

Neural Style Transfer



L. A. Gatys, A. S. Ecker and M. Bethge, "Image Style Transfer Using Convolutional Neural Networks," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, 2016, pp. 2414-2423, doi: 10.1109/CVPR.2016.265.

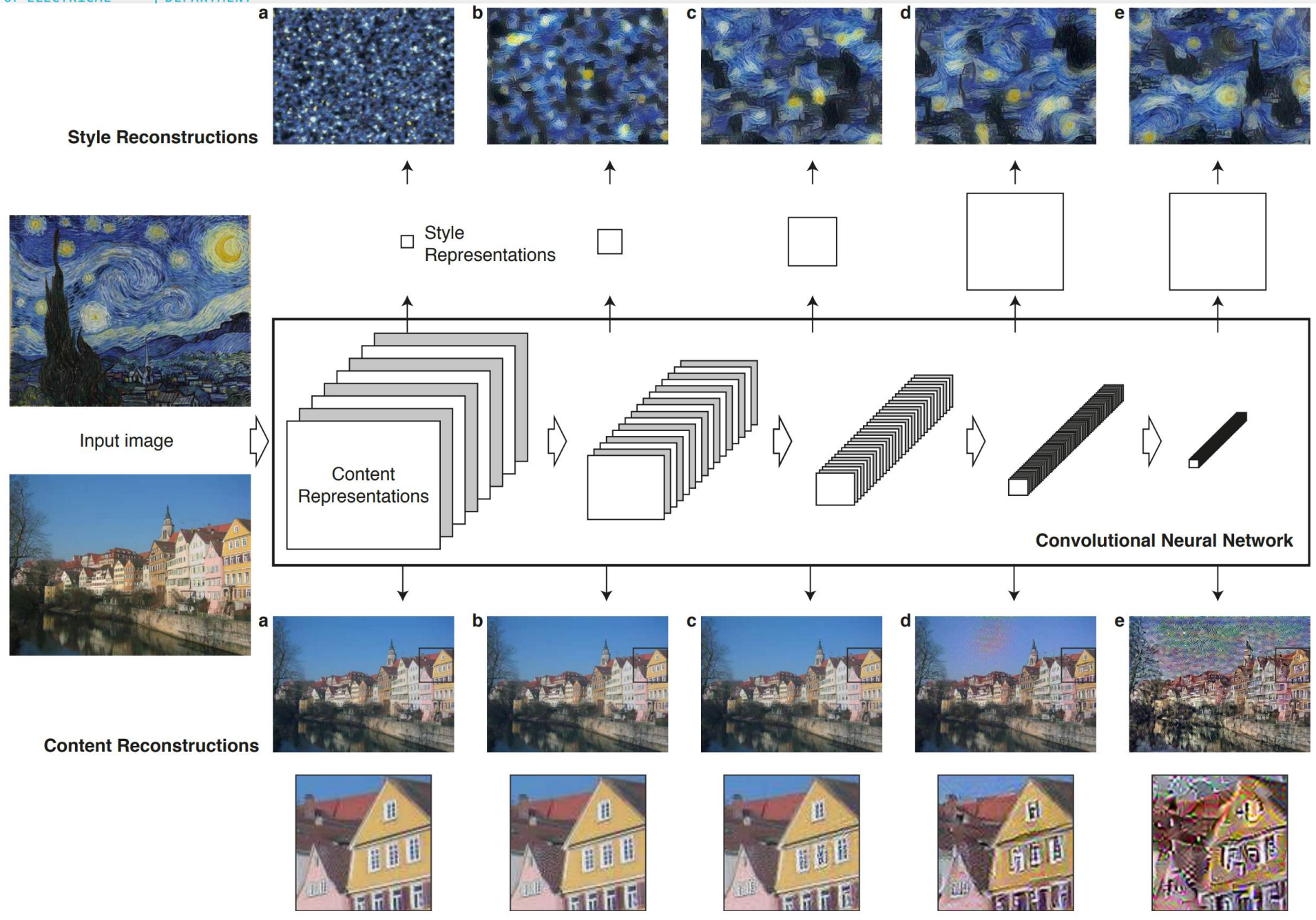
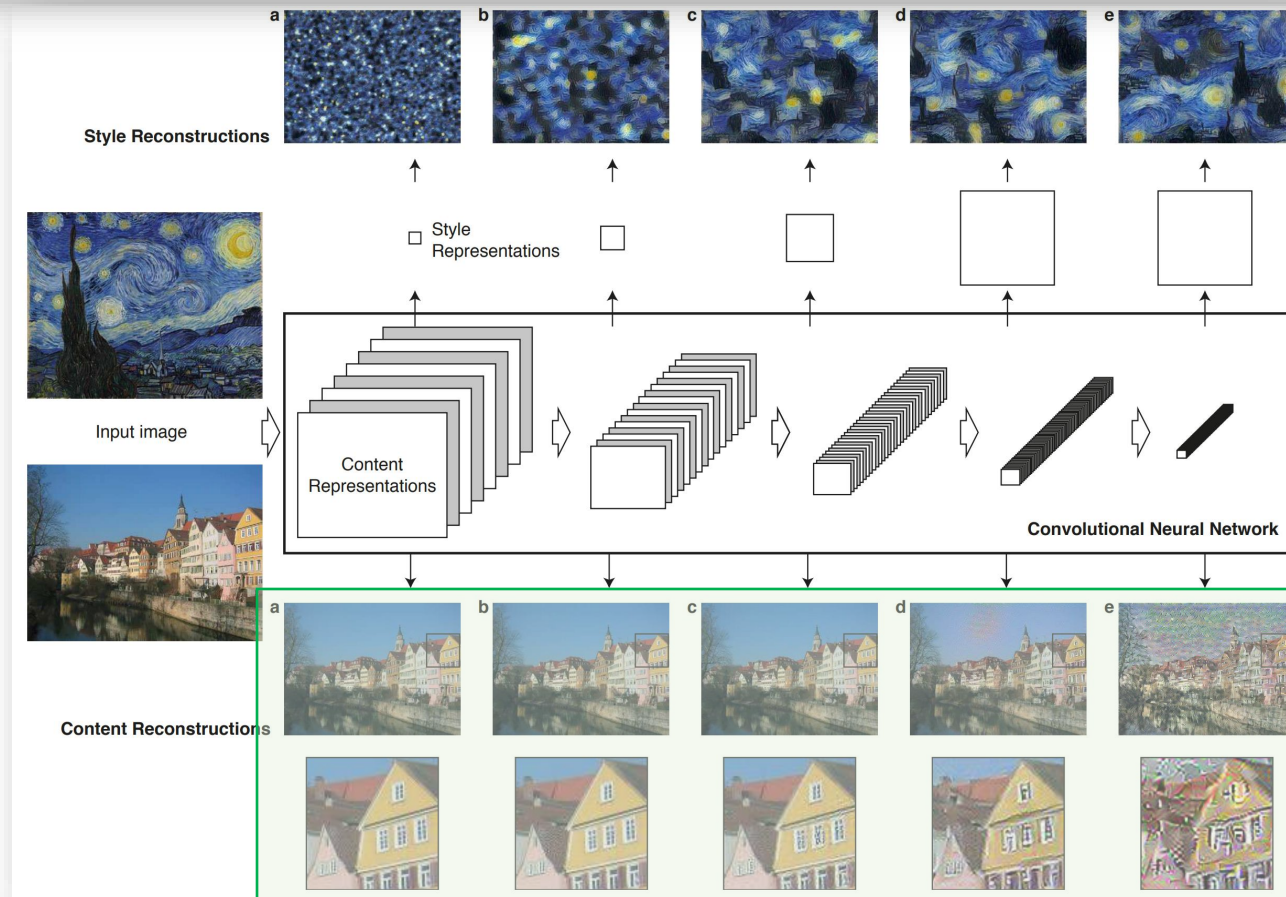
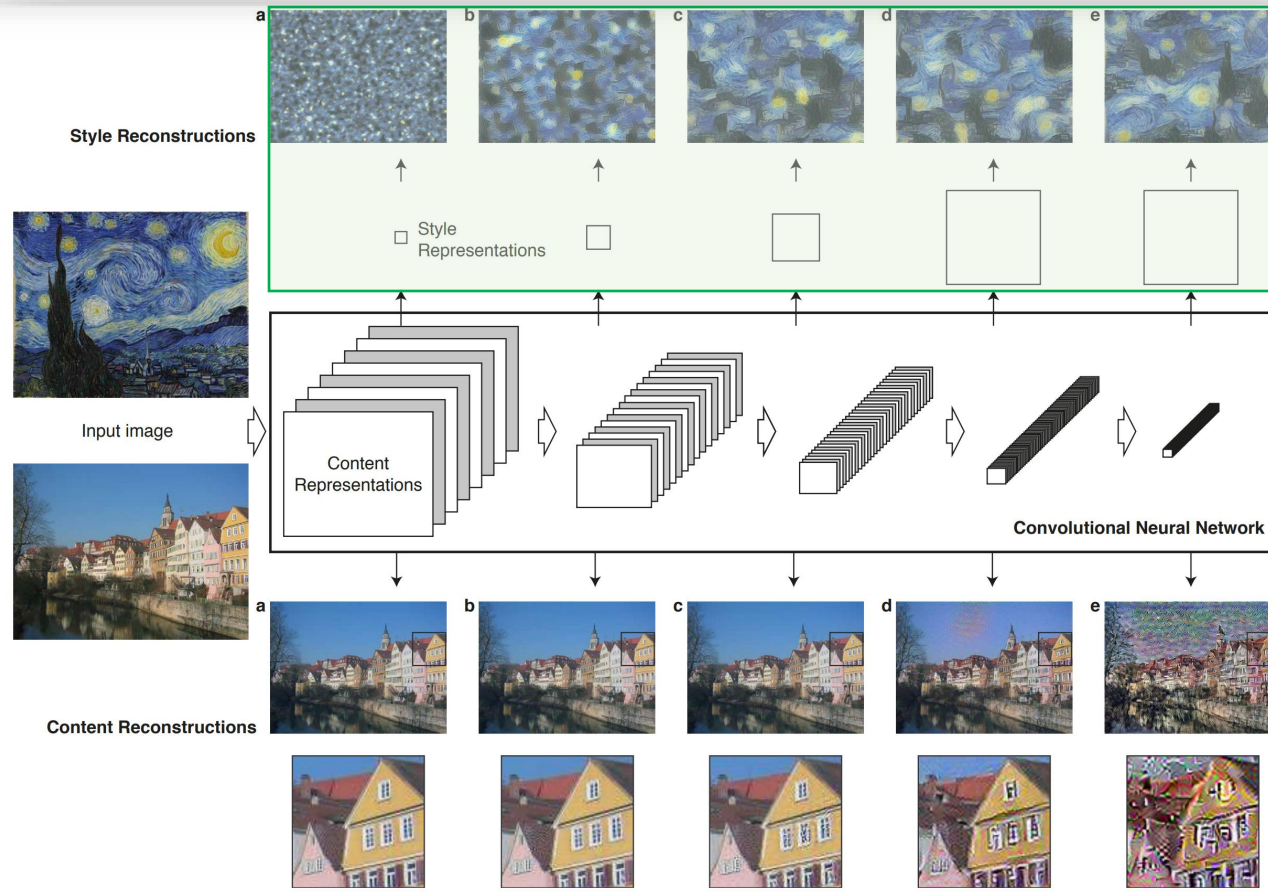


Figure 1. Image representations in a Convolutional Neural Network (CNN). A given input image is represented as a set of filtered images at each processing stage in the CNN. While the number of different filters increases along the processing hierarchy, the size of the filtered images is reduced by some downsampling mechanism (e.g. max-pooling) leading to a decrease in the total number of units per layer of the network. **Content Reconstructions.** We can visualise the information at different processing stages in the CNN by reconstructing the input image from only knowing the network's responses in a particular layer. We reconstruct the input image from from layers 'conv1_2' (a), 'conv2_2' (b), 'conv3_2' (c), 'conv4_2' (d) and 'conv5_2' (e) of the original VGG-Network. We find that reconstruction from lower layers is almost perfect (a–c). In higher layers of the network, detailed pixel information is lost while the high-level content of the image is preserved



Style Reconstructions. On top of the original CNN activations we use a feature space that captures the texture information of an input image. The style representation computes correlations between the different features in different layers of the CNN. We reconstruct the style of the input image from a style representation built on different subsets of CNN layers ('conv1_1' (a), 'conv1_1' and 'conv2_1' (b), 'conv1_1', 'conv2_1' and 'conv3_1' (c), 'conv1_1', 'conv2_1', 'conv3_1' and 'conv4_1' (d), 'conv1_1', 'conv2_1', 'conv3_1', 'conv4_1' and 'conv5_1' (e). This creates images that match the style of a given image on an increasing scale while discarding information of the global arrangement of the scene.



Style Reconstructions. On top of the original CNN activations we use a feature space that captures the texture information of an input image. The style representation computes correlations between the different features in different layers of the CNN. We reconstruct the style of the input image from a style representation built on different subsets of CNN layers (‘conv1_1’ (a), ‘conv1_1’ and ‘conv2_1’ (b), ‘conv1_1’, ‘conv2_1’ and ‘conv3_1’ (c), ‘conv1_1’, ‘conv2_1’, ‘conv3_1’ and ‘conv4_1’ (d), ‘conv1_1’, ‘conv2_1’, ‘conv3_1’, ‘conv4_1’ and ‘conv5_1’ (e). This creates images that match the style of a given image on an increasing scale while discarding information of the global arrangement of the scene.

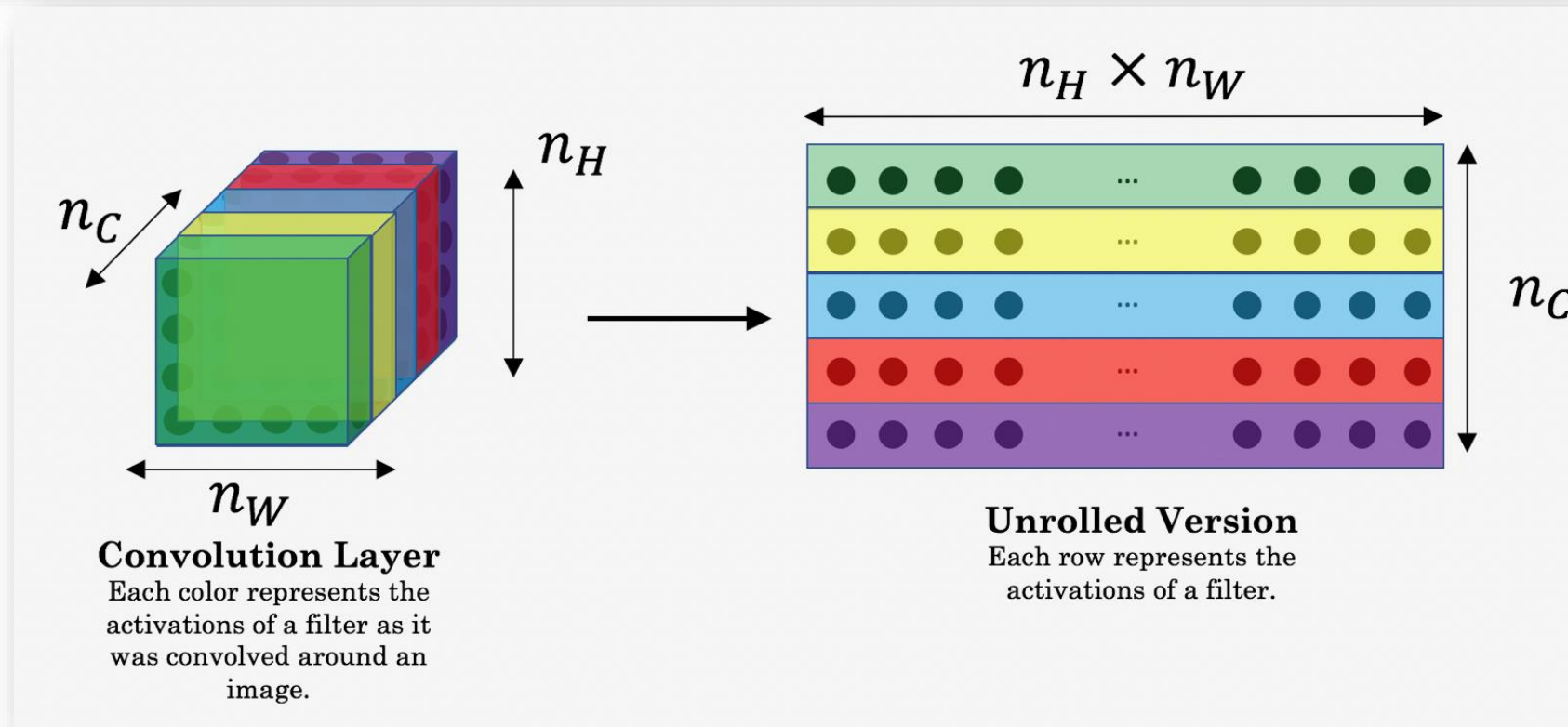
2.2. Style representation

To obtain a representation of the *style* of an input image, we use a feature space designed to capture texture information [10]. This feature space can be built on top of the filter responses in any layer of the network. It consists of the correlations between the different filter responses, where the expectation is taken over the spatial extent of the feature maps. These feature correlations are given by the Gram matrix $G^l \in \mathcal{R}^{N_l \times N_l}$, where G_{ij}^l is the inner product between the vectorised feature maps i and j in layer l :

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l. \quad (3)$$

By including the feature correlations of multiple layers, we obtain a stationary, multi-scale representation of the input image, which captures its texture information but not the global arrangement. Again, we can visualise the informa-

Style Reconstructions. On top of the original CNN activations we use a feature space that captures the texture information of an input image. The style representation computes correlations between the different features in different layers of the CNN. We reconstruct the style of the input image from a style representation built on different subsets of CNN layers ('conv1_1' (a), 'conv1_1' and 'conv2_1' (b), 'conv1_1', 'conv2_1' and 'conv3_1' (c), 'conv1_1', 'conv2_1', 'conv3_1' and 'conv4_1' (d), 'conv1_1', 'conv2_1', 'conv3_1', 'conv4_1' and 'conv5_1' (e). This creates images that match the style of a given image on an increasing scale while discarding information of the global arrangement of the scene.



Style Reconstructions. On top of the original CNN activations we use a feature space that captures the texture information of an input image. The style representation computes **correlations between the different features in different layers of the CNN.** We reconstruct the style of the input image from a style representation built on different subsets of CNN layers ('conv1_1' (a), 'conv1_1' and 'conv2_1' (b), 'conv1_1', 'conv2_1' and 'conv3_1' (c), 'conv1_1', 'conv2_1', 'conv3_1' and 'conv4_1' (d), 'conv1_1', 'conv2_1', 'conv3_1', 'conv4_1' and 'conv5_1' (e). This creates images that match the style of a given image on an increasing scale while discarding information of the global arrangement of the scene.

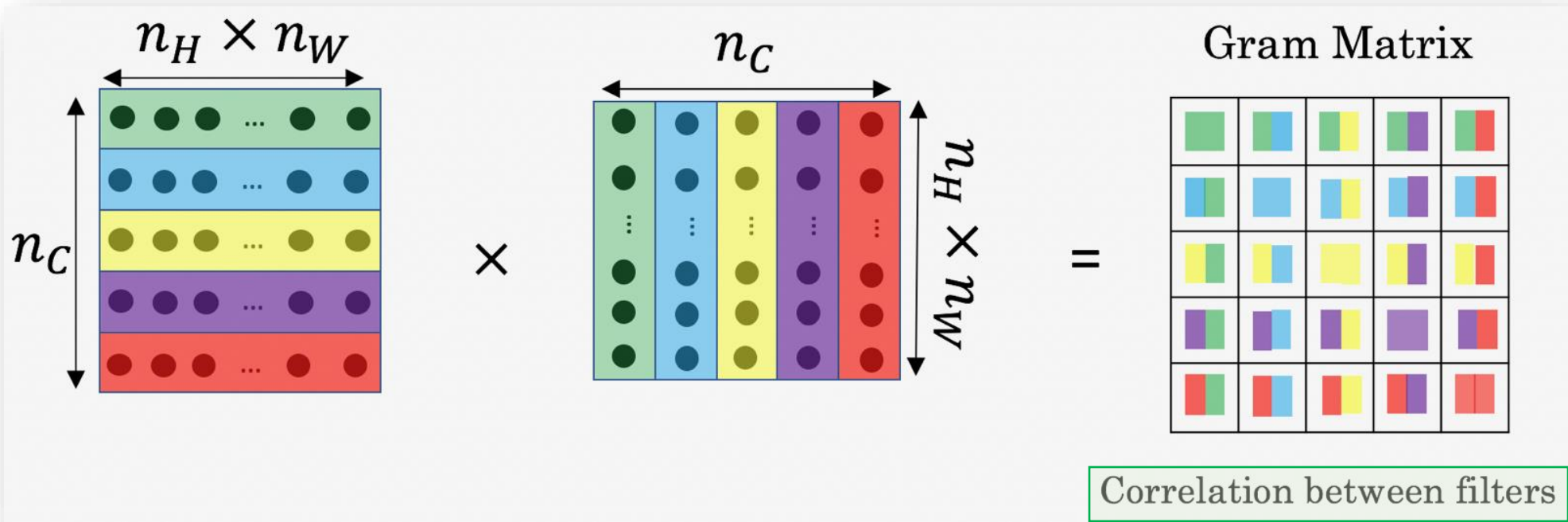


Figure 2. Style transfer algorithm. First content and style features are extracted and stored. The style image \vec{a} is passed through the network and its style representation A^l on all layers included are computed and stored (left). The content image \vec{p} is passed through the network and the content representation P^l in one layer is stored (right). Then a random white noise image \vec{x} is passed through the network and its style features G^l and content features F^l are computed. On each layer included in the style representation, the element-wise mean squared difference between G^l and A^l is computed to give the style loss \mathcal{L}_{style} (left). Also the mean squared difference between F^l and P^l is computed to give the content loss $\mathcal{L}_{content}$ (right). The total loss \mathcal{L}_{total} is then a linear combination between the content and the style loss. Its derivative with respect to the pixel values can be computed using error back-propagation (middle). This gradient is used to iteratively update the image \vec{x} until it simultaneously matches the style features of the style image \vec{a} and the content features of the content image \vec{p} (middle, bottom).

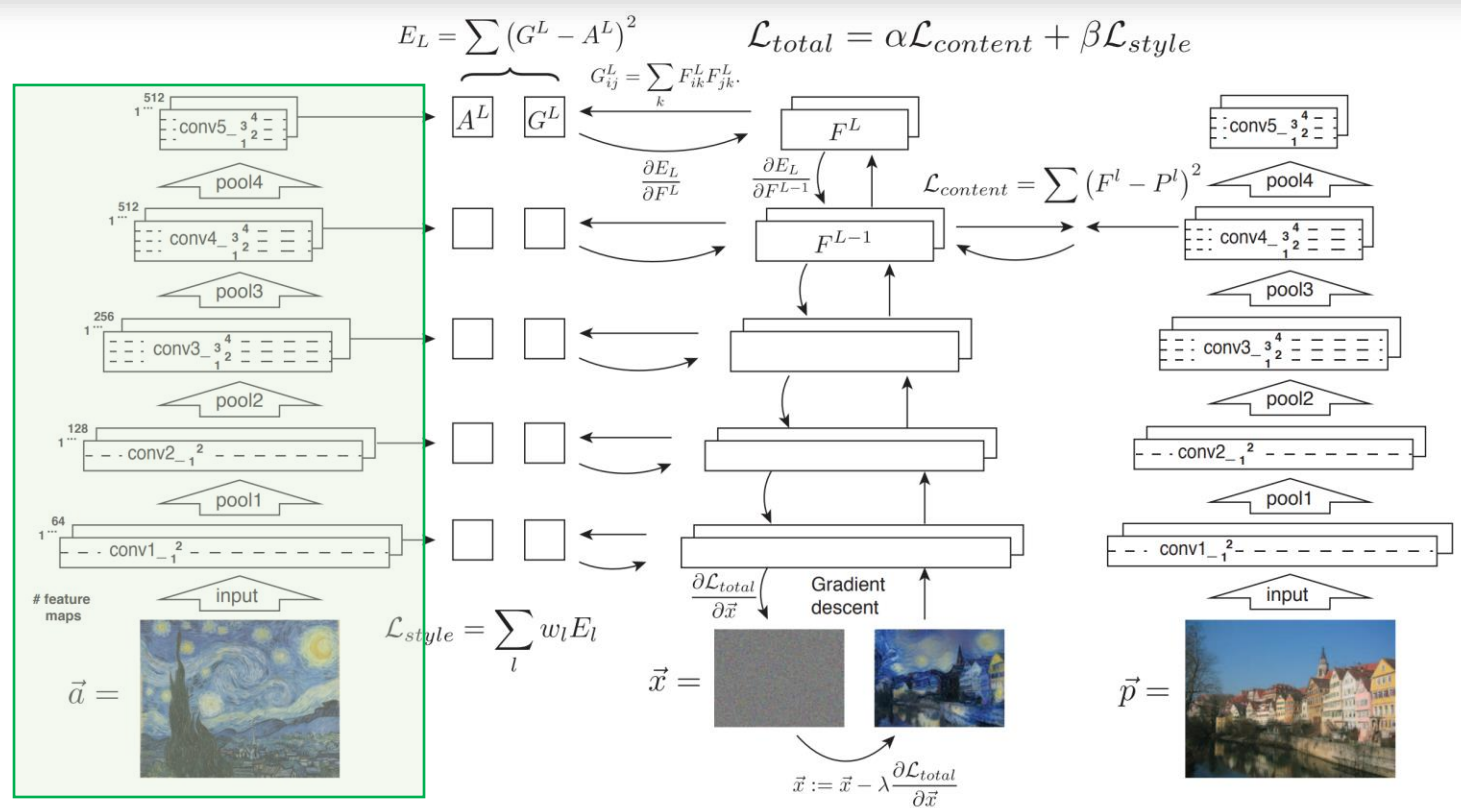


Figure 2. Style transfer algorithm. First content and style features are extracted and stored. The style image \vec{a} is passed through the network and its style representation A^l on all layers included are computed and stored (left). The content image \vec{p} is passed through the network and the content representation P^l in one layer is stored (right). Then a random white noise image \vec{x} is passed through the network and its style features G^l and content features F^l are computed. On each layer included in the style representation, the element-wise mean squared difference between G^l and A^l is computed to give the style loss \mathcal{L}_{style} (left). Also the mean squared difference between F^l and P^l is computed to give the content loss $\mathcal{L}_{content}$ (right). The total loss \mathcal{L}_{total} is then a linear combination between the content and the style loss. Its derivative with respect to the pixel values can be computed using error back-propagation (middle). This gradient is used to iteratively update the image \vec{x} until it simultaneously matches the style features of the style image \vec{a} and the content features of the content image \vec{p} (middle, bottom).

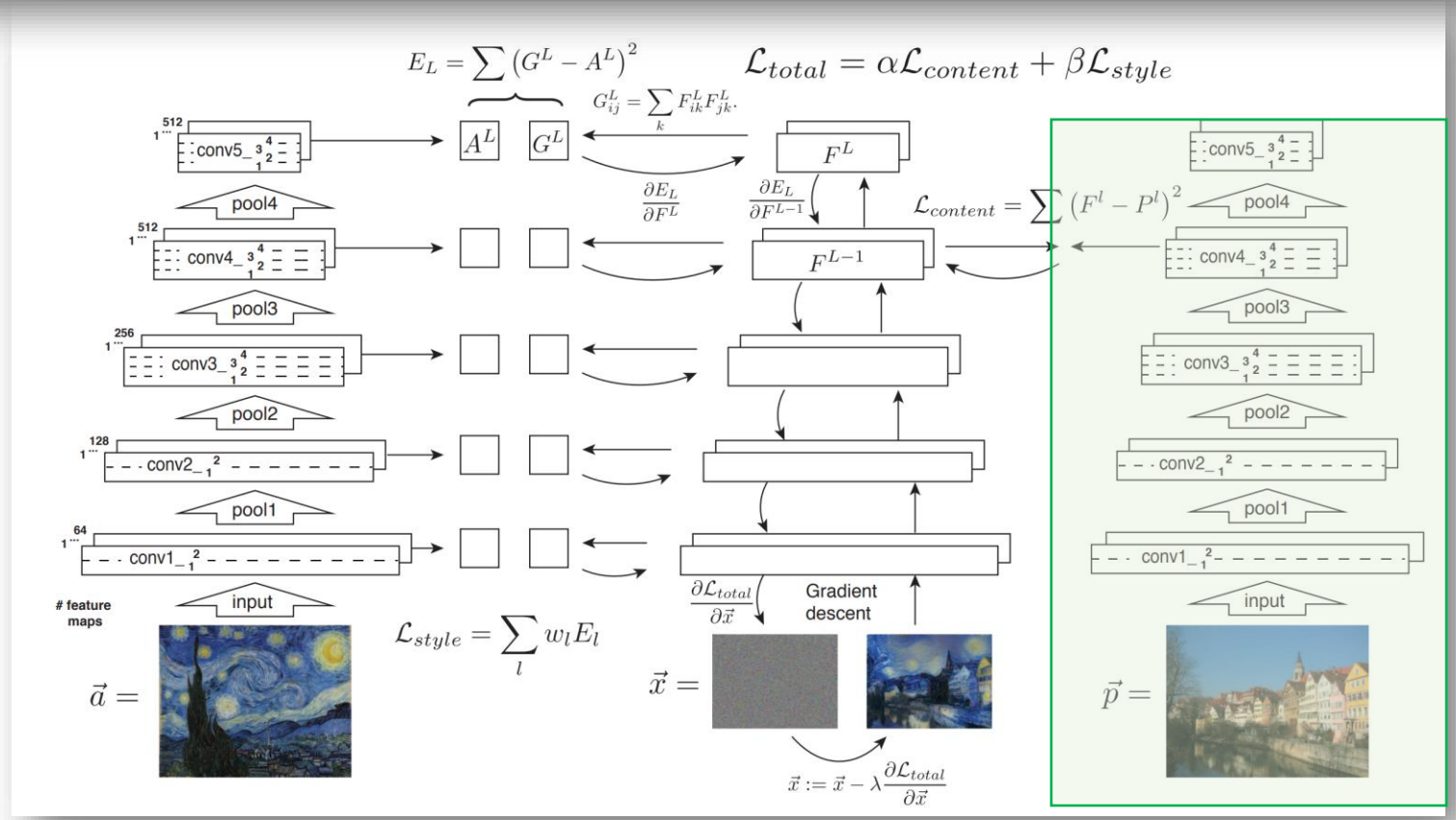


Figure 2. Style transfer algorithm. First content and style features are extracted and stored. The style image \vec{a} is passed through the network and its style representation A^l on all layers included are computed and stored (left). The content image \vec{p} is passed through the network and the content representation P^l in one layer is stored (right). Then a random white noise image \vec{x} is passed through the network and its style features G^l and content features F^l are computed. On each layer included in the style representation, the element-wise mean squared difference between G^l and A^l is computed to give the style loss \mathcal{L}_{style} (left). Also the mean squared difference between F^l and P^l is computed to give the content loss $\mathcal{L}_{content}$ (right). The total loss \mathcal{L}_{total} is then a linear combination between the content and the style loss. Its derivative with respect to the pixel values can be computed using error back-propagation (middle). This gradient is used to iteratively update the image \vec{x} until it simultaneously matches the style features of the style image \vec{a} and the content features of the content image \vec{p} (middle, bottom).

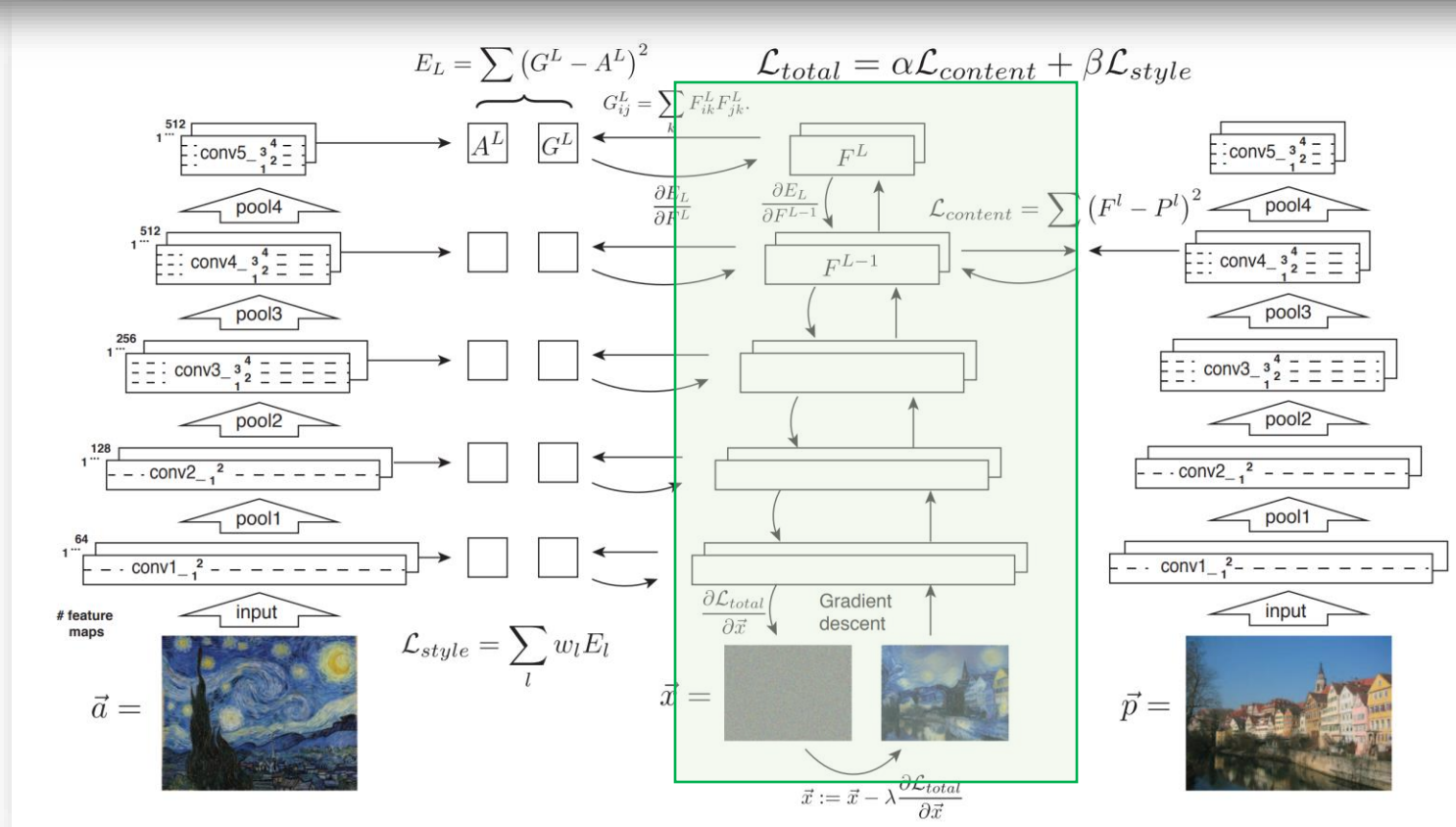


Figure 2. Style transfer algorithm. First content and style features are extracted and stored. The style image \vec{a} is passed through the network and its style representation A^l on all layers included are computed and stored (left). The content image \vec{p} is passed through the network and the content representation P^l in one layer is stored (right). Then a random white noise image \vec{x} is passed through the network and its style features G^l and content features F^l are computed. On each layer included in the style representation, the element-wise mean squared difference between G^l and A^l is computed to give the style loss \mathcal{L}_{style} (left). Also the mean squared difference between F^l and P^l is computed to give the content loss $\mathcal{L}_{content}$ (right). The total loss \mathcal{L}_{total} is then a linear combination between the content and the style loss. Its derivative with respect to the pixel values can be computed using error back-propagation (middle). This gradient is used to iteratively update the image \vec{x} until it simultaneously matches the style features of the style image \vec{a} and the content features of the content image \vec{p} (middle, bottom).

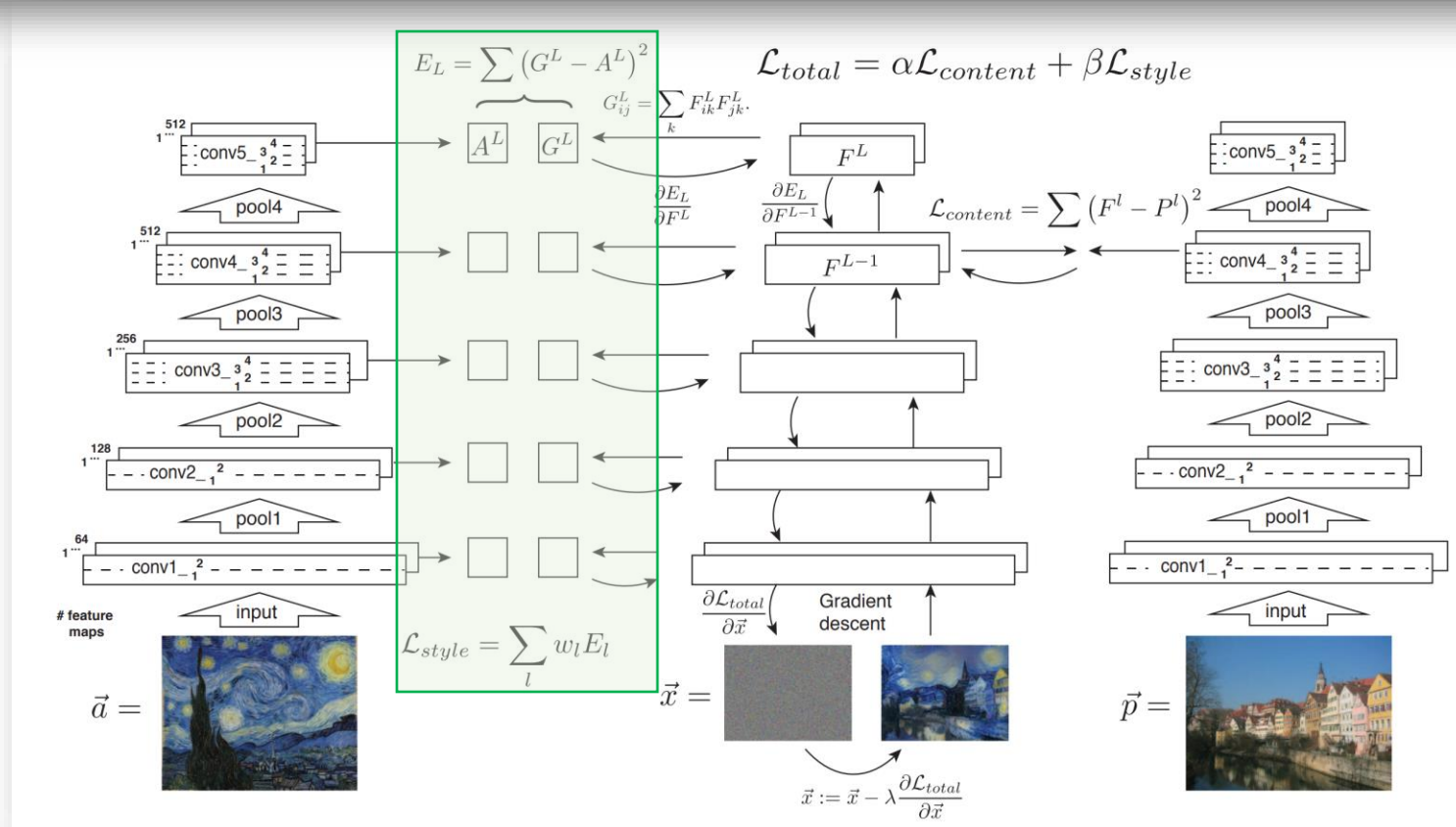


Figure 2. Style transfer algorithm. First content and style features are extracted and stored. The style image \vec{a} is passed through the network and its style representation A^l on all layers included are computed and stored (left). The content image \vec{p} is passed through the network and the content representation P^l in one layer is stored (right). Then a random white noise image \vec{x} is passed through the network and its style features G^l and content features F^l are computed. On each layer included in the style representation, the element-wise mean squared difference between G^l and A^l is computed to give the style loss \mathcal{L}_{style} (left). Also the mean squared difference between F^l and P^l is computed to give the content loss $\mathcal{L}_{content}$ (right). The total loss \mathcal{L}_{total} is then a linear combination between the content and the style loss. Its derivative with respect to the pixel values can be computed using error back-propagation (middle). This gradient is used to iteratively update the image \vec{x} until it simultaneously matches the style features of the style image \vec{a} and the content features of the content image \vec{p} (middle, bottom).

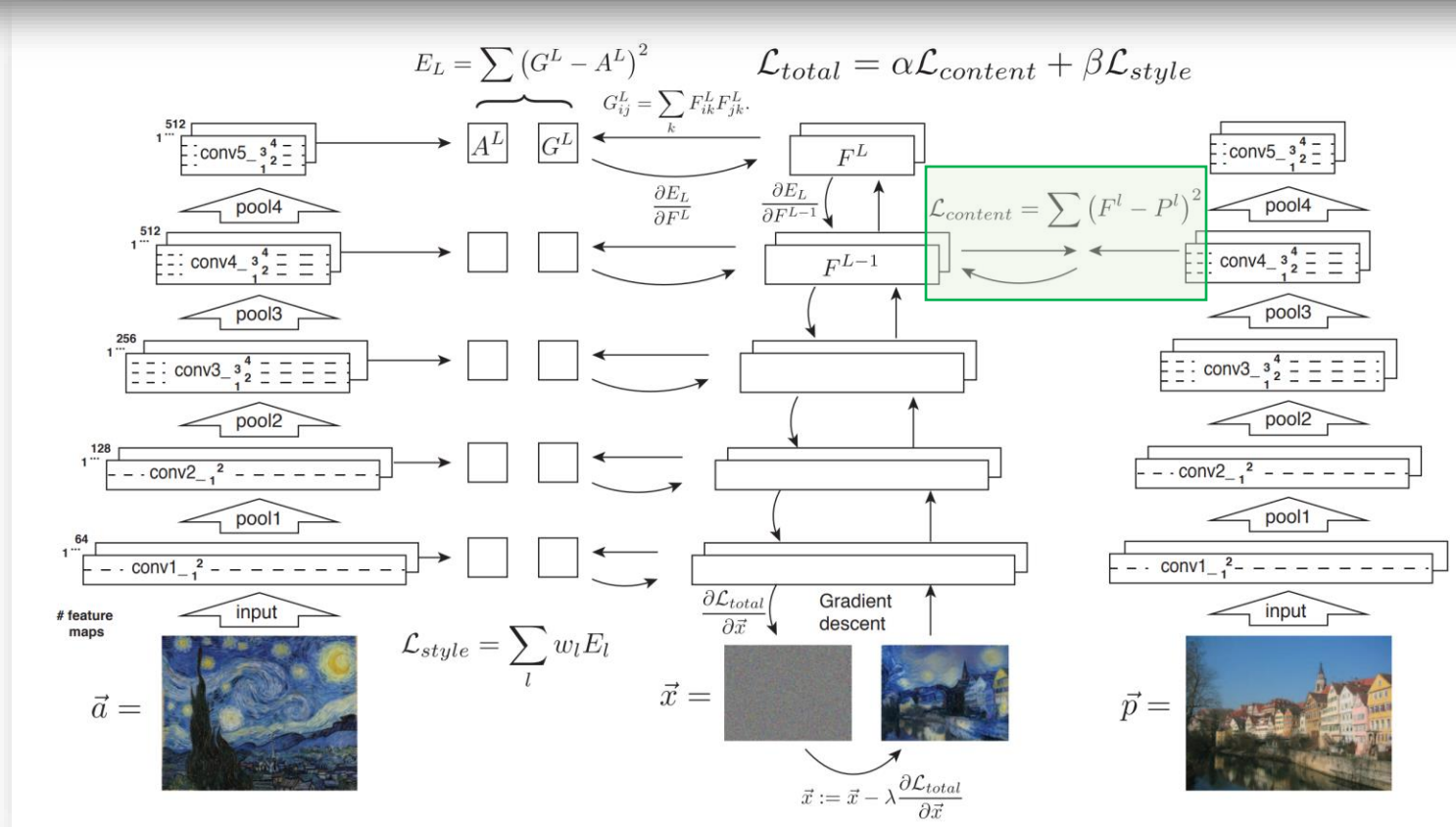
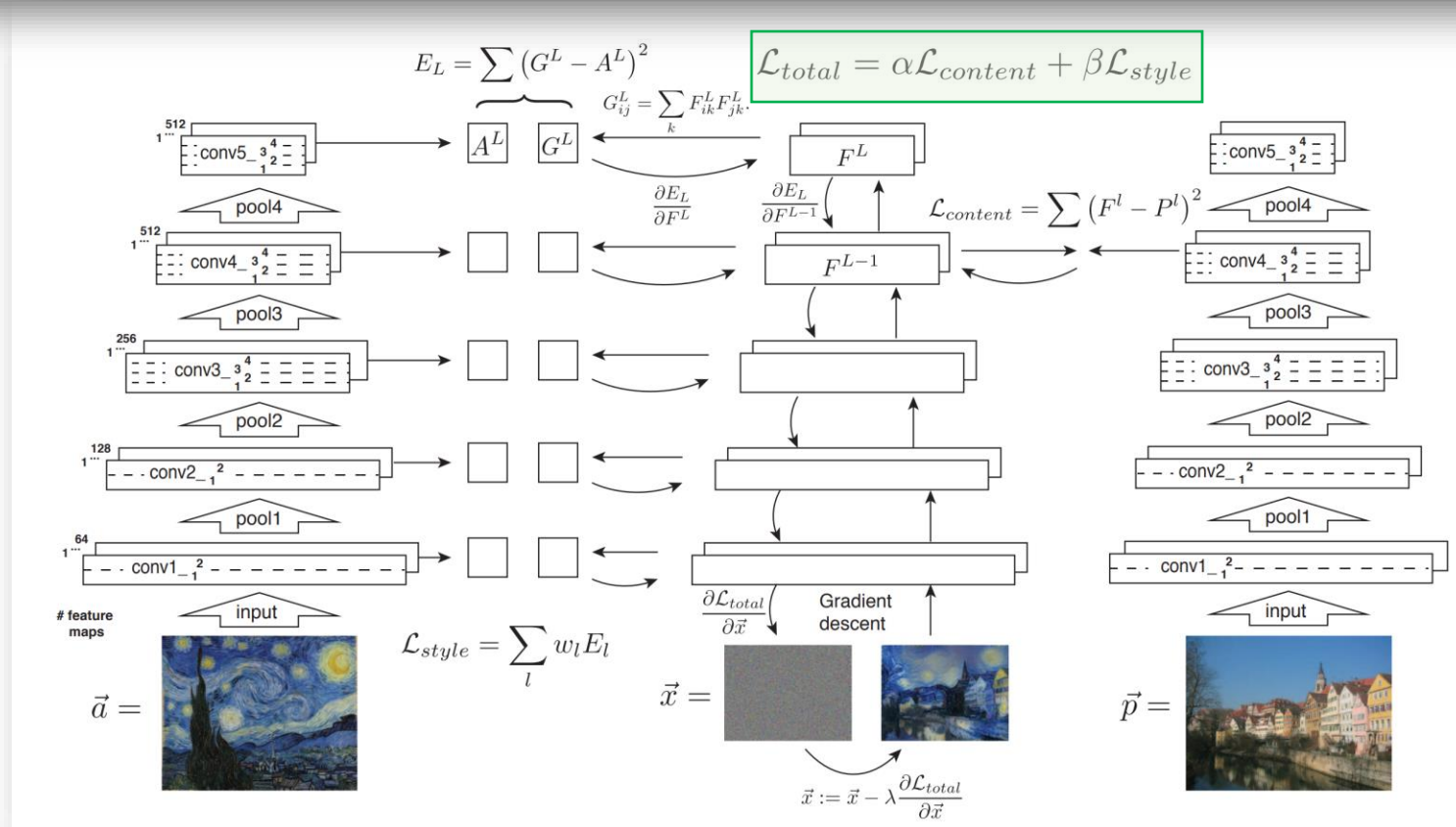


Figure 2. Style transfer algorithm. First content and style features are extracted and stored. The style image \vec{a} is passed through the network and its style representation A^l on all layers included are computed and stored (left). The content image \vec{p} is passed through the network and the content representation P^l in one layer is stored (right). Then a random white noise image \vec{x} is passed through the network and its style features G^l and content features F^l are computed. On each layer included in the style representation, the element-wise mean squared difference between G^l and A^l is computed to give the style loss \mathcal{L}_{style} (left). Also the mean squared difference between F^l and P^l is computed to give the content loss $\mathcal{L}_{content}$ (right). The total loss \mathcal{L}_{total} is then a linear combination between the content and the style loss. Its derivative with respect to the pixel values can be computed using error back-propagation (middle). This gradient is used to iteratively update the image \vec{x} until it simultaneously matches the style features of the style image \vec{a} and the content features of the content image \vec{p} (middle, bottom).



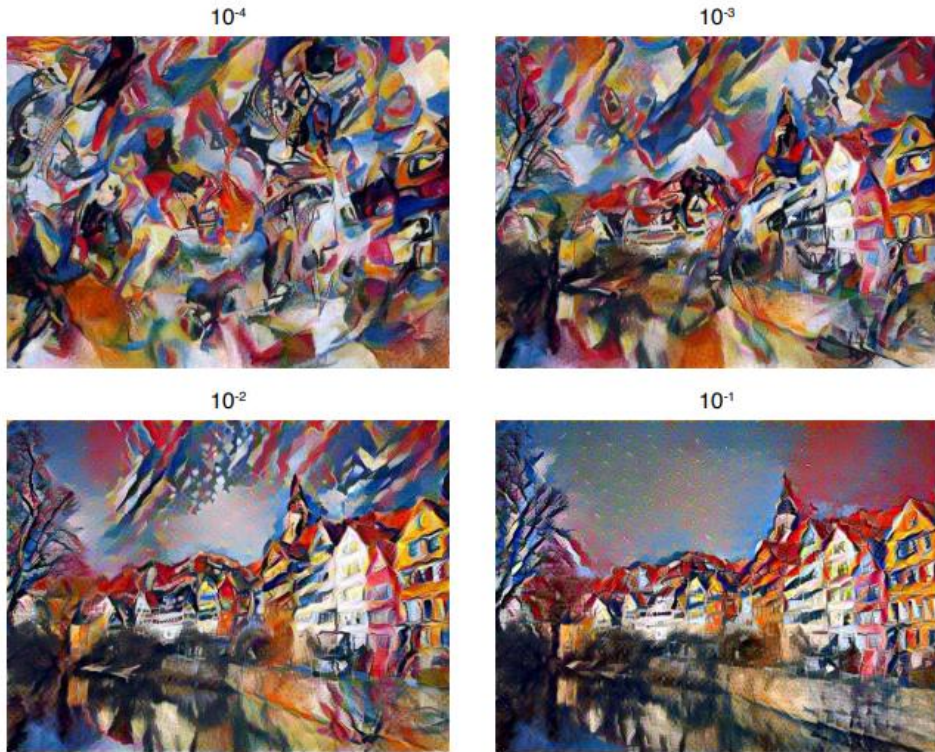
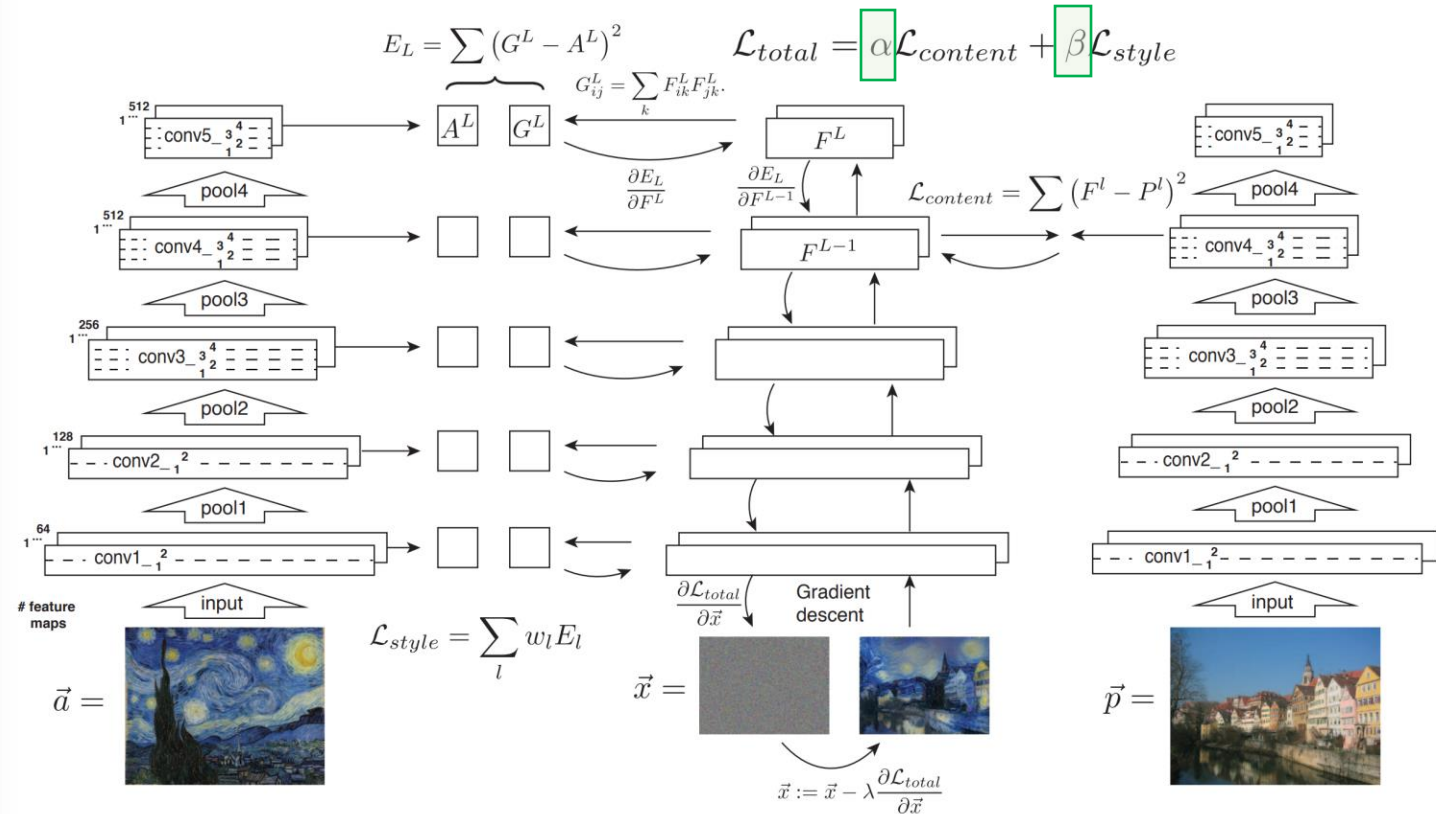


Figure 4. Relative weighting of matching content and style of the respective source images. The ratio α/β between matching the content and matching the style increases from top left to bottom right. A high emphasis on the style effectively produces a textured version of the style image (top left). A high emphasis on the content produces an image with only little stylisation (bottom right). In practice one can smoothly interpolate between the two extremes.



https://pytorch.org/tutorials/advanced/neural_style_tutorial.html

PyTorch Recipes [+]

Introduction to PyTorch [-]

[-] Learn the Basics

Quickstart

Tensors

Datasets & DataLoaders

Transforms

Build the Neural Network

Automatic Differentiation with `torch.autograd`

Optimizing Model Parameters

Save and Load the Model

[+] Introduction to PyTorch - YouTube Series

Learning PyTorch [+]

Image and Video [+]

Audio [+]

Backends [+]

Reinforcement Learning [+]

Tutorials > Neural Transfer Using PyTorch



Neural Transfer Using PyTorch

Author: Alexis Jacq

Edited by: Winston Herring

Introduction

This tutorial explains how to implement the **Neural-Style algorithm** developed by Leon A. Gatys, Alexander S. Ecker and Matthias Bethge. Neural-Style, or Neural-Transfer, allows you to take an image and reproduce it with a new artistic style. The algorithm takes three images, an input image, a content-image, and a style-image, and changes the input to resemble the content of the content-image and the artistic style of the style-image.



Perceptual Losses for Real-Time Style Transfer and Super-Resolution

Justin Johnson, Alexandre Alahi, Li Fei-Fei
{jcjohns, alahi, feifeili}@cs.stanford.edu

Department of Computer Science, Stanford University

Abstract. We consider image transformation problems, where an input image is transformed into an output image. Recent methods for such problems typically train feed-forward convolutional neural networks using a *per-pixel* loss between the output and ground-truth images. Parallel work has shown that high-quality images can be generated by defining and optimizing *perceptual* loss functions based on high-level features extracted from pretrained networks. We combine the benefits of both approaches, and propose the use of perceptual loss functions for training feed-forward networks for image transformation tasks. We show results on image style transfer, where a feed-forward network is trained to solve the optimization problem proposed by Gatys *et al* in real-time. Compared to the optimization-based method, our network gives similar qualitative results but is three orders of magnitude faster. We also experiment with single-image super-resolution, where replacing a per-pixel loss with a perceptual loss gives visually pleasing results.

<https://arxiv.org/abs/1603.08155>

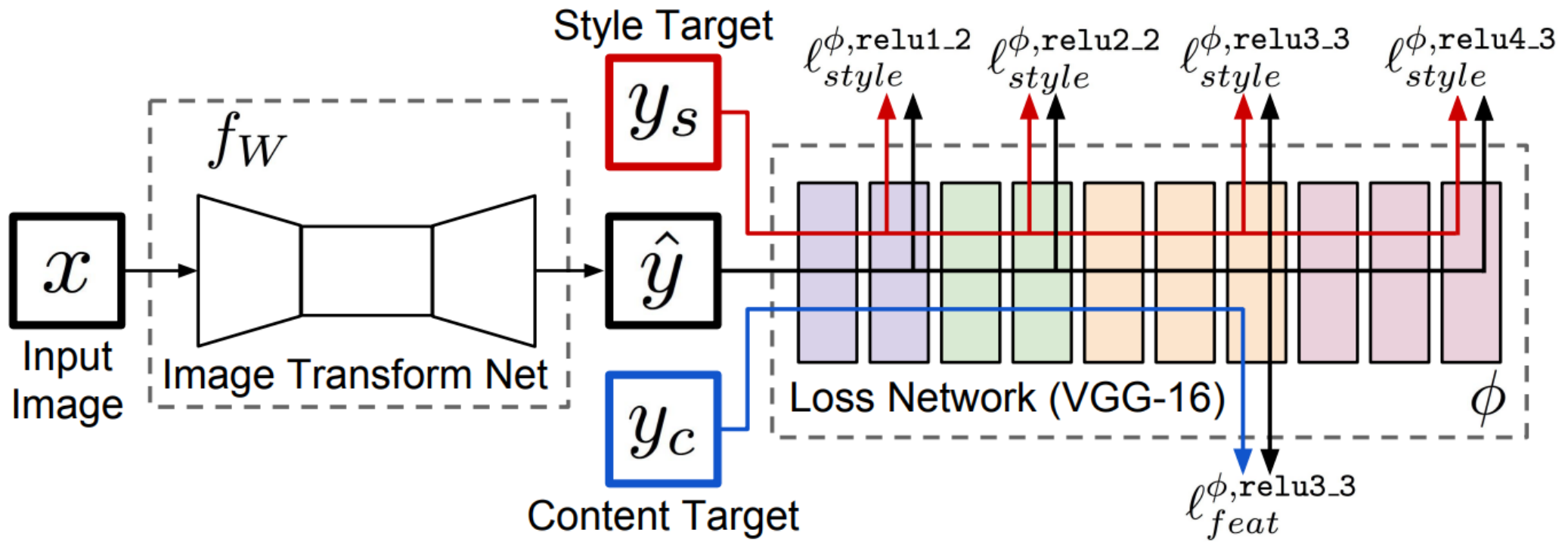


Fig. 2. System overview. We train an *image transformation network* to transform input images into output images. We use a *loss network* pretrained for image classification to define *perceptual loss functions* that measure perceptual differences in content and style between images. The loss network remains fixed during the training process.

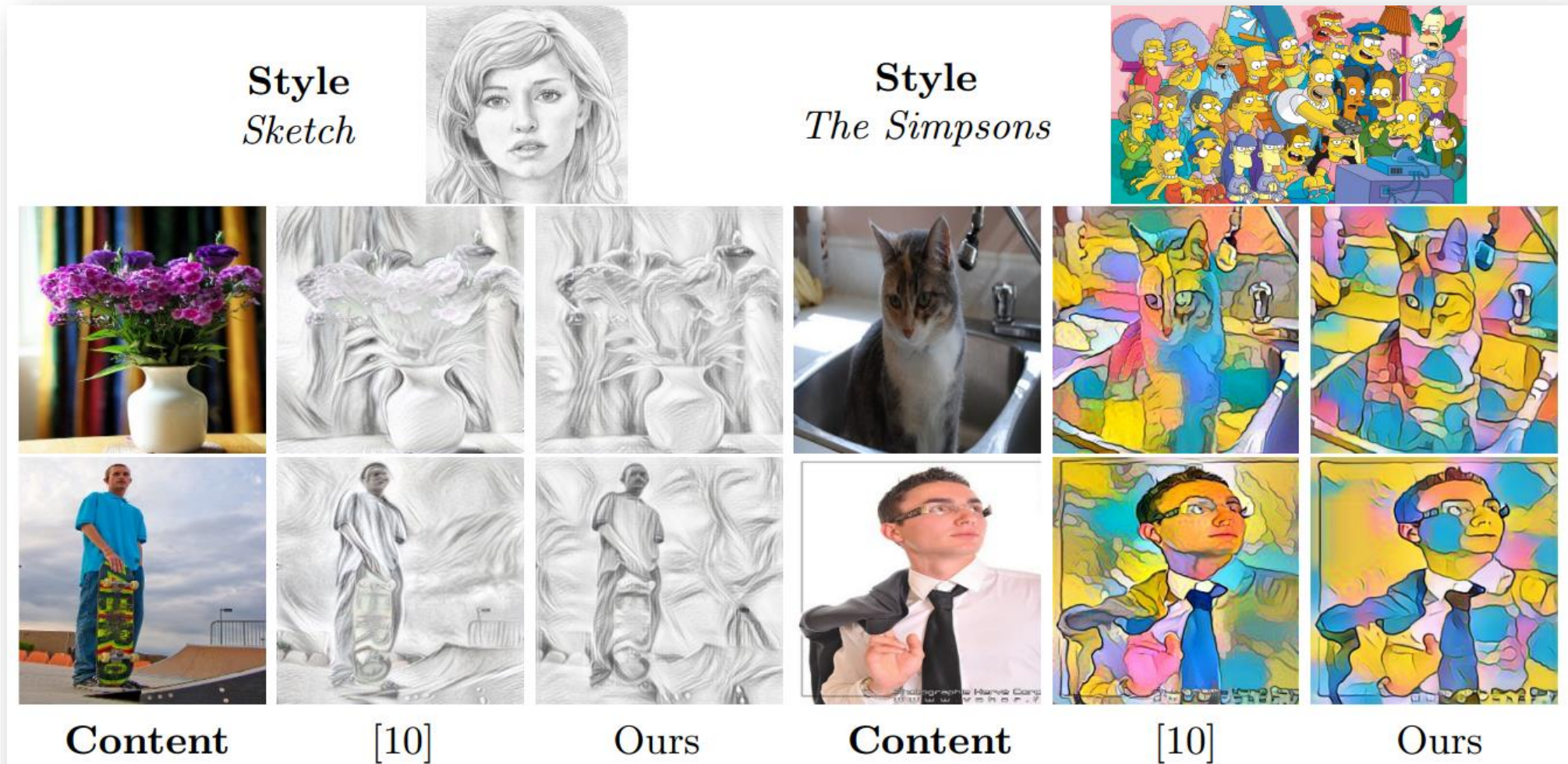


Fig. 6. Example results of style transfer using our image transformation networks. Our results are qualitatively similar to Gatys *et al* [10] but are much faster to generate (see Table 1). All generated images are 256×256 pixels.

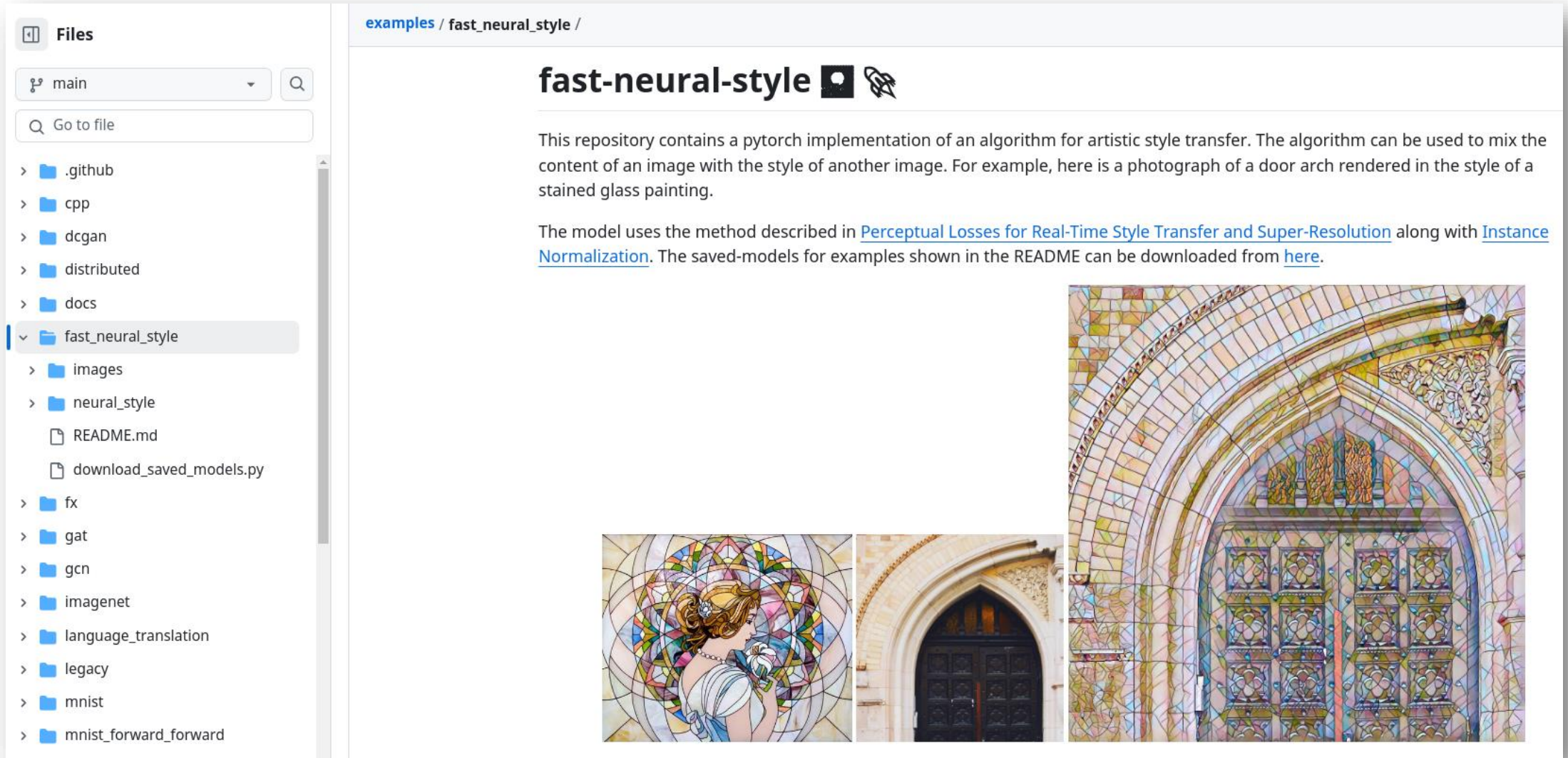


Image Size	Gatys <i>et al</i> [10]			Ours	Speedup		
	100	300	500		100	300	500
256 × 256	3.17	9.52s	15.86s	0.015s	212x	636x	1060x
512 × 512	10.97	32.91s	54.85s	0.05s	205x	615x	1026x
1024 × 1024	42.89	128.66s	214.44s	0.21s	208x	625x	1042x

Table 1. Speed (in seconds) for our style transfer network vs the optimization-based baseline for varying numbers of iterations and image resolutions. Our method gives similar qualitative results (see Figure 6) but is faster than a single optimization step of the baseline method. Both methods are benchmarked on a GTX Titan X GPU.

Fig. 6. Example results of style transfer using our image transformation networks. Our results are qualitatively similar to Gatys *et al* [10] but are much faster to generate (see Table 1). All generated images are 256 × 256 pixels.

https://github.com/pytorch/examples/tree/main/fast_neural_style



The screenshot shows the GitHub repository page for 'fast-neural-style'. On the left is a file explorer sidebar showing the directory structure, with 'fast_neural_style' selected. The main content area shows the repository title 'fast-neural-style' with a rocket icon, a description of the repository's purpose, and a list of example images. The description states: 'This repository contains a pytorch implementation of an algorithm for artistic style transfer. The algorithm can be used to mix the content of an image with the style of another image. For example, here is a photograph of a door arch rendered in the style of a stained glass painting.' It also mentions the model uses 'Perceptual Losses for Real-Time Style Transfer and Super-Resolution' and 'Instance Normalization', and provides a link to download saved models. Below the text are three images: a stained glass window with a woman's profile, a photograph of a dark door arch, and the same door arch rendered in a colorful stained glass style.

<https://arxiv.org/abs/1603.08155>