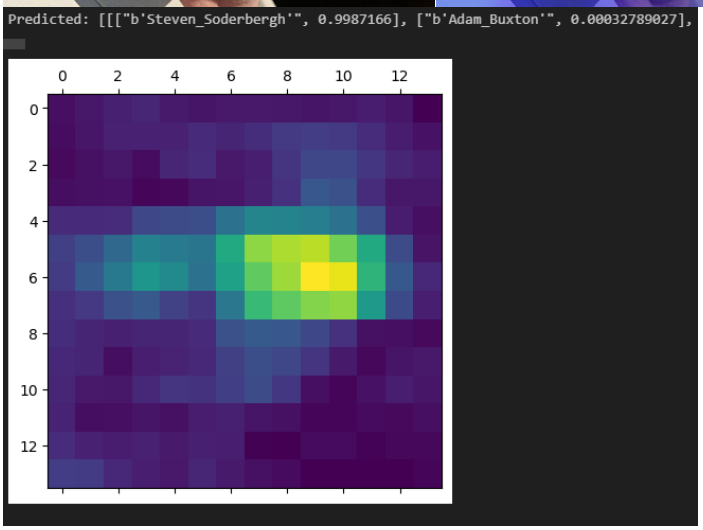
   Finally, we recover the gradient of the output neuron concerning the output feature map of the last conv layer. Then we create a vector where each entry is the average of the gradient intensity over a specific feature map channel.

   Ultimately, we multiply each channel in the feature map array by "how important this channel is" regarding the top predicted class then sum all the channels to obtain the class activation heatmap. And we finish by returning this heatmap normalized between 0 and 1.

   With this heatmap, we can overlay it with our initial image to see more precisely the part of the image that activated our neurons.

2) We use VGG16 for face recognition to identify personalities through their faces. To visualize the part of the image that localizes and highlights the important regions contributing to the prediction, we have decided to take the last convolutional layer of the last block, which detects more complex patterns.

Here is the photo of Steven Soderbergh that we provided along with the results obtained:



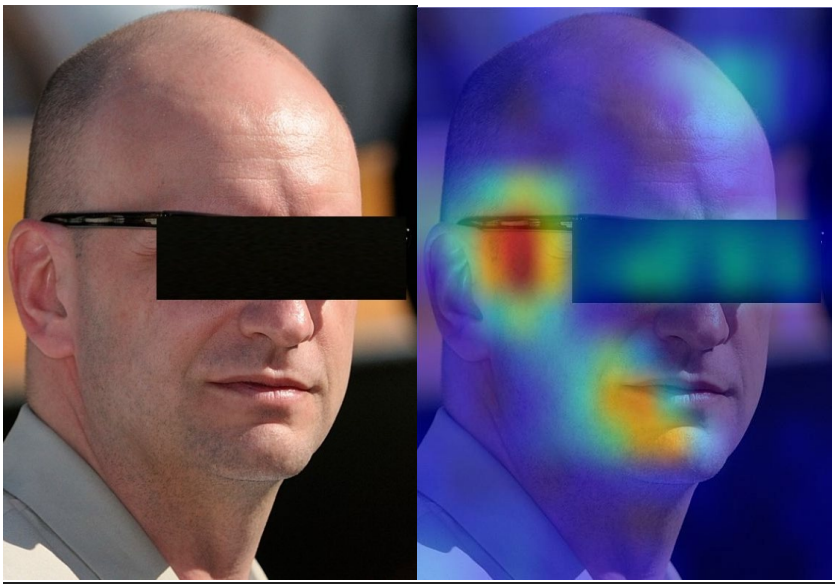
Our model predicts John Hodgman very well, with over 99% confidence. The targeted localization is on Steven Soderbergh's eyes and glasses.

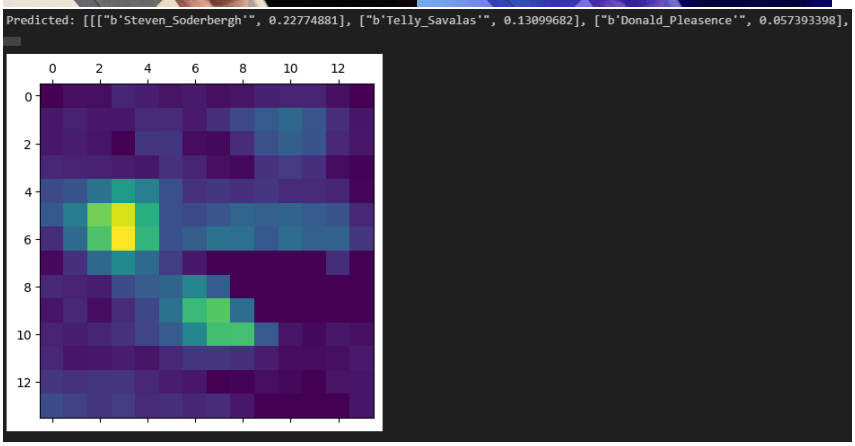
Interpretability is therefore useful in this context because it allows us to understand what our model sees to justify its prediction. With this analysis, it would be interesting to test our model on Steven masking their glasses/eyes to see if our model can still predict it.

It would also be interesting to train our model on steven's masked eyes to help it detect other features of steven's face.

3) We repeated the same experiment with Grad-CAM by applying a black mask over Steven Soderbergh's eyes to see if this could anonymize him.

Here are our results:



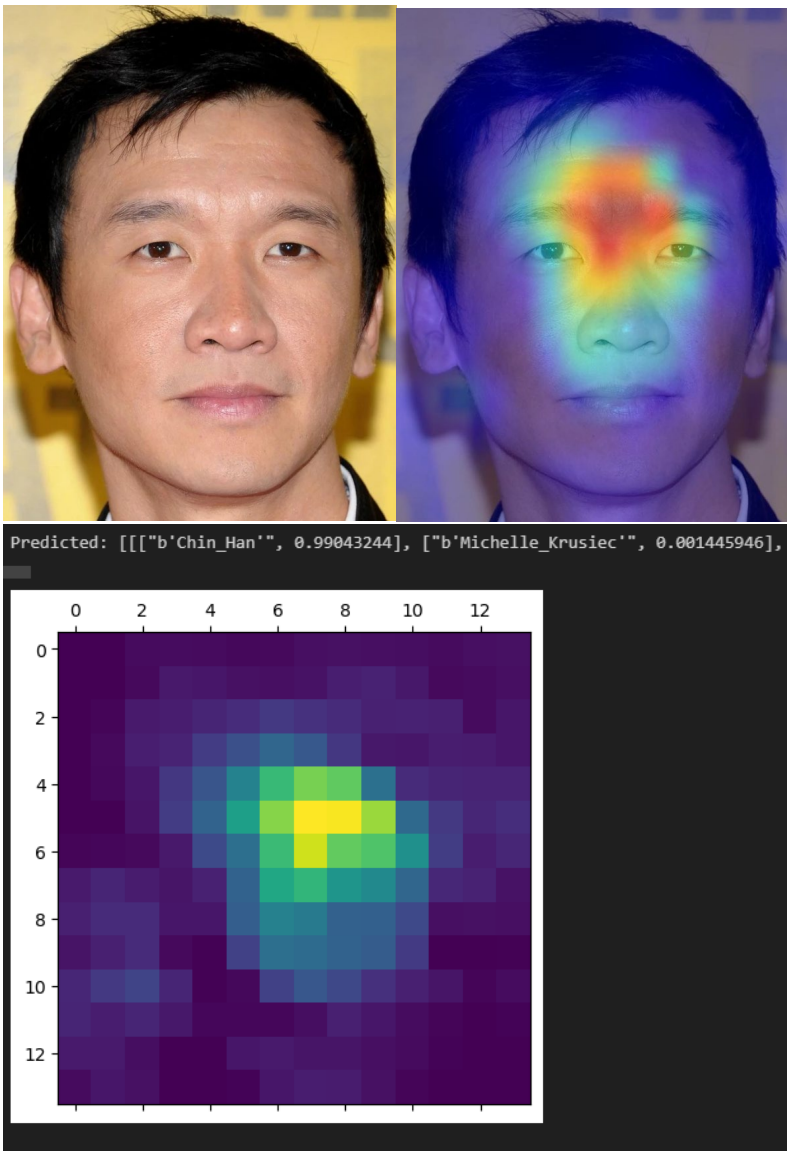


What we can see is that the neural network no longer focuses solely on the eye region but on two areas. The first area is his temple, where the arm of his glasses is located (glasses being an important and characteristic element of Steven Soderbergh's face), and on the side of his mouth. Thus, the network no longer focuses only on the eye region and shifts its focus to two new regions, even if our model somewhat activates over the location of the black mask, which is odd because it provides no information.

Regarding the prediction of our model, it continues to predict Steven Soderbergh but with a percentage of 23%, which is significantly lower than the previous prediction. It also predicted Telly Savalas with a percentage of 13%, which is quite realistic because the two personalities look very similar. Thus, we understand the decrease in scores because we are reducing the amount of information available with the mask, but this mask does not allow for anonymization.

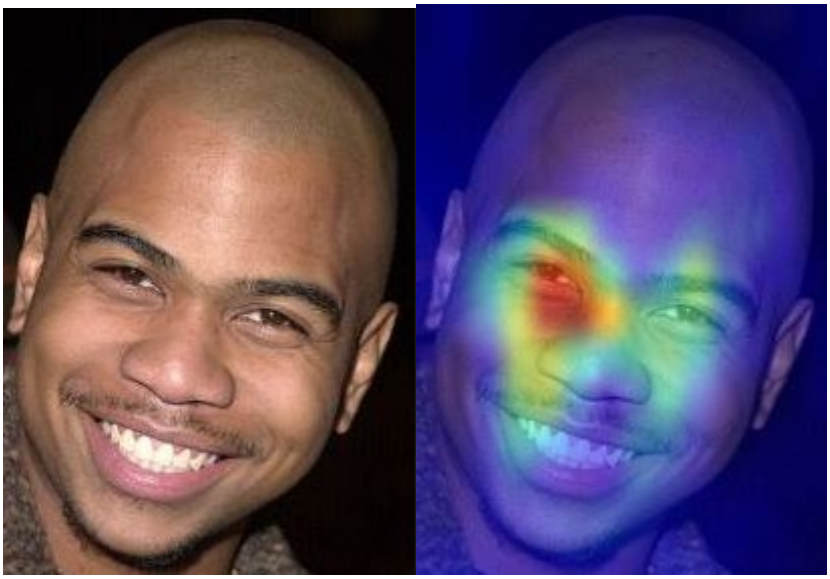
Always using the last convolutional layer of the model, we conducted the Grad-CAM for the Asian personality Chin Han and the Black personality Omar Gooding.

Here are the results obtained for Chin Han:

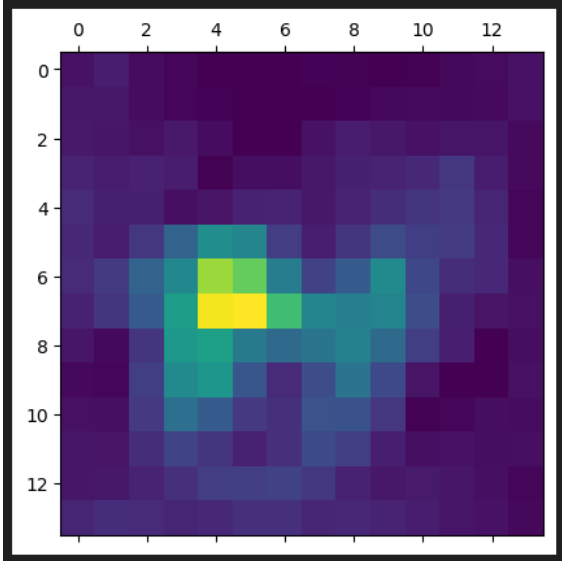


Our model predicts the correct person with 99% accuracy, and it is the area between the eyebrows that is predominantly activated.

Here are the results obtained for Omar Gooding:



```
Predicted: [[["b'Omar_Gooding'", 0.82803786], ["b'Tawny_Cypress'", 0.0064026727],
```



Our model predicts the correct person with 83% accuracy, and it is the area between the nose and the eye that is predominantly activated.

What we can conclude is that different areas are activated depending on the personalities. It is not from three images that we can draw definitive conclusions, but what happens is that our model has been primarily trained on Caucasian individuals and to a lesser extent on Asians and Blacks. There is underfitting in these populations.

This leads to a facial recognition bias based on ethnicity, which results in a different target on the face according to ethnicity. For example, for Caucasians, it's more the eyes that are activated, for Blacks, it's the nose and mouth, and for Asians, it's the chin.

Therefore, it would be interesting to place a black mask over the eyes to train our model to be less activated on Caucasian eyes. Moreover, it would be interesting to create a synthetic mask to replace the ethnic specificities on the eyes but the best solution would be to have a balanced dataset between ethnicity.