Combining the Deep Dream and style transfer algorithm to create art

May 16, 2018
Jente Insing, Bonny Vloet

Introduction

For this assignment we had the freedom to choose what kinds of neural network architectures to experiment with. We chose the deep dream model as well as the style transfer model. The deep dream algorithm generates dream-like variations on input images, while the style transfer algorithm generates images with the content of one input image and the style of another input image. In this report we explain how both of these models are operating. Furthermore, we reflect on the results of our experimentation with the networks. Finally, we merge the algorithms, thereby combining the deep dream and the style transfer functionality to create animated artworks.

1 The Deep Dream Algorithm

The deep dream algorithm is essentially amplifying patterns that are present in the input images, thereby outputting hallucinatory images of which an example is shown in figure 1 below. What kind of patterns emerge depends on which classes were present in the dataset that the network is trained on, as well as which of the network's layers is chosen for amplification. In a trained image classification network, each layer progressively extracts higher and higher-level features, until the output layer finally decides on what the image contains. In the example below, the algorithm particularly tends to produce more complex patterns like animals and vehicles, reflecting an amplification of higher layers. When medial layers are chosen for amplification, more semi-complex patterns like eyes and wheels emerge. For amplification of lower layers the emerging patterns tend to be rather geometrical.

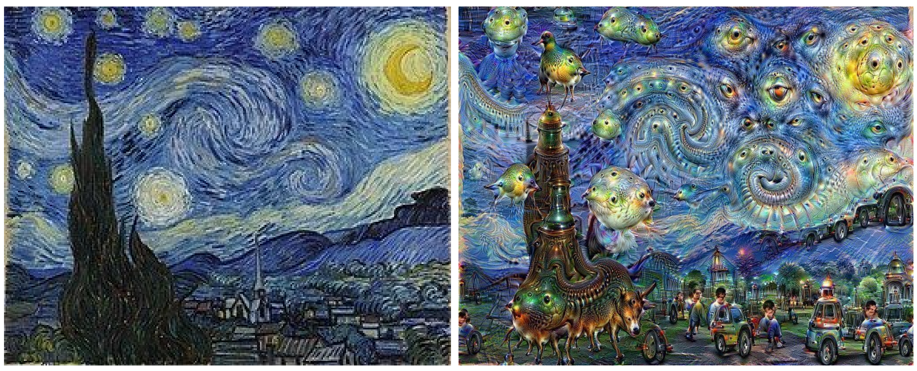


***Figure 1:** Vincent van Gogh's painting The Starry Night (left) and Deep Dream's output of it (right).*

The deep dream software uses the inception model (further explained below), which was originally designed to detect patterns such as creatures and objects in images. However, the inception model is especially famous for its ability to produce dream-like, hallucinogenic variations on input images. To perform this, a trained inception model is ran in reverse with the task to slightly adjust the input image in order to maximize the confidence scores of activated output neurons (representing an image class that it is trained on). In this way, an area in the image that shows some resemblance to for instance a bird, will get changed in such a way as to amplify the resemblance.

1.1 The Inception Model

As mentioned, the deep dream algorithm uses a pretrained model (trained on the ImageNet dataset), codenamed inception. This model introduced a clever method to increase both the depth and the width of the network, while keeping the computational cost constant (Szegedy et al., 2015). In figure 2 below the architecture of the inception model is shown. The input (a matrix of 300x300x3, representing a 300 by 300 pixels input image in RGB color) is passed through three vanilla convolutional layers to extract features, after which max pooling is performed to reduce variance and computational complexity by losing the information of the lowest activations within certain blocks. Subsequently, the data is passed through another two vanilla convolutional layers and undergoes another max pooling operation. After this, the data travels through successive stacks of inception modules using 1x1, 3x3 and 5x5 filters. We will go deeper into these inception modules later. In between, there is one intermediate softmax output layer (aux_logits). This extra output layer is added as a solution for the vanishing gradient problem that can occur during training of deep network architectures. The final loss then consists of a combination between the intermediate loss and the final loss, making it unlikely for the vanishing gradient problem to occur during backpropagation of the error. The final output layer uses average pool (thereby retaining some information on low activations), dropout (a regularization technique to reduce overfitting) and softmax (assigning probabilities to every class that the model is trained on).

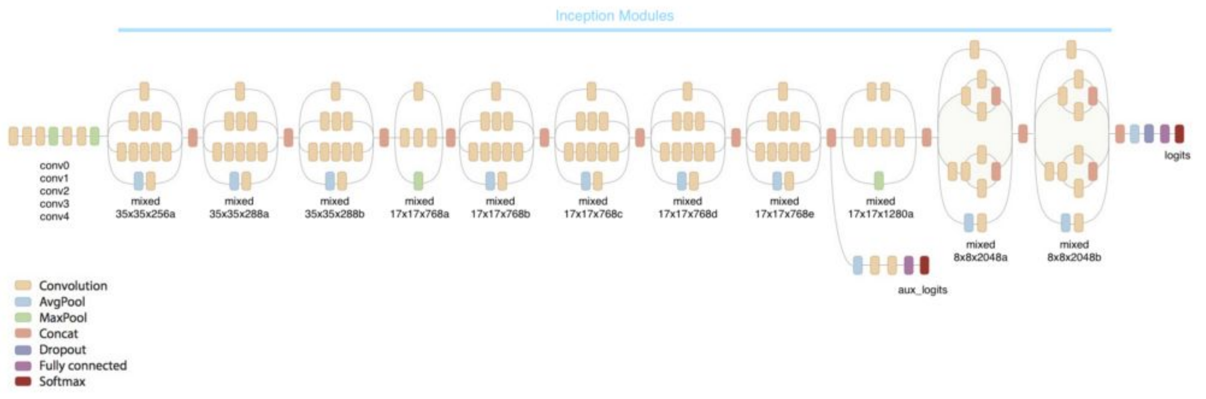


Figure 2: *The architecture of the inception model which is used in the deep dream algorithm.*

The inception modules, of which the architecture is shown in figure 3 below, are designed to recognize features at multiple scales (1x1, 3x3 and 5x5). Since these convolutional processes are rather expensive, especially given the deep architecture, 1x1 convolutions are preceding the 3x3 and 5x5 convolutions, thereby reducing the dimensionality of the data. Since in vision networks the output of nearby activations can be expected to be highly correlated, reducing the activations before aggregation should result in similarly expressive local activations (Szegedy et al., 2016). This clever approach greatly reduces the amount of parameters (to about 5 million) and thereby the computational cost.

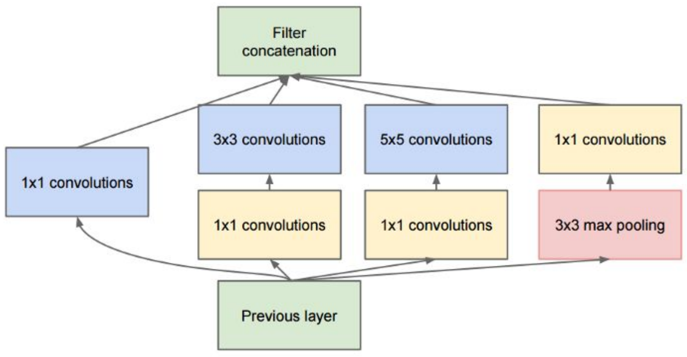


Figure 3: *The cost-reducing architecture of the inception modules.*

As a method to visualize what image representations the network has learned, the network can be ran in reverse, while it is asked to enhance a random noise input image in such a way that a particular pattern gets elicited (Mahendran & Vedaldi, 2015). This is essentially how the deep dream algorithm is operating: A reversed inception network is provided with an input image and its target is set to enhance the input image in such a way that nodes that are activated get elicited more and more, thereby gradually creating new instances of a variety of classes within the original input image.

1.2 Hvass Laboratory's Deep Dream Algorithm

For our experimentation with the deep dream algorithm we used a comprehensible version developed by Hvass Laboratory. The algorithm, which uses the inception5h model, is available in Hvass Laboratories' TensorFlow Tutorials GitHub repository. (To make use of the deep dream algorithm, copy the entire GitHub repository, since other parts of the repository contain some dependencies for the inception algorithm to work properly.) We consider the Hvass Laboratories deep dream algorithm to be an improvement on the original TenserFlow's deep dream algorithm, not only because the code is written and commented in a more comprehensible way, but also because the algorithm was designed to produce more large-scale patterns. Since the inception model is trained on rather small images (300x300), the deep dream algorithm would create a lot of small patterns in large input images. Hvass Laboratories version of the algorithm calls the deep dream algorithm recursively, a straightforward technique to ensure the enhancement of more global patterns, thereby preserving the original content of the image to a greater extent. As shown in figure 4, the algorithm repeatedly downscales the image before the image is sent through the deep dream algorithm. The result is then upscaled and blended with a same-sized downscaled version of the input and passed through another deep dream operation. This procedure is then repeated to produce the final result.

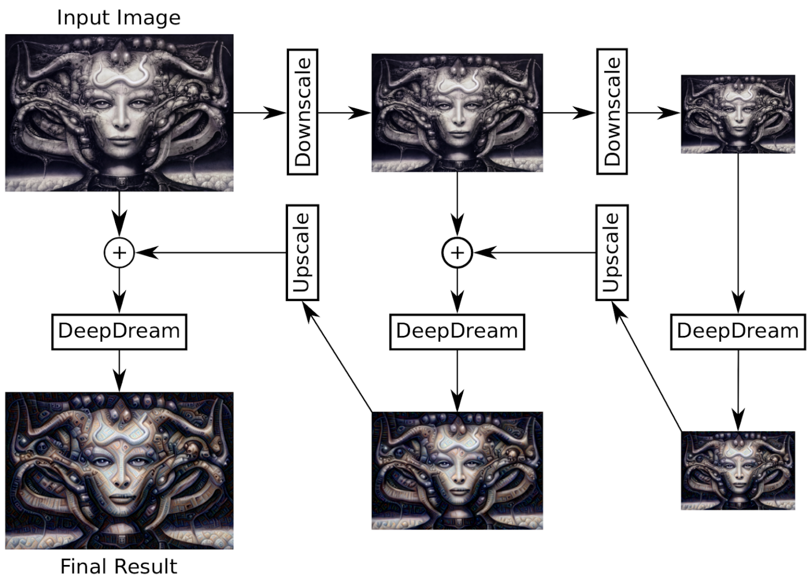


Figure 4: *An overview of the Hvass Laboratory's deep dream algorithm. As shown by the final result, this method enhances global patterns rather than local ones.*

1.3 Deep Dream Experiments

To obtain a better understanding about the pattern emergence, we performed a couple of experiments with the Hvass Laboratory's deep dream algorithm. We inputted all-black, all-white and random noise images to see what patterns would emerge. Furthermore, we experimented with adjusting the recursion settings to produce more refined patterns and changing the feature channels to amplify specific features of the inception model's layers.

1.3.1 Feeding an All-White and All-Black Image

As a first experiment we fed the deep dream algorithm with an all-white and an all-black image of equal size, of which the output images are shown in figure 5 below.

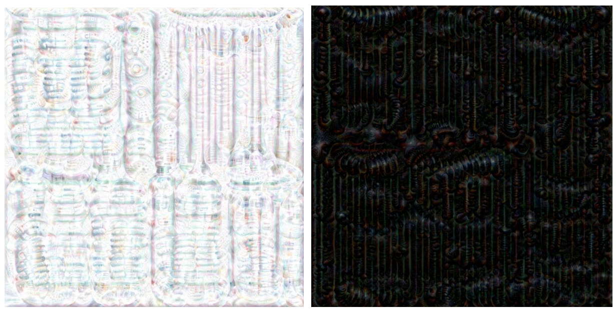


Figure 5: *Deep dream inception output for an all-white image (left) and an all-black image (right) maximizing the fourth tensor.*

The emerging patterns are caused by the `get_tile_size` and the `tiled_gradient` helper functions of the `optimize_image` function. Because the inception model could handle input images from every size, a problem that could occur is that the computer could run out of RAM. To keep the RAM usage low the `get_tile_size` function divides the image in multiple tiles depending on the image resolution and calculates the gradient of the tiles by using the `tiled_gradient` function to eventually calculate the gradient of the whole image and normalizing the gradient of the different tiles. The tiles get a small random offset to make the borders of the tiles less visible in the output image. Because of this an all-white or all-black image gets gradients of the borders of the tiles, this effect on the image will be amplified by the deep dream algorithm, resulting in output in figure 6.

The `optimize_image` function calculates the value of the gradient of a certain layer in the model and manipulates the gradient, like blurring the gradient and color-channels to smoothen the image. The deep dream algorithm will classify features recognized by the specific layer and maximizes the score of it. To classify the features in the image, we have used the Inception model. The first layers in the network has learned to classify low level features like lines and edges which emerges to recognizing complex patterns the network was trained on, in this case recognizing dogs. By using backpropagation we update the input image with the maximized patterns the network classified as a dog which will result in an image looking like a hallucination or dream.

1.3.2 Feeding an Image with Random Noise

Another experiment we conducted was feeding the network an image of randomly valued RGB pixels to find out if the algorithm would be able to detect patterns in randomness. For this experiment we chose to amplify the seventh layer of the network, which is predominantly recognizing dogs and fur. The result is shown in figure 6.

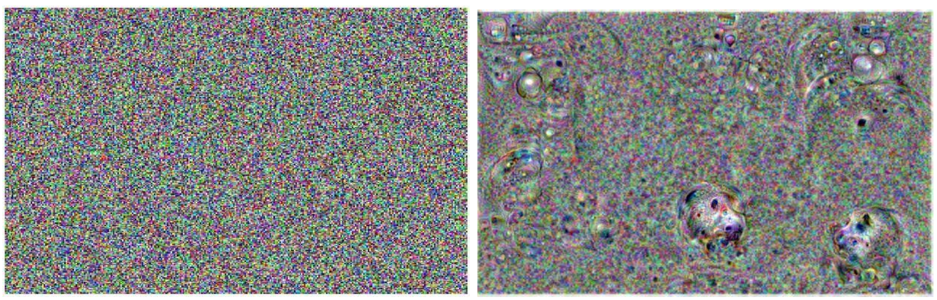


Figure 6: *The randomly valued RGB pixel input image (left) and the deep dream output (right).*

We see some dog-like faces appearing in the output image just below the center and in the bottom-right corner. There are a couple of reasons why the network recognizes complex patterns in random noise. First, although random noise is random, there are still patterns recognizable by the network which are amplified by the deep dream algorithm. Secondly, as mentioned earlier, the network divides the images into small tiles to reduce the computational load while calculating the gradient. The borders of these tiles create a pattern in the image.

1.3.2 Adjusting the Recursion Settings

To attempt to make the network produce more refined patterns, we experimented with adjusting the recursion settings. The inception model is trained on an imageset with small images (300x300). Therefore we have to downscale the size of an higher resolution input image with the `recursive_optimize` function. This function downscales the input image and blends the output of the network with the input image and upscales this input image back to its original size in multiple steps in order to refine the amplified features more appropriately with the original size of the image. This results in a better looking output image. See figure 7 for a comparison. The results in figure 7 are generated with the default parameter settings: `rescale_factor=0.7`, `num_repeats=4` and `blend_mode=0.2`.

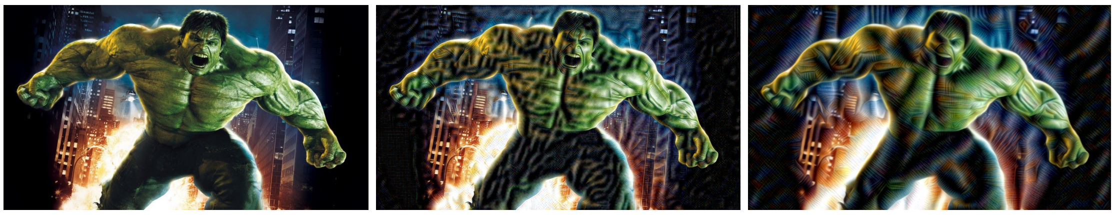


Figure 7: *Original image (left), deep dream output without recursion (center) and with recursion (right) both maximizing the third tensor of the inception model.*

1.3.3 Adjusting the Feature Channels

In order to amplify patterns with a lower level complexity, we experimented with adjusting the feature channels that the algorithm calls upon. In the previous experiments we used full layers of the inception model, but instead of using a full layer with all features it is possible to select a certain feature channel within a layer. A feature channel within a certain layer represent smaller patterns within a complex pattern. A problem with the feature channels is that it is difficult to find out which features are in which channel. The only way to find out which features are represented by which channels is by means of trial and error. In figure 8 we have maximized the full 7th layer of the inception model and the same image maximized with only the first 3 feature channels of the 7th layer. When we look at the image on the left we clearly see complex patterns like dogs appearing. In the right image the algorithm focuses clearly more on round patterns.

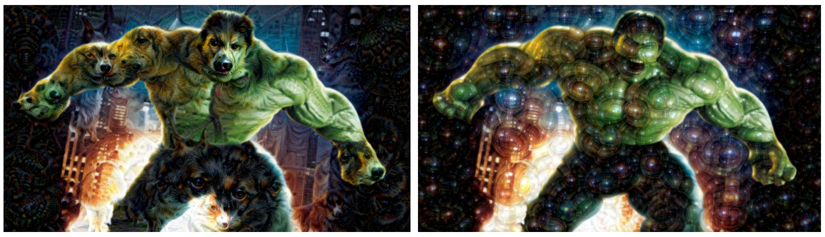


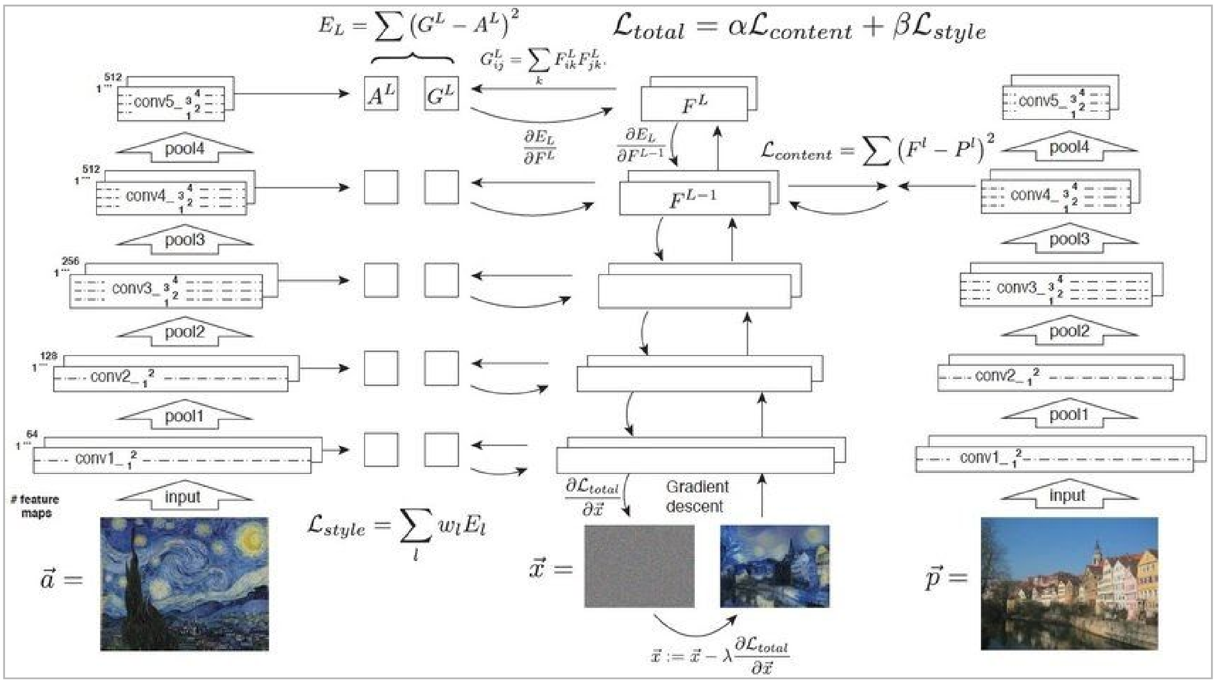
Figure 8: *Output with the full 7th layer (left) and with only the first 3 feature channels of the 7th layer (right).*

2 The Style Transfer Algorithm

Another, perhaps even more interesting method for image manipulation is the deep style transfer algorithm. This algorithm is fed two input images and outputs a merged image that contains the semantic content of the first and the style of the second input image. To achieve this, the algorithm first needs to extract a representation of the content and the style of the input images. Obtaining a representation of the content is simply achieved by feeding the image through a convolutional neural network (CNN) that is trained on object recognition, in this case the VGG-16 model with 13 convolutional layers. The feature responses in the higher layers of the network form a representation of the content (Gatys et al., 2015). Obtaining a representation of the style is achieved by using a feature space that is originally designed to capture information about the texture of images (Gatys et al., 2015). This future space consists of the correlations between the various filter responses over the spatial extent of the feature maps and is built on top of the filter responses of each layer. By including the feature correlations of each of the layers the algorithm constructs a texturised version of the input image on multiple scales, in which the local spatial information is preserved but the global spatial arrangement is lost. It is of crucial importance that the VGG's learned filters, inherent in its weights, are preserved for texture generation and thus style representation to succeed.

During the synthesis of the mixed image, the loss function to be minimized contains two terms, one for the content and one for the style. Therefore, the emphasis on either image content or style can be regulated smoothly. For the purpose of mixing the content of image A with the style of image B, we minimize the loss function for content for image A and the loss function for style for image B. However, this duplex construction of the loss function also allows to construct different trade-offs between content and style preservation for both of the images, thereby creating alternative aesthetics (Gatys et al., 2015).

In figure 9 a visualization of the style transfer algorithm is shown. On the left side the style features (A^l) of the style image are computed, whereas on the right side the content features (P^l) of the content image are calculated. To generate the mixed image, a random white noise image is passed through the network, for which the style features (G^l) and content features (F^l) are computed. Subsequently, the mean squared error between the style features of the left input image (A^l) and those of the mixed image (G^l) is computed, as well as the mean squared error between the content features of the right input image (P^l) and those of the mixed image (G^l). Through the iterations, the mixed image is adjusted is such a way as to minimize the sum of these mean squared errors with the gradient descent method, finally resulting in a mixed image that contains the style of the left and the content of the right input image.



$E_L = \sum (G^L - A^L)^2$
$\mathcal{L}_{total} = \alpha\mathcal{L}_{content} + \beta\mathcal{L}_{style}$
$G^L_{ij} = \sum_k F^L_{ik}F^L_{jk}$
A^L
G^L
F^L
$\frac{\partial E_L}{\partial F^L}$
$\frac{\partial E_L}{\partial F^{L-1}}$
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
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
$\vec{a} =$
$\vec{x} =$
$\vec{p} =$
$\vec{x} := \vec{x} - \lambda \frac{\partial \mathcal{L}_{total}}{\partial \vec{x}}$

Figure 9: *The architecture of the style transfer algorithm (Gatys et al., 2016). The left side extracts the style features of the first input image, whereas the right side extracts the content features of the second input image. The mixed image, initially a random white noise image, is adjusted in such a way as to minimize the total error.*

2.1 Hvass Laboratory's Style Transfer Algorithm

For our experimentation, we used the style transfer algorithm of Hvass Laboratory, which is available in the Hvass Laboratory's TensorFlow Tutorials GitHub repository. (To make use of the style transfer algorithm, copy the entire GitHub repository, since other parts of the repository contain some dependencies for the algorithm to work properly.) The algorithm uses the pre-trained VGG-16 model, containing 13 convolutional layers. A representation of the style features is retrieved from all of the 13 convolutional layers, ensuring texture preservation on multiple scales. A representation of the content features is retrieved from the fifth layer only, thereby reducing computational load while giving a satisfiable result. Both of the loss functions for content and style are normalized so that one can easily tweak the ratio between the two, thereby shifting the emphasis toward either content or style preservation.

To begin with, we fed the network both meaningful content images (`content_image`) and applicable style images (`style_image`), of which some results are shown in figure 10 below. It can be seen clearly that the generated mixed images (on the right) are representing the content of the images in the top-right corner and the style of the images in the bottom-right.

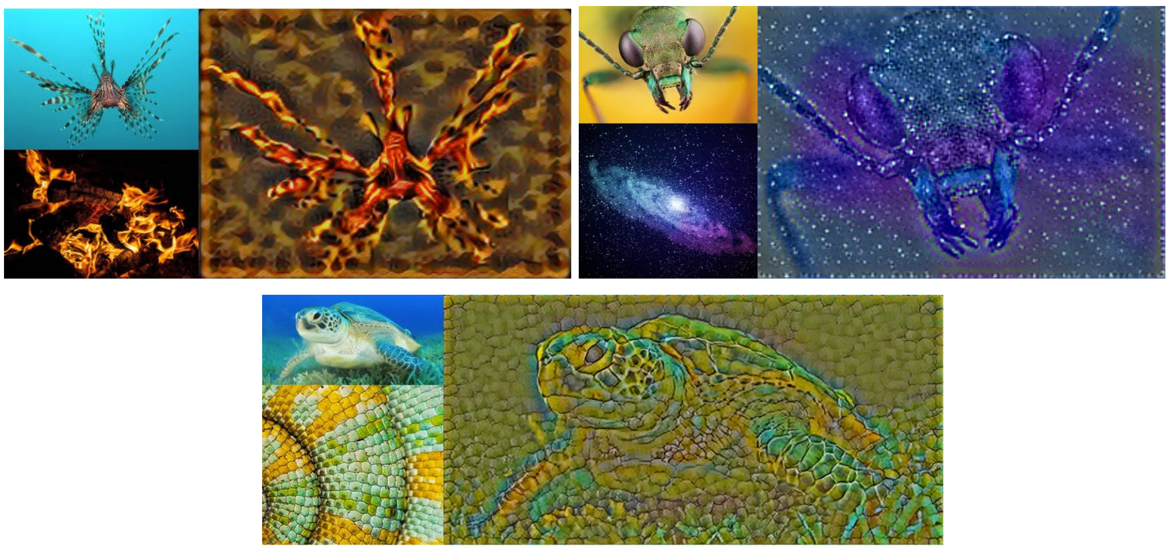


Figure 10: *Some results generated with the style transfer algorithm. The mixed images (the right part of each of the images) are clearly representing the content (top-left) and the style (bottom-left) of the input images.*

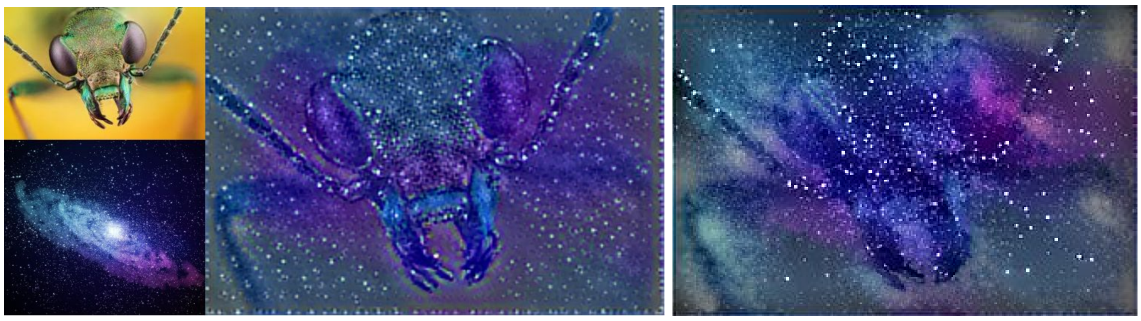
The results above were generated with the default parameter settings: the gradient descent optimization method, 120 iterations, a step size of 10.0 for the gradient in each of the iterations, a maximum height or width of 300 for the style image (to reduce the computational load of the algorithm), a denoising factor of 0.3 (for noise reduction in the generated mixed image) and a ratio of style to content loss of 10.0 to 1.5 (emphasising style representation to a greater extent than content representation). For the next section, we experimented with some of these default parameter settings.

2.2 Style Transfer Experiments

We attempted to improve the generated images by tweaking the algorithm's default settings. We tried increasing the number of iterations while decreasing the step size taking in each iteration, tweaking the denoising factor, changing the maximum allowed size for the style image and tweaking the trade-off between content and style preservation. The results of these experiments are showed and interpreted below.

2.2.1 Running More Iterations of the Algorithm

The most obvious way to improve the results is to increase the number of iterations while the step size taken in each iteration is decreased. We changed the default settings from `num_iterations=120` and `step_size=10.0` to `num_iterations=3600` and `step_size=1.0`. The results are shown in figure 11 below.



num_iterations=120, step_size=10.0
num_iterations=3600, step_size=1.0

Figure 11: *The input content (top left) and input style (bottom left) images and their style transfer results with the default settings (center) and with an increased number of iterations and decreased step size (right).*

Looking at figure 11 we conclude that increasing the number of iterations while decreasing the step size for each iteration results in a stronger representation of the style features since the color and texture show more similarity to the input style image. However, the updated settings simultaneously result in some loss in the content features representation, since the overall contours get less obvious. Which of the settings is to be preferred is a matter of taste rather than a substantive choice. In our opinion, increasing the number of iterations while decreasing the step sizes results in more interesting images that spike the imagination to a greater extent than the images generated with the original settings. Though, it must be considered whether it is worth the time, since it takes several hours (on a 2013 MacBook Pro) to perform that many iterations.

2.2.2 Tweaking the Denoising Factor of the Output Image

Another obvious way for trying to improve the generated mixed image is tweaking the denoising factor that is used to update the mixed image. In the implemented denoising algorithm, the image is shifted one pixel along the x- and y-axis, after which the absolute difference with the original image is calculated and the sum of these differences over all the pixels is taken. The resulting value, `loss_denoise`, is to be minimized to suppresses some of the noise in the generated mixed image. The default setting for the factor with which the denoise loss is updated is 0.3. We experimented with both a lower denoising factor (0.1) and a higher denoising factor (0.6), of which the results are shown in figure 12 below.

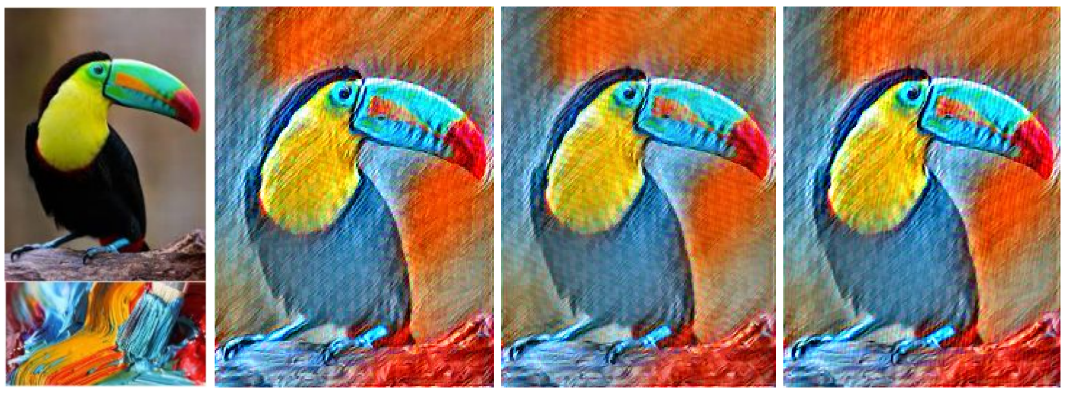


Figure 12: *The input images for content (top left) and style (bottom left) and the outputted images with different settings for the denoising factor (annotated).*

As can be seen in figure 12 above, the differences in the output images are very minimal. When examining the results more closely we see that the denoising factor of 0.3 results in somewhat more equalization for adjacent pixels compared to denoising factor 0.1, demonstrated by the somewhat fading pattern on the bird's belly. Updating the denoising factor to 0.6 somewhat increases the contrast compared to denoising factor 0.3. However, this difference is so small that we do not consider them to be an improvement on the default settings. Yet, for different input images changing the denoising factor could have a bigger impact.

2.2.3 Changing the Maximum Size of the Style Image

To reduce the computational load of the algorithm, the default settings allow the style image to have a maximum width or height of 300 pixels. When either the height or width is exceeding 300 pixels, the image is automatically resized so that the maximum size equals 300 pixels while the proportions between the width and height are preserved. For testing different restrictions on the maximum size of the style input image, we used a style image of 570 x 380 pixels and run the algorithm on the default maximum setting of 300, a decreased maximum setting of 150 (half the default value) and a non-restricted maximum setting (so the image retains its maximum size of 570, which is approximately twice the default value). The results are shown in figure 13 below.

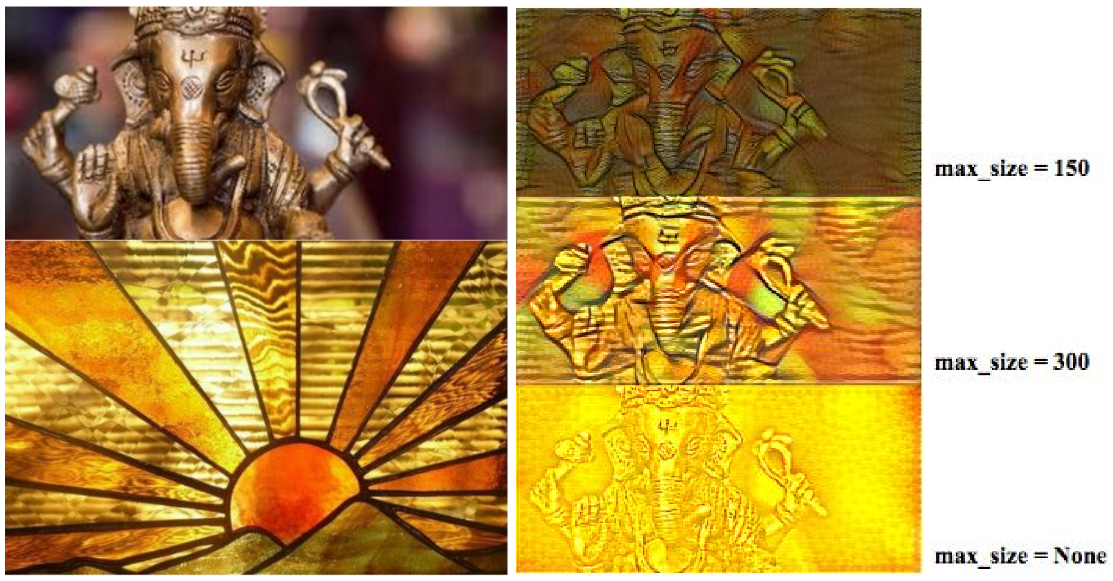


Figure 13: *The input content (top left) and input style (bottom left) images and the resulting generated mixed image (right) for different maximum sizes of the style input image (annotated).*

Evidently, the different restrictions on the maximum size of the style input image are producing rather distinct aesthetics. However, the style of the original style image (figure 13 bottom-left) is best captured by the default setting of `max_size=300` (figure 13, center-right), since it matches the colors, brightness and texture of the style image most. For the setting `max_size=None`, we see that the output image is predominantly colored in yellow. We thought this was rather strange, since that particular color has a minimal presence in the style input image.

2.2.4 Tweaking the Trade-off Between Content and Style Preservation

We were curious to find out to what extent the generated mixed image would be affected if we were to change the trade-off between the style and content representation. To make the difference in emphasis on style and content the generated mixed images evident, we used the same input images for all of the style content ratios. Therefore, we changed the `weight_content` variable to 10 to match the value of the `weight_style` variable, establishing a content style ratio of 1 : 1. Subsequently, we inverted the original settings, thereby creating a content style ratio of 10 tot 1.5. The results are shown in figure 14 below.

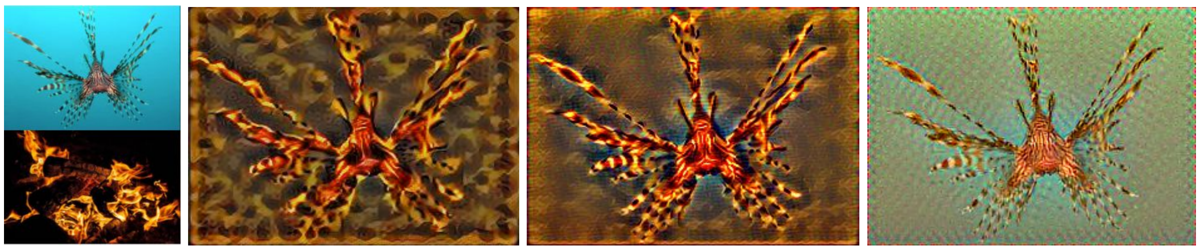


Figure 14: *The content (top left) and the style (bottom left) input images and the generated results for different content style ratios (annotated).*

Looking at the resulting images in figure 14 from left to right, as expected, we see an increasing emphasis on the content representation (the fish) and a decreasing emphasis on the style representation (the bonfire). Most prominently, we see that the style colors (orange and black) do not cover the background when the content style ratio was set 10 to 1.5. Furthermore, the contours of the object in the content image are preserved more clearly when the style factor is decreased. Personally we prefer the content style ratio of 1 to 1, but this is just a matter of taste.

3 Integration of the Deep Dream and Style Transfer Algorithms

For this section we adjusted the deep dream algorithm to make it dream recursively and then merged it with the style transfer algorithm functionality. Furthermore, we performed a GIF maker function on the generated results, thereby creating animated artworks.

3.1 Deep Dream Inception

We have altered the deep dream code to create a deep dream inception (not to be confused with the inception model itself, but in the sense of creating new outputs with previous outputs, resulting in a recursive dream-within-dream inception). Instead of only using one layer of the inception model and maximizing the score of the layer to change the values in the input image, we used the blended image of a layer as input for the next layer. We repeated this process until the last layer of the inception model, thereby generating a recursive deep dream. Because all layers of the inception model are maximized one after each other we expected that the generated images would create a new effect. To start with the deep dream inception we used the image in figure 15 (top-left) as input.

By altering the code to create a deep dream inception (from layer 1 till layer 11) we created the following images shown in figure 15. We did not use the first (0th) layer, because this messes up the starting image by making the borders of the tiles too visible.

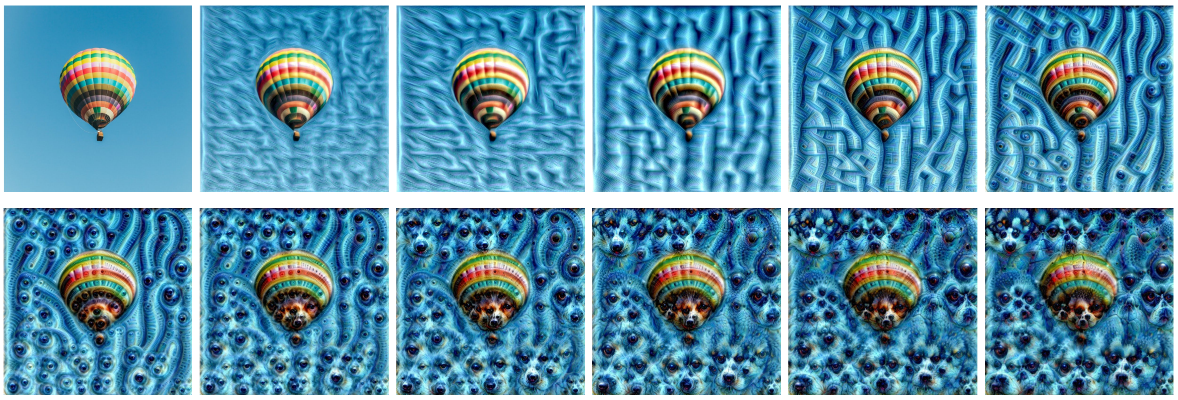


Figure 15: *The outputs for every subsequent layer (from layer 1 till 11) blended with the previous layer.*

Since every layer amplifies the previous layer, the patterns in the images will get amplified more and more, resulting in a clearly different output compared with the standard deep dream method which only uses one layer to amplify the original image. See figure 16 for a comparison.

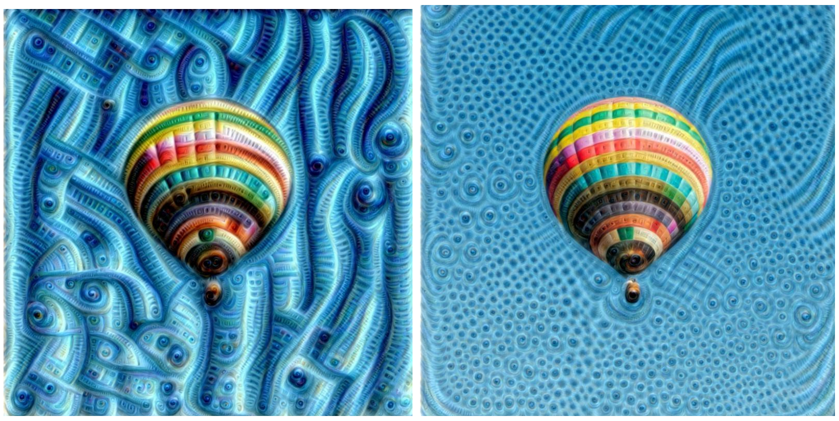


Figure 16: *The outputs of our inception deep dream (left) using all of the layers from 1 till 5, and the standard deep dream (right), using only the 5th layer of the inception model*

3.1.1 Tweaking the Feature Channels

Since we did not want to end up with only dogfaces (see figure 15, bottom-right) the feature channels of the last 5 layers were randomly chosen so that only a certain amount of them is used for amplification. Looking at the results in figure 17 below we can conclude that this method succeed in getting rid of the constant appearance of the dogs. Instead of dogs, more abstract patterns have arisen.

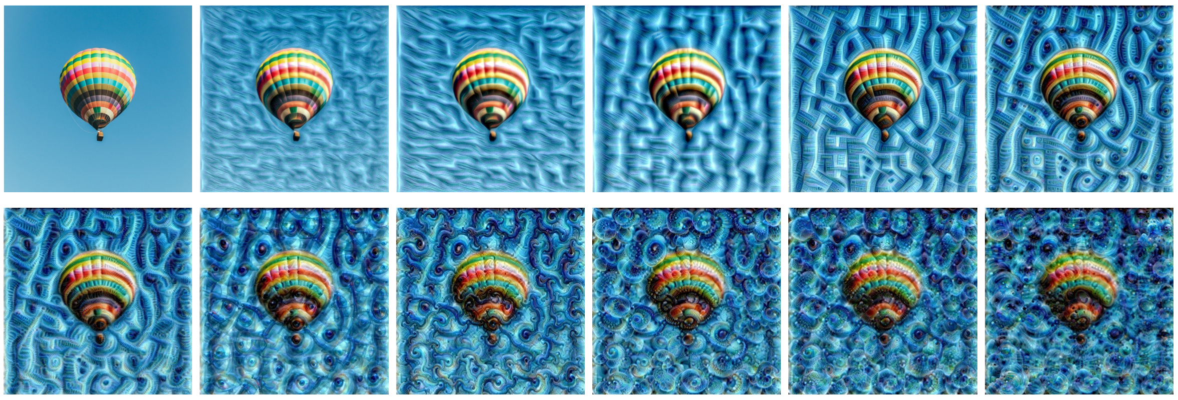


Figure 17: *The input image (top left) and the resulting output images of every layer that are automatically blended with the output of the previous layer. From the 6th layer on, the feature channels are chosen randomly (bottom row).*

3.2 Deep Dream Style Transfer

After we generated the deep dream inception images with our recursive algorithm, we used the style transfer algorithm to experiment with the style of the recursive deep dream images to create a unique effects. For this experiment we used an art piece of Karel Appel named *Little Boy 1969,* see figure 18.

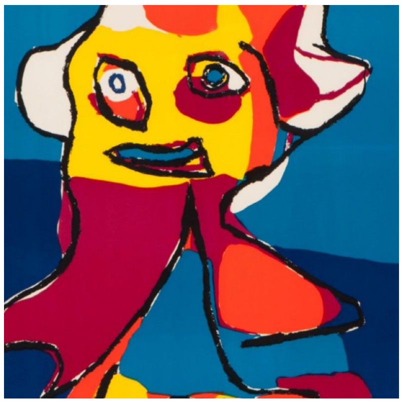


Figure 18:*The style image used for the style transfer algorithm.*

After the style transfer process we got the following results found in figure 19. The results were generated with the following parameter settings: the gradient descent optimization method, 100 iterations, a step size of 10.0 for the gradient in each of the iterations, no maximum width/height for the style image, a denoising factor of 0.3 and a ratio of style to content loss of 10.0 to 1.5.

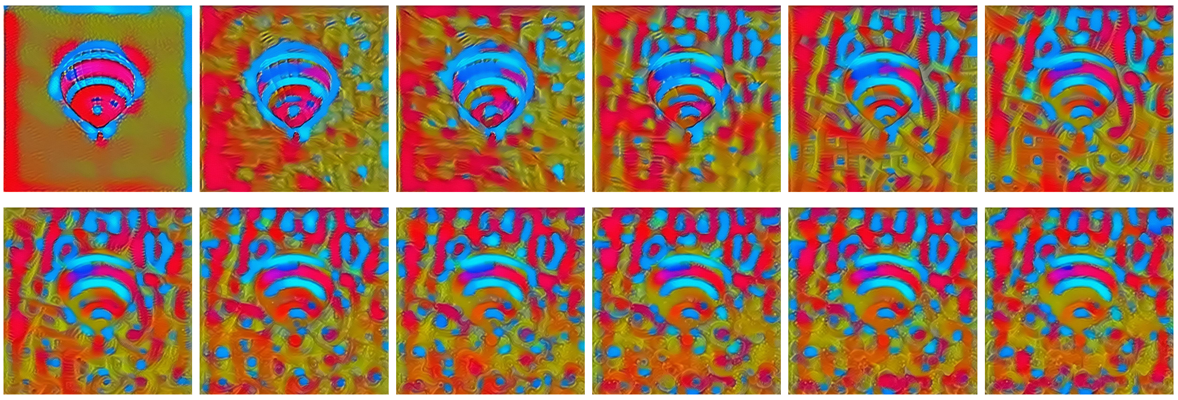


Figure 19: *output of every layer after the style transfer process of the deep dream images.*

The results are rather disappointing in our opinion. The generated images are not very similar to Karel Appel's artwork and the contours of the content image are not preserved well. The results could be improved by adjusting the parameter settings, like increasing the iterations, decreasing the step size or adjusting the content/style ratio. The appropriate parameter settings highly depend on the input image as well as the style image, so tweaking the parameters is a matter of trial and error. Since running the algorithm for 100 iterations takes about 6 hours (on a 2013 MacBook Pro) we managed to run the experiment only 3 times with different parameters. Therefore, the result above are unfortunately of unsatisfiable quality.

3.3 Deep Dream Style Transfer GIF

Next, we wanted to animate the images to visualise the changes the algorithms makes in the images, thereby creating a dynamic effect. To achieve this effect we implemented a function based on the code of Alex Buzunov (Quora, 2018) to create GIFs from the images that the deep dream inception algorithm produced. The results can be found in the code folder.

References

Gatys, L. A., Ecker, A. S., & Bethge, M. (2015). A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576.*

Gatys, L. A., Ecker, A. S., & Bethge, M. (2016, June). Image style transfer using convolutional neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2016 IEEE Conference on* (pp. 2414-2423). IEEE.

Mahendran, A., & Vedaldi, A. (2015). Understanding deep image representations by inverting them.

Quora. (2018). How do I create a GIF file from a bunch of JPEG files using Python?. [online] Available at: https://www.quora.com/How-do-I-create-a-GIF-file-from-a-bunch-of-JPEG-files-using-Python [Accessed 11 May 2018].

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015, June). Going deeper with convolutions. Cvpr.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 2818-2826).