Conservatoire National des Arts et Métiers
UE RCP209

Gilles Augustins
03/07/2017

Image Style Transfer Using Convolutional Neural Networks

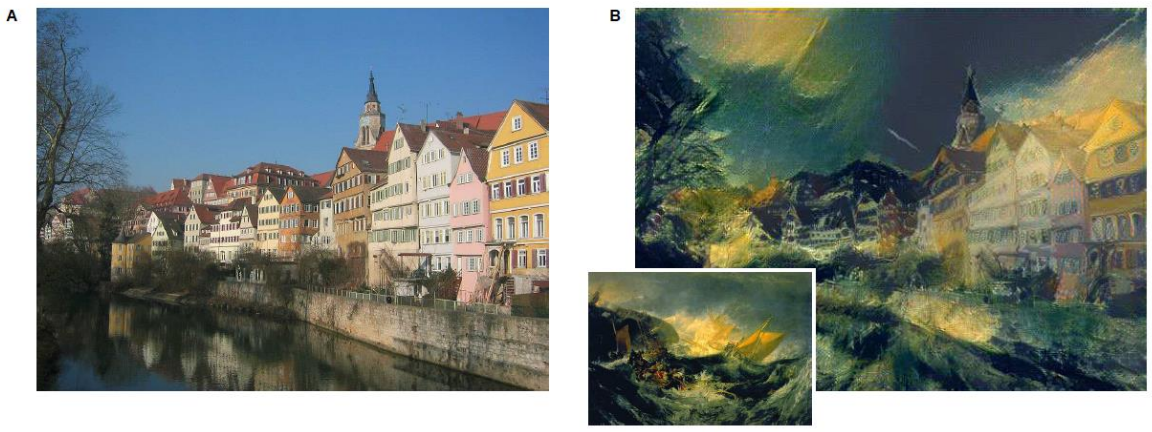
Project Description

The internal representations of Convolutional Neural Networks have demonstrated their ability to capture semantic visual information. It is possible to use that semantic space to transfer the style of a painting onto pictures or photographs.

Leon Gatys, Alexander Ecker and Matthias Betge, from the Centre for Integrative Neuroscience, have developed an Neural algorithm that is extracting semantic features of picture and artistic style, and recombining them to create a new picture.

http://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Gatys_Image_Style_Transfer_CVPR_2016_paper.pdf

The results achieved by this team look as follow:



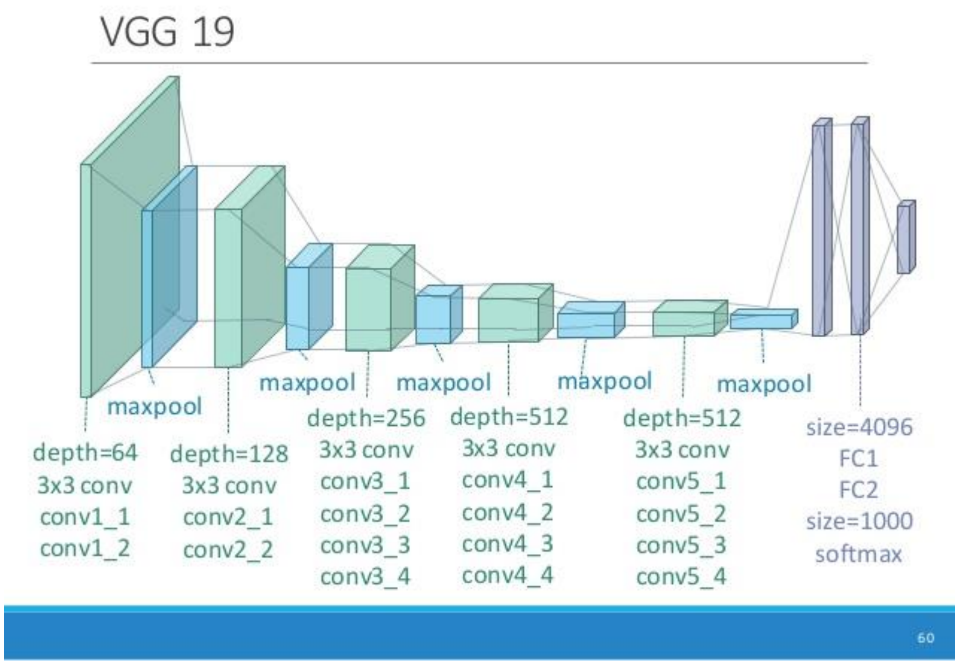
The purpose of this project is to analyze the paper and implement and test the algorithm described.

We have chosen here the Keras frontend / TensorFlow backend. The code is developed in the Python environment.

Description of the Neural Algorithm Used to Perform Style Transfer

Deep Images Representations with VGG Network

We use here a 19-layer VGG network, comprising 16 convolutional and 5 pooling layers, trained on the ImageNet database. The global idea of the method is to use the fact that, deeper are the layers of the network, the more of a semantic nature are the features they extract from picture and style.



In this algorithm, we don't use the fully connected layers, as we are only interested in the representations generated by the filters from the convolutional layers.

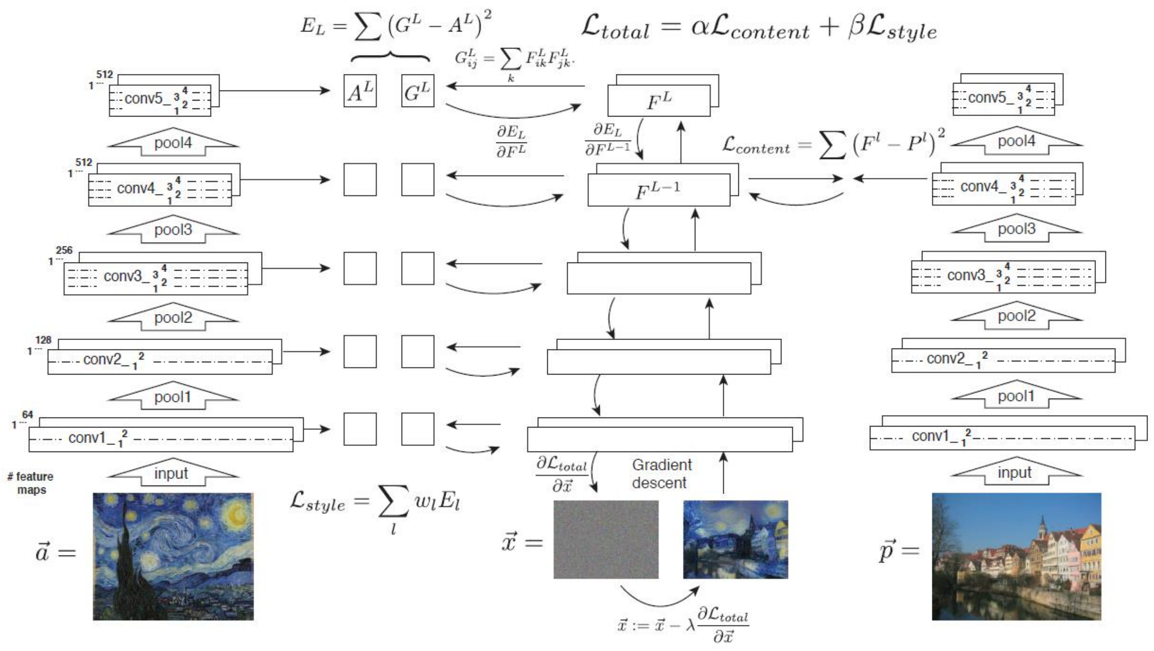
Global Overview of the Algorithm

The global algorithm is as follow:

1- Extract the <u>content features</u> of the picture $\vec{p}$ on which we want to transfer the style. This is typically the response of deep filters such as conv4_2 when we feed the network with the picture $\vec{p}$ as input.
2- Extract the <u>style features</u> of the style $\vec{a}$ we want to apply. Here we compute the responses of filters at different depths in the network when feeding the network with $\vec{a}$, as we want to extract both high-levels and low-levels features of the style.
3- Feed the network with a white noise image $\vec{x}$ and compute its style and content features.
4- Compute the <u>style loss</u> : the loss of the style features of the white noise image $\vec{x}$ versus the style features of the style image $\vec{a}$

5- Compute the content loss : the loss of the content features of the white noise image $\vec{x}$ versus the content features of the style image $\vec{p}$
6- Compute the combination loss, as a linear combination of style loss and content loss
7- Compute the gradients of input image $\vec{x}$ with regard to this combination loss
8- Construct the hybrid image by Gradient Descent on the input image $\vec{x}$ while minimize the combination loss

A pretty good graphical representation is given in the paper, reproduced here:


$E_L = \sum (G^L - A^L)^2$
$\mathcal{L}_{total} = \alpha \mathcal{L}_{content} + \beta \mathcal{L}_{style}$
$G^L_{ij} = \sum_k F^L_{ik} F^L_{jk}$
A^L
G^L
F^L
F^{L-1}
$\frac{\partial E_L}{\partial F^L}$
$\frac{\partial E_L}{\partial F^{L-1}}$
$\mathcal{L}_{content} = \sum (F^l - P^l)^2$
conv5_1 2 3 4
pool4
conv4_1 2 3 4
pool3
conv3_1 2 3 4
pool2
conv2_1 2
pool1
conv1_1 2
input
512
256
128
64
# feature maps
$\mathcal{L}_{style} = \sum_l w_l E_l$
$\frac{\partial \mathcal{L}_{total}}{\partial \vec{x}}$
Gradient descent
$\vec{a} =$
$\vec{x} =$
$\vec{p} =$
$\vec{x} := \vec{x} - \lambda \frac{\partial \mathcal{L}_{total}}{\partial \vec{x}}$

Content Representation and loss

The right-hand side of the diagram illustrates the extraction of the content features of image $\vec{p}$. The content features are computed as the response of one layer of the conv4 convolutional block: l. The content loss $\mathcal{L}_{content}$ is the squared difference between the content features of $\vec{p}$ and $\vec{x}$:

$$\mathcal{L}_{content} = \sum \left(F^l - P^l\right)^2$$

Where F^l is the response of $\vec{x}$ and P^l is the response of $\vec{p}$ at layer l.

Style Representation and loss

The left-hand side of the diagram illustrates the extraction of the style features of image $\vec{a}$. The style feature at a given layer is the Gram matrix of the response of the filters at that layer, where F is the 2D vectorized representation of the filters:

$$A^l_{i,j} = \sum_k F^l_{ik} F^l_{jk} = \mathrm{FF^T}$$

The Gram matrix computes correlations between the different filters of a convolutional layer and thus gives a better representation of the style. The style features are computed for all convolutional blocks.

The style loss $\mathcal{L}_{style}$ is a weighted sum of the squared differences between the style features of $\vec{\boldsymbol{a}}$ and $\vec{\boldsymbol{x}}$ at each convolutional block.

$$\mathcal{L}_{style} = \sum_{l} \sum w_l \left(G^l - A^l\right)^2$$

Where G^l and A^l are the Gram matrix of the responses of $\vec{\boldsymbol{x}}$ and $\vec{\boldsymbol{a}}$ at layer l.

Style Transfer

The middle part of the diagram illustrates the synthesis of combination image. The total loss is the linear combination of the style loss and content loss.

Implementation Methodology

Keras / TensorFlow

We have implemented the algorithm with Keras, using the TensorFlow backend, in Python. Most of the functions have been developed using the backend, as Keras frontend functions are not flexible enough for the tasks we undertake: typically, retrieving filter responses at specific layers, defining custom losses, and calculating gradients with regard to custom losses is not possible with the frontend.

Methodology

We have implemented the algorithm in 3 steps

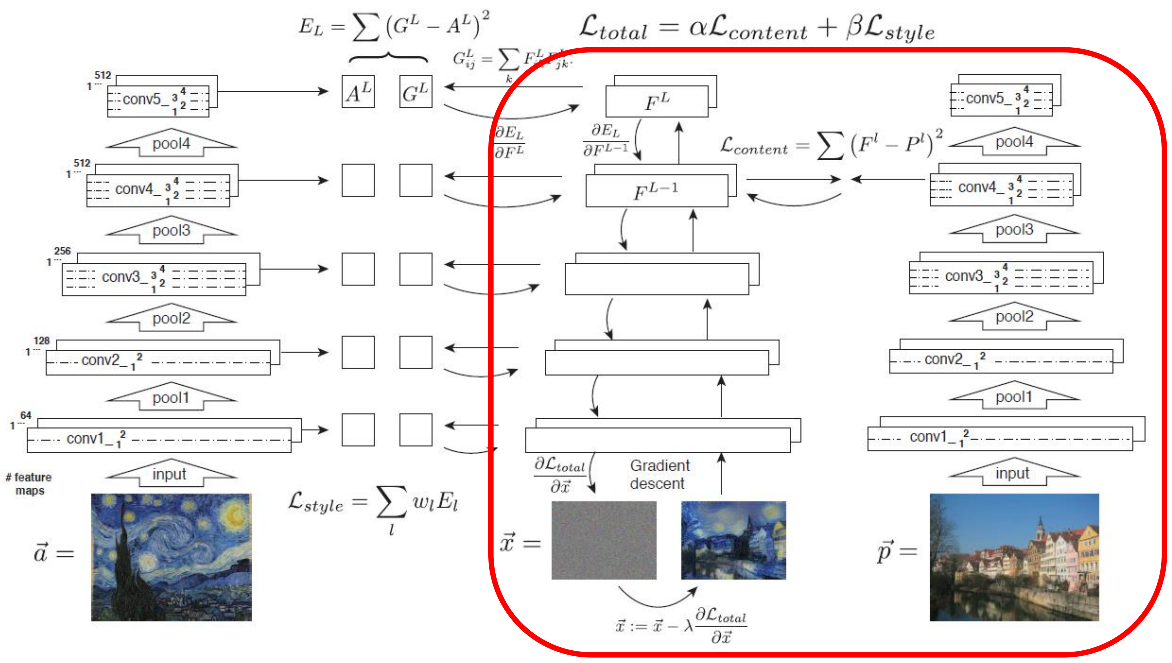
1- Reconstruct a picture by minimizing the content loss only. As well as validating the global code architecture, this step validates:
   a. The VGG19 model construction
   b. The image pre- and post-processing functions
   c. The content loss computation
   d. The gradient descent procedure

2- Reconstruct a picture by minimizing the style loss only. As well as validating the global code architecture, this step validates:
   a. The Gram matrix computation
   b. The style loss computation

3- Style Transfer

Implementation of Image Reconstruction from Content

In this first step, we focus on the middle and right-hand part of the algorithm.


$E_L = \sum (G^L - A^L)^2$
$\mathcal{L}_{total} = \alpha\mathcal{L}_{content} + \beta\mathcal{L}_{style}$
$G^L_{ij} = \sum_k F^L_{ik} F^L_{jk}$
A^L
G^L
F^L
F^{L-1}
$\frac{\partial E_L}{\partial F^L}$
$\frac{\partial E_L}{\partial F^{L-1}}$
$\mathcal{L}_{content} = \sum (F^l - P^l)^2$
conv5_1 2 3 4
pool4
conv4_1 2 3 4
pool3
conv3_1 2 3 4
pool2
conv2_1 2
pool1
conv1_1 2
input
# feature maps
512
256
128
64
$\mathcal{L}_{style} = \sum_l w_l E_l$
$\frac{\partial \mathcal{L}_{total}}{\partial \vec{x}}$
Gradient descent
$\vec{a} =$
$\vec{x} =$
$\vec{p} =$
$\vec{x} := \vec{x} - \lambda \frac{\partial \mathcal{L}_{total}}{\partial \vec{x}}$

We include the necessary libraries. The VGG19 model is retrieved from the Keras Applications libraries. We take the “no top” version as we are not interested in the fully connected layers.

```
from scipy.misc import imsave, imshow
import numpy as np
from keras.applications import vgg19
from keras import backend as K
from keras.preprocessing import image
from utils import *
```

We define the main constants.

```
img_width, img_height = 128, 128
```

We build the VGG model with no fully connected layers on top and we create the dictionary of all layers output tensors

```
model = vgg19.VGG19(weights='imagenet', include_top=False)
outputs_dict = dict([layer.name, layer.output] for layer in model.layers)
```

We load and preprocess the content image P

```
img = image.load_img('elephant.jpg', target_size=(img_width, img_height))
content_img = preprocess_image(img)
```

We reconstruct image from all layers

```
for layer_name in ['block1_conv1', ..., 'block5_conv4']:
```

We get the response of the filter of the desired layer and store into a constant tensor:

```
    get_response = K.function([model.input], [outputs_dict[layer_name]])
    content_feature = K.constant(get_response([content_img])[0])
```

We get the output tensor of the desired layer, and define the loss function as the mean square between this output tensor and the response from the content image. We define the Gradients function, which compute the gradients on the input image with regard to this loss:

```
layer_output = outputs_dict[layer_name][0]
loss = K.sum(K.square((content_feature - layer_output))
grads = K.gradients(loss, model.input)[0]
func_loss = K.function([model.input], [loss])
func_grads = K.function([model.input], [grads])
```

We start from a white noise image with some random noise:

```
gray_img = np.random.random((1, img_width, img_height, 3)) # Channel Last
```

We instantiate the gradient and loss functions, and perform gradient descent on the white noise image:

```
for i in range (iteration):
    loss_value = func_loss([gray_img])[0]
    grads_value = func_grads([gray_img])[0]
    gray_img -= grads_value * step
rec_img = gray_img[0]
bfgs=''
```

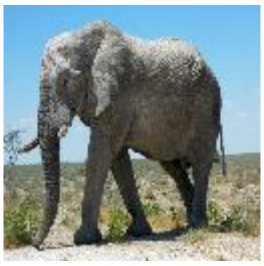
We decode and save the resulting input image:

```
reconstructed_img = deprocess_image(rec_img)
img_name = 'img_reconst_'+bfgs+layer_name+'.png'
imsave(img_name, reconstructed_img)
```

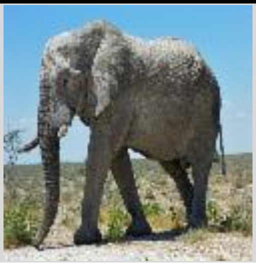
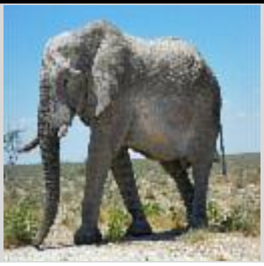
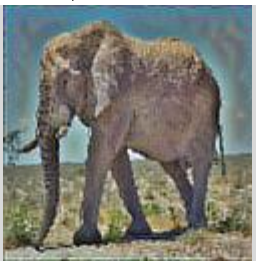
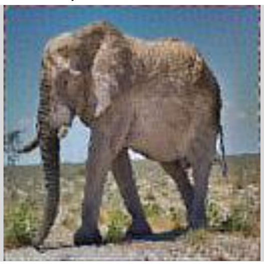
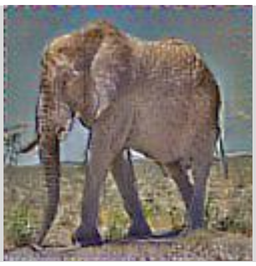
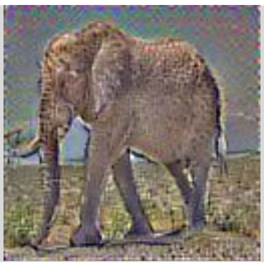
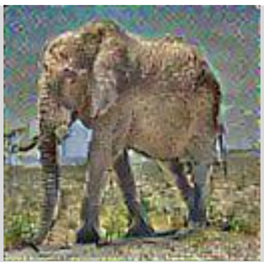
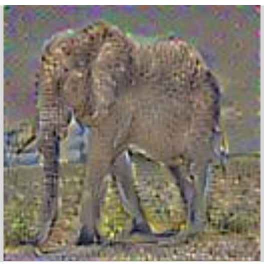
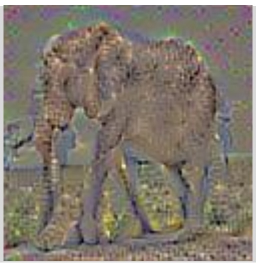
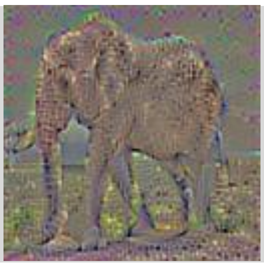
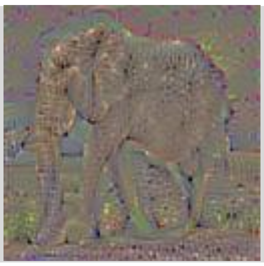
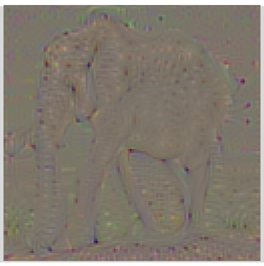
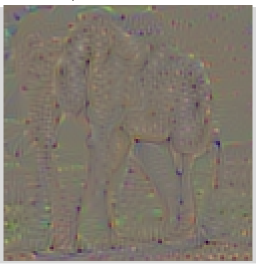
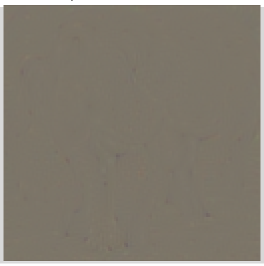
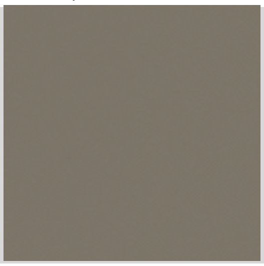
Results with simple gradient descent

The following pictures are reconstructed with the algorithm described above, with simple gradient descent, with `step` and `iteration` values depending upon the layer considered.

We use the following original picture:



Results are given below:

	Conv1	Conv2	Conv3	Conv4
Block1	 20 : 0,01	 20 : 0,001		
Block2	 50 : 0,0001	 50 : 0,0001		
Block3	 50 : 0,0001	 50 : 0,0001	 50 : 0,0001	 50 : 0,00001
Block4	 50 : 0,00001	 50 : 0,00001	 50 : 0,00001	 50 : 0,00001
Block5	 50 : 0,0001	50 : 0,00001	 50 : 0,00001	 50 : 0,00001

Defining the values of the step and of the number of iteration ensuring the convergence of the gradient descent is a pretty difficult task. For instance, we don't manage to get proper convergence for block 5 with this method. In order to obtain better convergence and better results, we have used in the following the BFGS optimization available from SciKit. The authors use this optimization in the final optimization loop of the style transfer algorithm, so we also take the opportunity here to validate its correct implementation.

Implementation with BFGS optimization

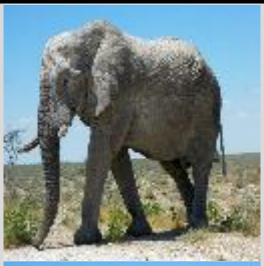
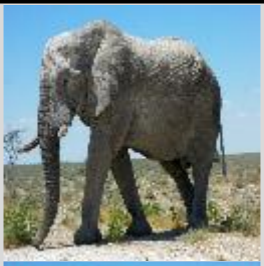
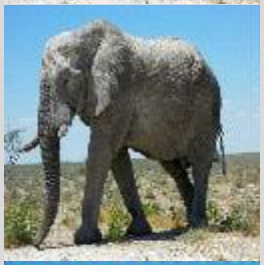
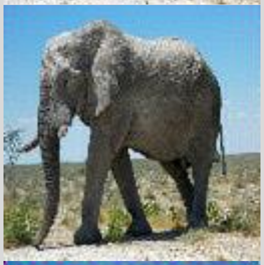
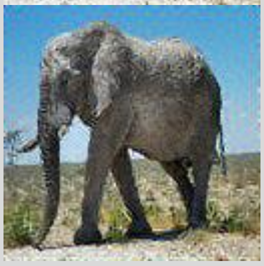
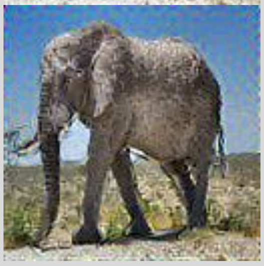
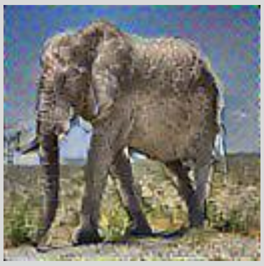
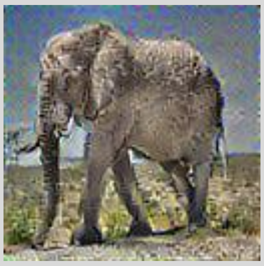
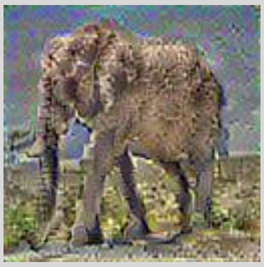
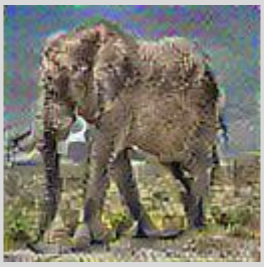
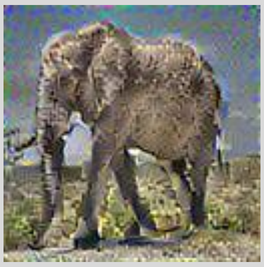
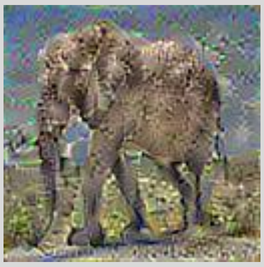
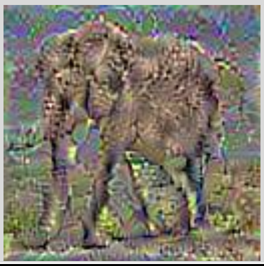
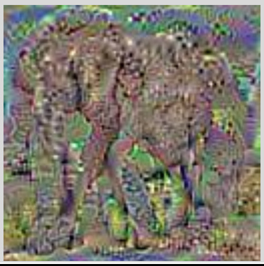
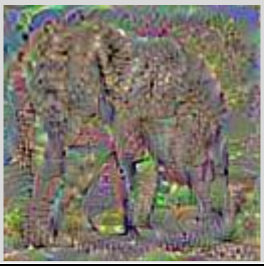
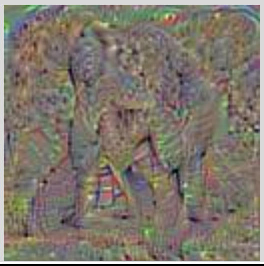
The BFGS optimization (which will be used at a later stage), is implemented with the following code. This code replaces the gradient descent loop.

```
def fn_loss (x):
    x = x.reshape((1, img_width, img_height, 3))
    l = func_loss([x])[0]
    return l.flatten().astype('float64')

def fn_grads (x):
    x = x.reshape((1, img_width, img_height, 3))
    g = func_grads([x])[0]
    return g.flatten().astype('float64')

for i in range (0, 5):
    print ('iteration ',i)
    gray_img, min_val, info = fmin_l_bfgs_b(fn_loss, gray_img, fn_grads)


rec_img = gray_img.reshape((img_width, img_height, 3))
```

Results with BFGS optimization

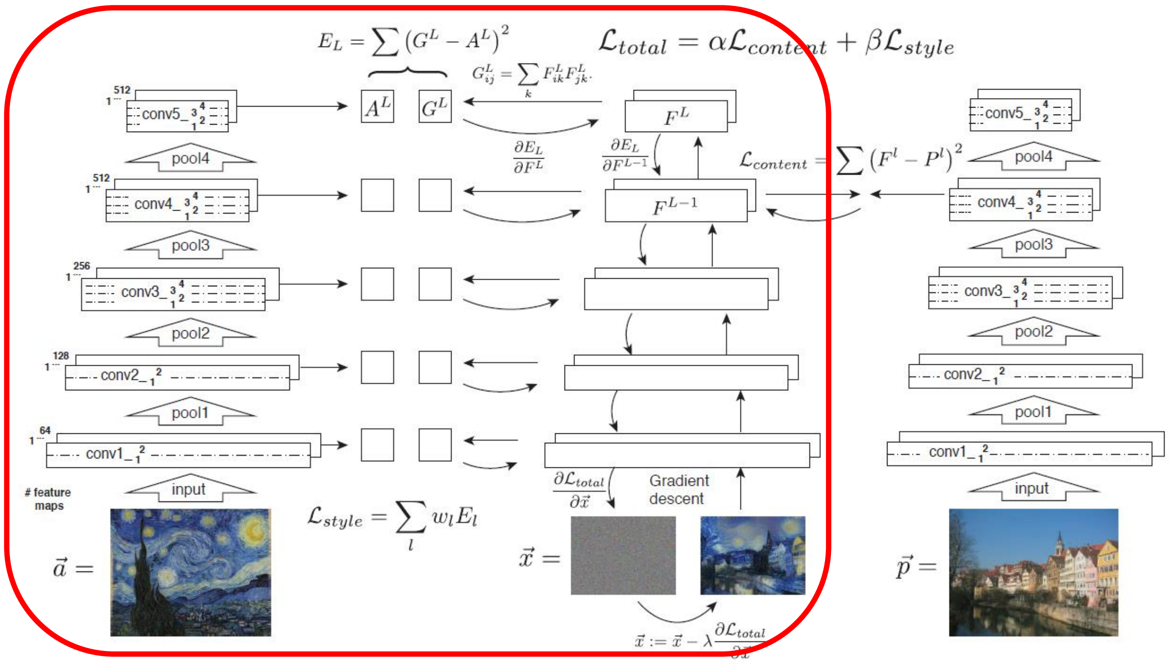
We still use the following original picture:

The results with BFGS optimization are given below.

	Conv1	Conv2	Conv3	Conv4
Block 1				
Block 2				
Block 3				
Block 4				
Block 5				

Implementation of Image Reconstruction from Style

Here we concentrate on the left hand part of the algorithm. We intend to reconstruct styles from the response at different layers of an input style picture. Contrarily to what is described in the diagram below, here we will just concentrate on the loss at one layer (and not summing up losses at all layers), as our intent is to visualize the style representations at different layers, and validate the loss and gradient descent functions.


$E_L = \sum (G^L - A^L)^2$
$\mathcal{L}_{total} = \alpha\mathcal{L}_{content} + \beta\mathcal{L}_{style}$
$G^L_{ij} = \sum_k F^L_{ik}F^L_{jk}$
A^L
G^L
F^L
F^{L-1}
$\frac{\partial E_L}{\partial F^L}$
$\frac{\partial E_L}{\partial F^{L-1}}$
$\mathcal{L}_{content} = \sum (F^l - P^l)^2$
conv5_1
conv4_1
conv3_1
conv2_1
conv1_1
pool4
pool3
pool2
pool1
input
# feature maps
512
256
128
64
$\mathcal{L}_{style} = \sum_l w_l E_l$
$\frac{\partial \mathcal{L}_{total}}{\partial \vec{x}}$
Gradient descent
$\vec{a} =$
$\vec{x} =$
$\vec{p} =$
$\vec{x} := \vec{x} - \lambda \frac{\partial \mathcal{L}_{total}}{\partial \vec{x}}$

The implementation is fairly similar to the content reconstruction implementation and will not be repeated here in its entirety.

The simple gradient descent function doesn't converge here. We have therefore implemented a BFGS optimization as described above.

The loss function also changes to take into account Gram matrices:

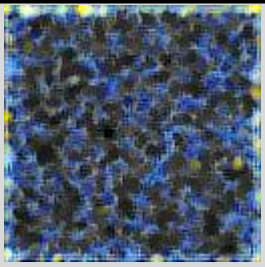
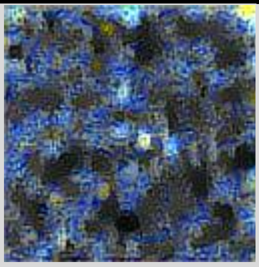
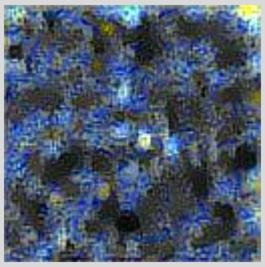
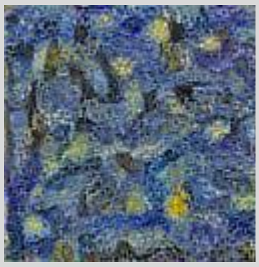
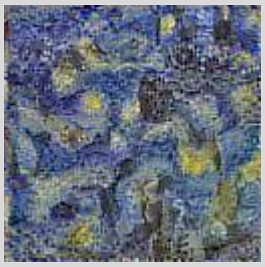
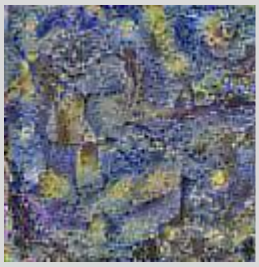
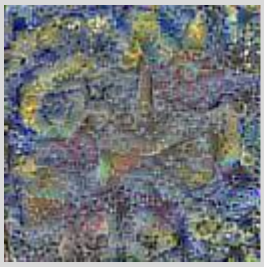
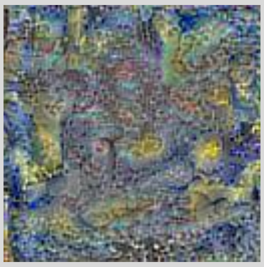
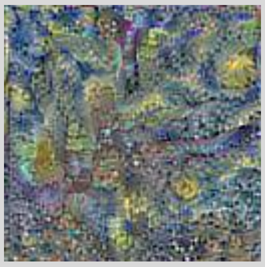
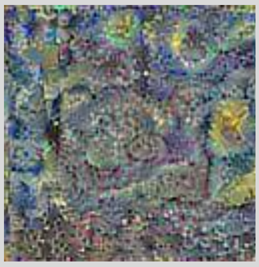
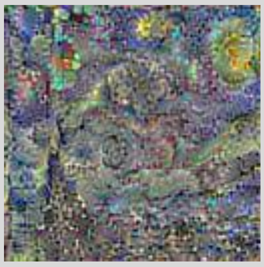
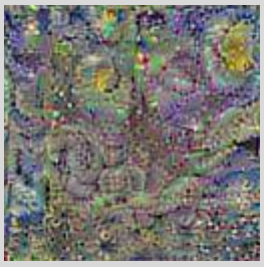
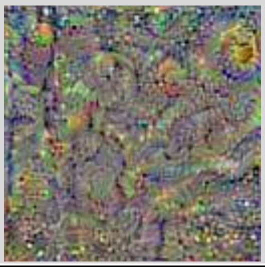
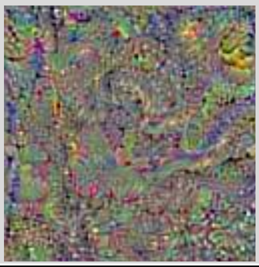
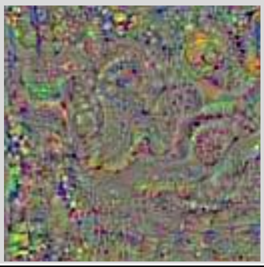
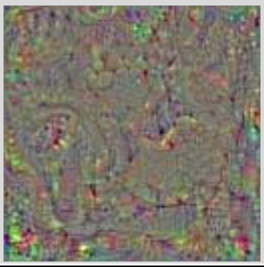
```
def gram_matrix (x):
    assert K.ndim(x)==3
    features = K.batch_flatten(K.permute_dimensions(x, (2,0,1)))
    gram = K.dot(features, K.transpose(features))
    return gram
```

We then write the loss as follows:

```
loss = K.sum(K.square(gram_matrix(style_feature[0]) - gram_matrix(layer_output)))
```

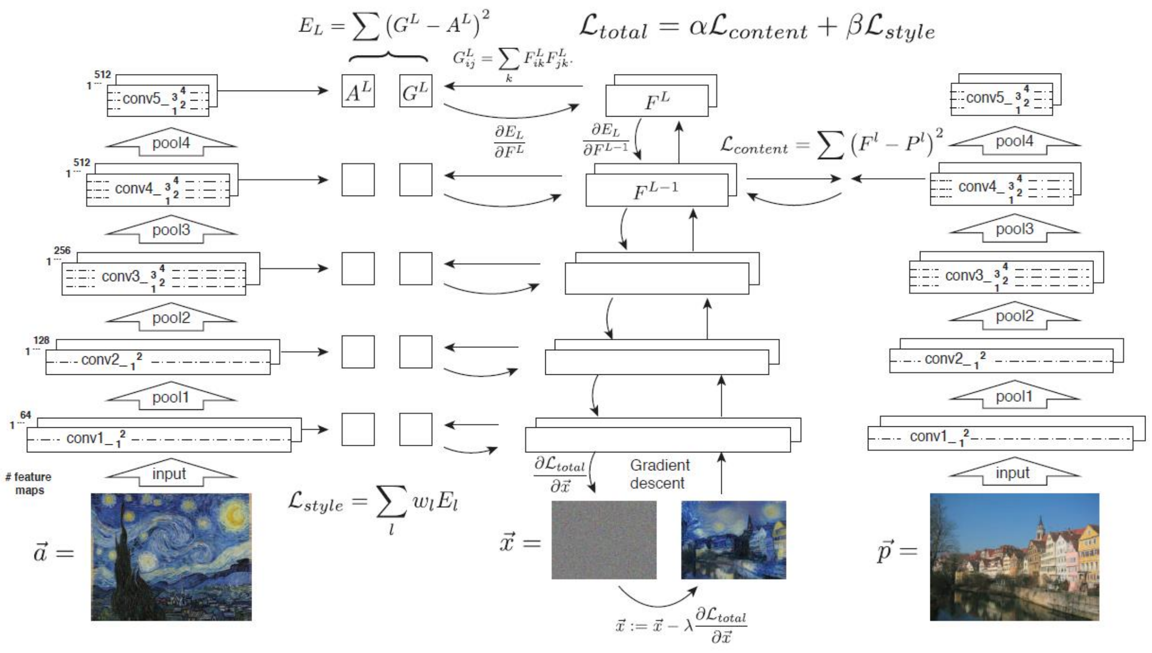
Results with BFGS optimization

Results below are shown at all layers of the network. The deeper the layer, the more semantic information about the style is captured. We use the following original style picture:

	Conv1	Conv2	Conv3	Conv4
Block1				
Block2				
Block3				
Block4				
Block5				

Implementation of Style Transfer

Now we implement the whole algorithm:


$E_L = \sum (G^L - A^L)^2$
$\mathcal{L}_{total} = \alpha \mathcal{L}_{content} + \beta \mathcal{L}_{style}$
$G^L_{ij} = \sum_k F^L_{ik} F^L_{jk}$
$\frac{\partial E_L}{\partial F^L}$
$\frac{\partial E_L}{\partial F^{L-1}}$
$\mathcal{L}_{content} = \sum (F^l - P^l)^2$
conv5_1
conv4_1
conv3_1
conv2_1
conv1_1
pool4
pool3
pool2
pool1
input
# feature maps
$\frac{\partial \mathcal{L}_{total}}{\partial \vec{x}}$
Gradient descent
$\mathcal{L}_{style} = \sum_l w_l E_l$
$\vec{a} =$
$\vec{x} =$
$\vec{p} =$
$\vec{x} := \vec{x} - \lambda \frac{\partial \mathcal{L}_{total}}{\partial \vec{x}}$

The major parts of the algorithm have already been implemented in the sections above, where we have computed separately the content loss and the style loss for a given layer.

The style loss is now computed for all blocks (one layer per block) of the network, as we want the style represented at different depths (we want low-levels features and high-levels features to be represented in the combination picture). Each layer bears a specific weight in the total style loss.

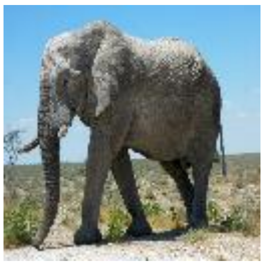
To implement the whole algorithm, we now combine linearly the content loss and style loss into one loss, we compute the gradients of the combination picture we want to synthesize with regard to this total loss, and we perform loss minimization with the BFGS algorithm as above.

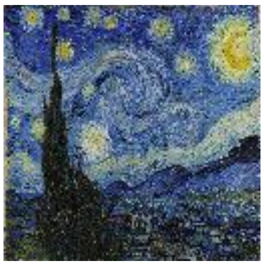
We then have the following parameters to tune to make this algorithm work:

- The style layers we consider and their respective weights in the total style loss
- The content layer we consider
- The alpha and beta parameters: respective contributions of the content loss and style loss
- The number of iterations for the BFGS algorithm (we have set it here to 10 which gives good enough results while ensuring manageable runtimes on a simple laptop with no graphic card).

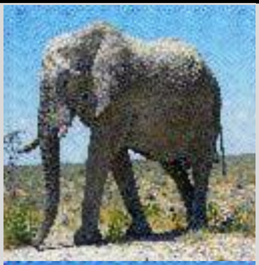
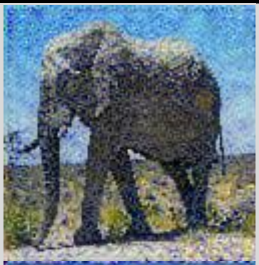
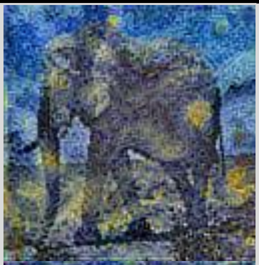
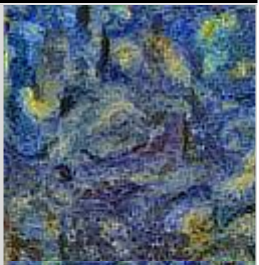
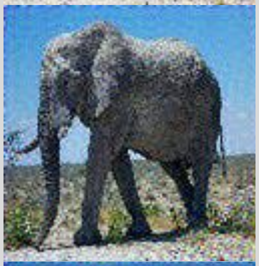
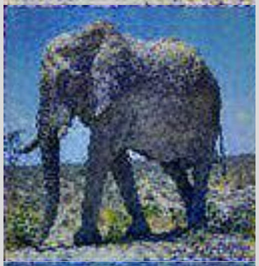
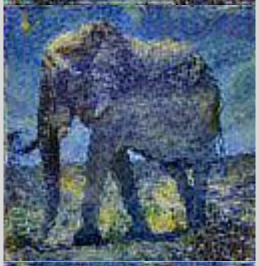
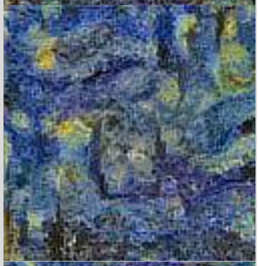
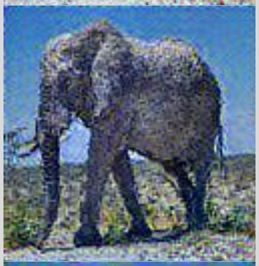
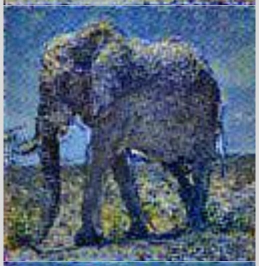
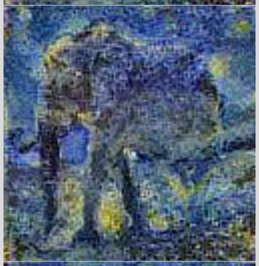
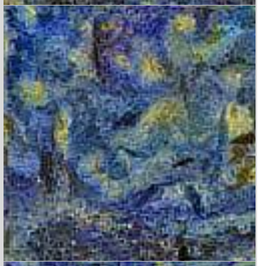
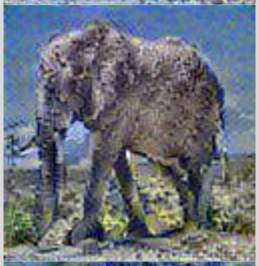
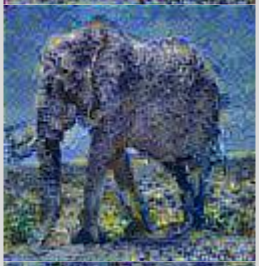
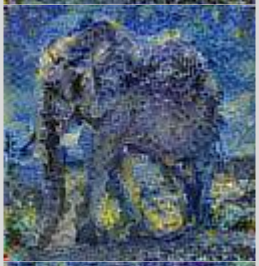
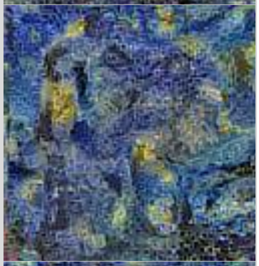
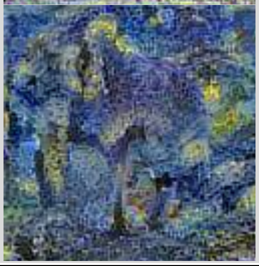
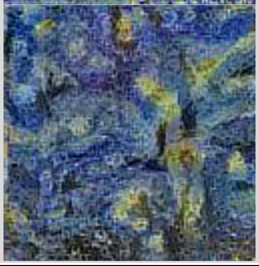
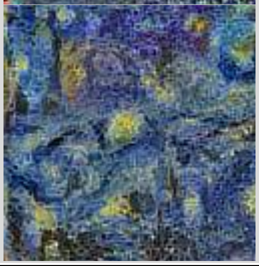
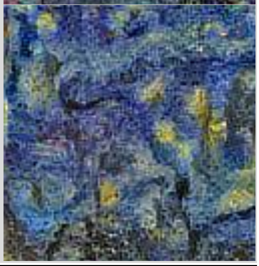
Results

We use the following original style picture:



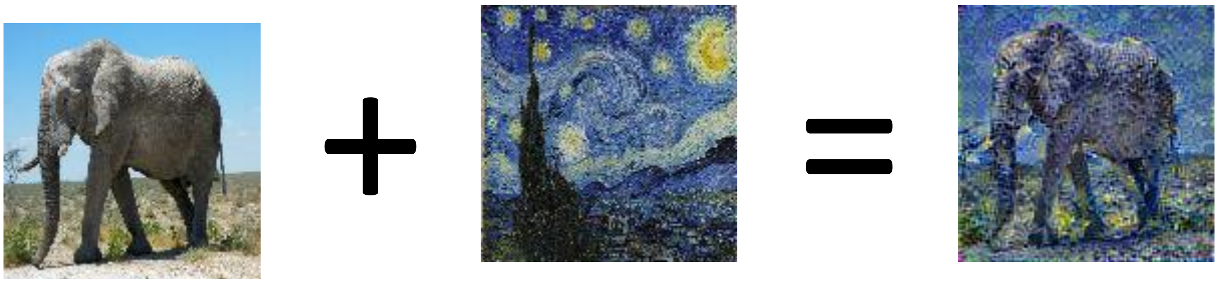


The following results are given for the following style loss parameters: Layers = block1_conv1, block2_conv1, block3_conv1, block4_conv1, block5_conv1. Weights = 0.2, 0.2, 0.4, 0.2, 0.0

	α = 1e-4	α = 1e-5	α = 1e-6	α = 1e-7
Content = Block1 Conv2				
Content = Block2 Conv2				
Content = Block3 Conv2				
Content = Block4 Conv2				
Content = Block5 Conv2				

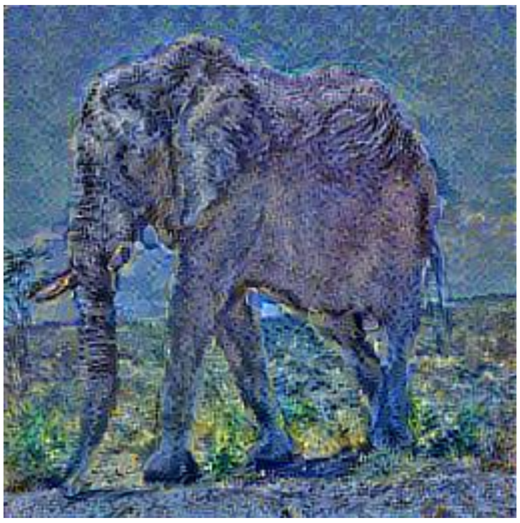
Performing the algorithm on bigger pictures

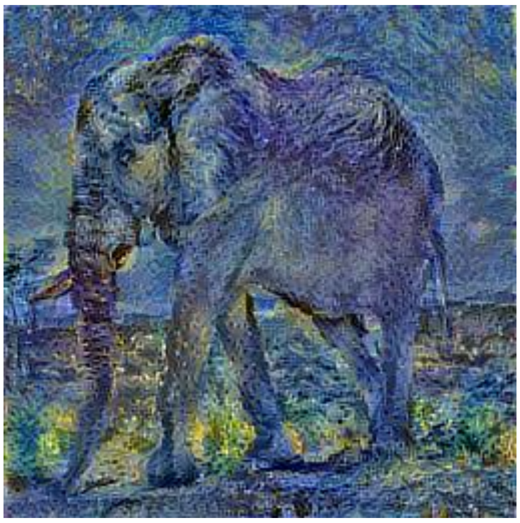
We managed to implement the algorithm and get satisfactory results. Analyzing and implementing the algorithm was a pretty exciting task. The best result is achieved with layer Block4_conv2 and alpha = 5e-6.



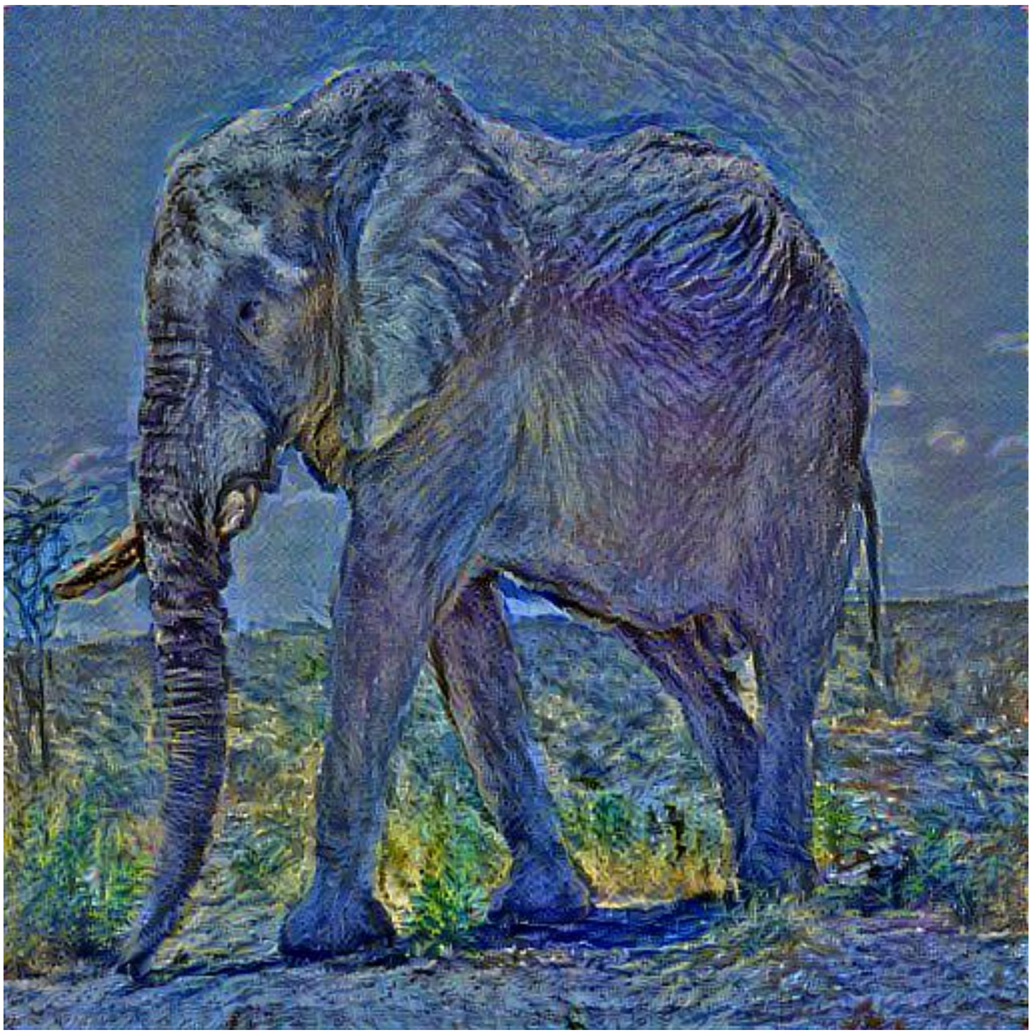
For computing reasons, we have performed all the experiments above picture size limited to 128x128.

More appealing results can be achieved with bigger pictures, but the runtime is significantly longer. It takes 30min for synthetizing a 256x256 image with 10 optimization cycles on an Intel Core i5. In the following, we have tested the algorithm with different values of α, we present here the results for 1e-6 and 1e-7 which are particularly striking. It is interesting to see, that these values of α are not giving good results in smaller resolution.





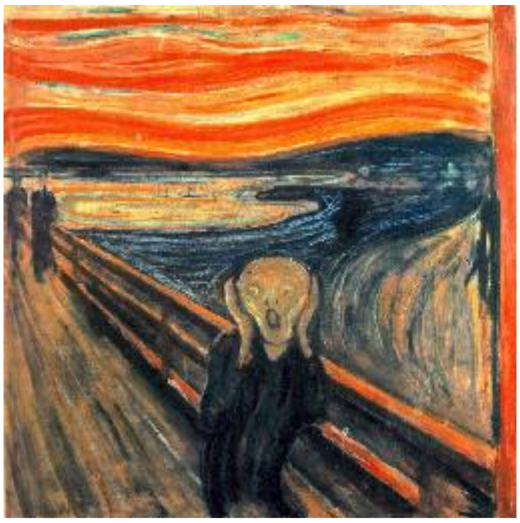
Even better results are obtained with a resolution of 512x512, as shown below, here for alpha=1e-7, and the content layer still Block 4 Conv 2. Synthesis runtime is 1h40mn for 10 optimization cycles.

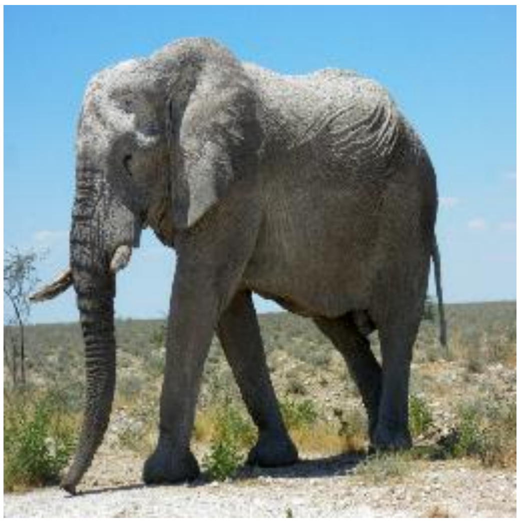


Performing the algorithm with other styles

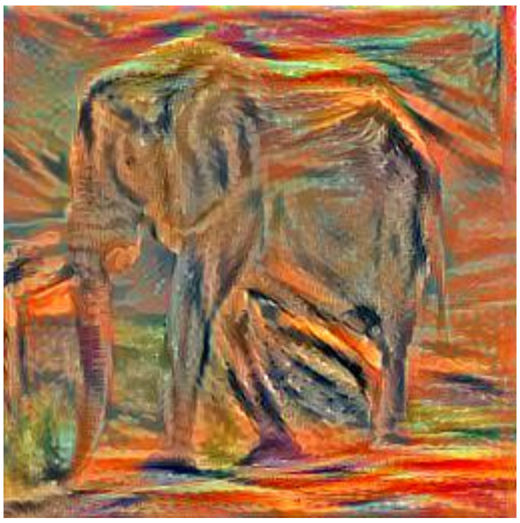
We've tested the algorithm with the style extracted from Munch's Cry picture. The result is given below for α=1e-7 and α=1e-6. The content picture loss is still extracted at Block 4 conv2.

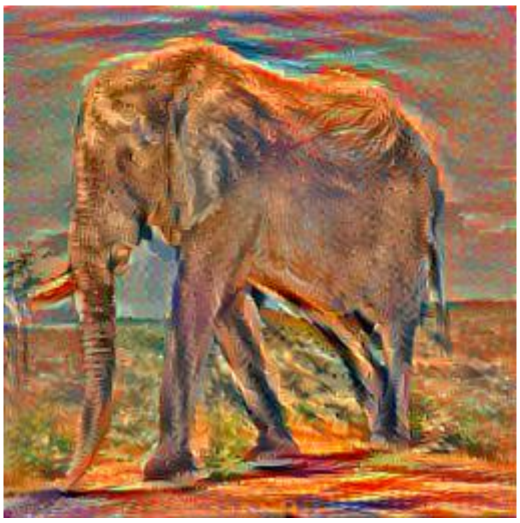
Original pictures:





Synthesized pictures for α=1e-7 and α=1e-6:





Discussion – Sources

Fine tuning of the algorithm parameters consumes a lot of CPU resources. I believe better results could be achieved by exploring which layers to select, and how to weigh them correctly. However, this requires covering possibilities in a space of dimension layers x possible weights.

The main challenges encountered were related to handling of the K backend – the Keras documentation is pretty light. Hopefully there are good examples on GitHub:

https://github.com/fchollet/keras/tree/master/examples