Visualization of convolutional networks and neural style transfer

Javier Zazo

April 4, 2018

Abstract

In this section we will present some advances in the visualization and understanding of convolutional networks. We will discuss how the different activation elements in a network correspond to specific image patches of an input image. We will also discuss about how textures in images are coded in a trained network. And finally, we will see how using these two visualization ideas result in the creation of artistic images that combine content and style in a very appealing form.

1 Visualizing convolutional networks

When studying neural networks we have little insight about what the network is actually learning and the internal operations. Understanding how neural networks work through visualization, can help us understand how the input stimuli excites the individual feature maps. It can allow us to observe the evolution of features and diagnose potential problems during training. It can also help us make more substantiated designs, rather than simply building models through trial and error. All in all, improve general performance if we can address all of these matters.

1.1 Architecture

There has been several advances on how to visualize deep neural networks, and how to understand what they are actually learning. Here, we will focus on one of the first approaches presented in [1].

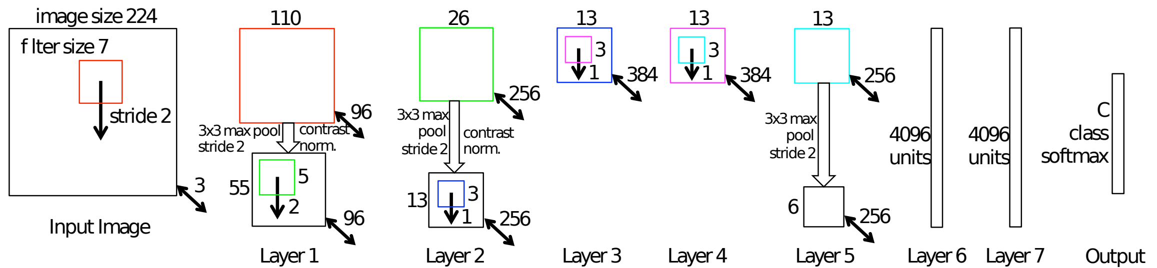


Figure 1: Convolutional network for image visualization [1], inspired by [2].

The authors follow the similar architecture as AlexNet, a trained network discussed in [2]. They train a network on the ImageNet 2012 training database and the input are images of size $256 \times 256 \times 3$. The network is depicted in Figure 1. In summary, it uses convolutional layers, max-pooling and has fully connected layers in the last steps. The size of the feature set decreases in depth, while the number of channels grows. The network is trained for classification of 1000 elements.

For visualization, the authors employ a *deconvolutional network* (deconvnet) [3], where the objective is to project the hidden feature maps into the original input space. The deconvnet has a structure depicted in Figure 2. However, the name "deconvolutional" network may be unfortunate, since the network does not perform any deconvolutions [4]. We will return to this matter later.

On the right side of Figure 2 a normal convolutional layer is depicted. It is formed of a convolution operation, a rectified linear activation, and max-pooling. On the left side, the "deconvolutional layer". We will refer to it as "transposed convolutional layer". It is formed by a convolutional layer with transposed filters (flipped horizontally and vertically), rectified activation, and unpooling layer.

- **Unpooling:** The max-pooling operation is non-invertible, however we can follow the feature map that was used in the convnet layer by recording the locations of the maxima at each max-pooling operation (in *switch* variables). In the deconvnet, the unpooling operation places the reconstructed features from the feature mapping into the recorded locations. Figure 3 depicts this procedure.

- **Rectification:** To obtain a valid reconstructed signal at each layer (which should be positive), the signals go through a ReLu operation.

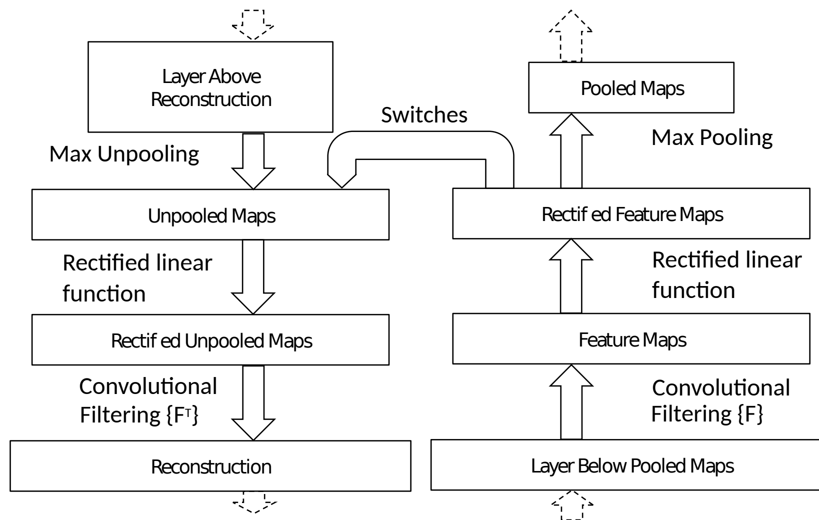


Figure 2: Deconvnet and convnet layer structure [1].

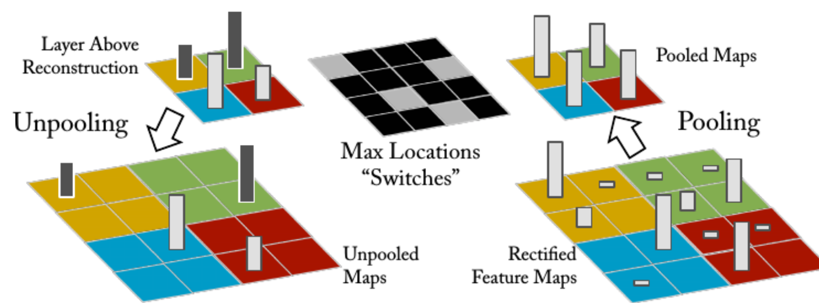


Figure 3: Unpooling operation in a deconvnet.

- **Filtering:** The deconvnet uses a transposed convolution of the learned filters from the convnet. In practice, the filters have to be flipped horizontally and vertically, but care has to be taken if padding or stride greater than one was used, see [5] for details. Neural network frameworks such as tensorflow and others implement the transposed convolution efficiently, such as `tf.conv2d_transposed` in tensorflow.

The purpose of the transposed convolution is to project the feature maps computed by the convnet back to input space. The "transpose" name comes from the analogy that backpropagation in multilayer perceptrons (MLPs) uses a transposed weight matrix $W^{[l]}$ corresponding to the gradient of layer

l. Similarly, the transposed convolution corresponds to the backpropagation gradient computation of convolutional networks.

Because of this scheme, the visualization method proposed in [1] computes gradients from hidden layers and projects them to input space through back-propagation. This interpretation was not commented in the original paper, but further explained in [6], which generalizes this visualization procedure to MLPs.

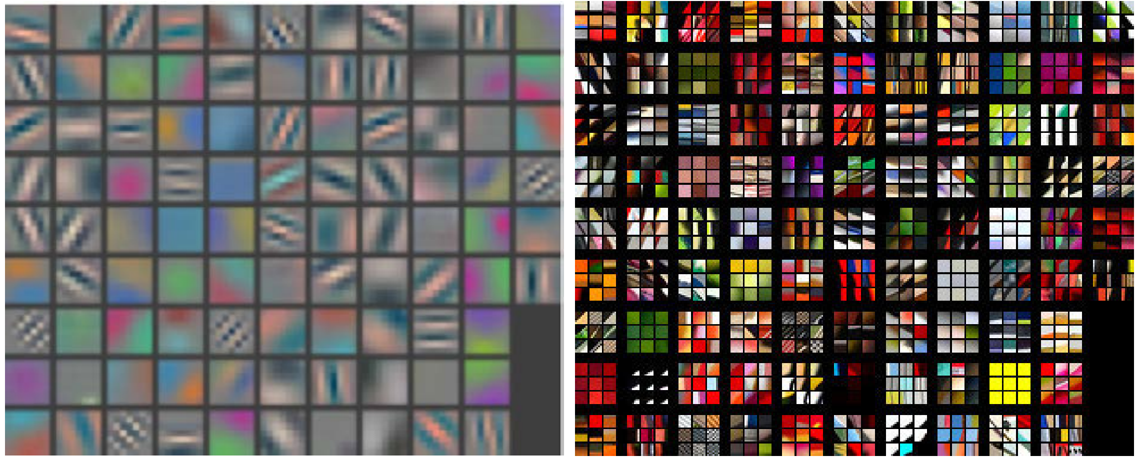
1.2 Feature visualization

The procedure of [1] to visualize features is as follows. To visualize the features that activate a specific neuron[1] in any layer, the authors evaluate the whole validation database on the trained network. Then, they record the nine highest activation values of each neuron's output. And finally, they use the deconvnet to project the recorded 9 outputs into input space for every neuron. In the paper they provide a few of these outputs.

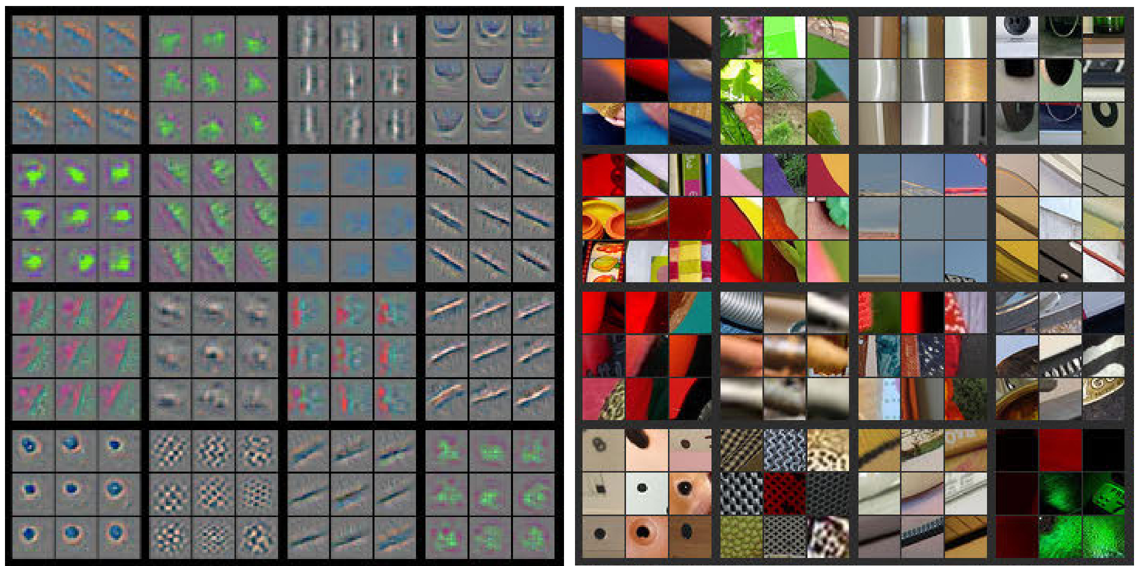
When using the deconvnet to project the layer's output to input space, all other activation units in the given layer are set to zero. This operation ensures that we are only observing the gradient of a single neuron. Note also, that the *switch variables* used to propagate the input signal are used in the unpooling layers, and must be recorded.

Figure 4 show the reconstruction of input images through the deconvnet, and their corresponding image patches. Figure 4a shows how the network learns simple features such as edges and lines. Figure 4c shows more complicated patterns, such as some backgrounds and more involved compositions. And finally, Figure 4e is already recognizing complicated patterns, faces, body parts, wheels and backgrounds as well. All in all, the authors obtained quite direct correspondence from a reconstructed activation to an actual image patch. Figure 5 show more complicated patterns, and we can discern, eyes, faces, legs, bike wheels, bird peaks, flowers, etc. It is quite surprising how a single neuron is excited by these kind of patterns. Note however, that sometimes it is not straightforward to delimit a topic within a neuron, such as birds, dogs, or wheels; this is possible because the same neuron may activated by different pathways in lower layers. And finally, sometimes the neuron is not activated by a main subject within an image, but by the background.

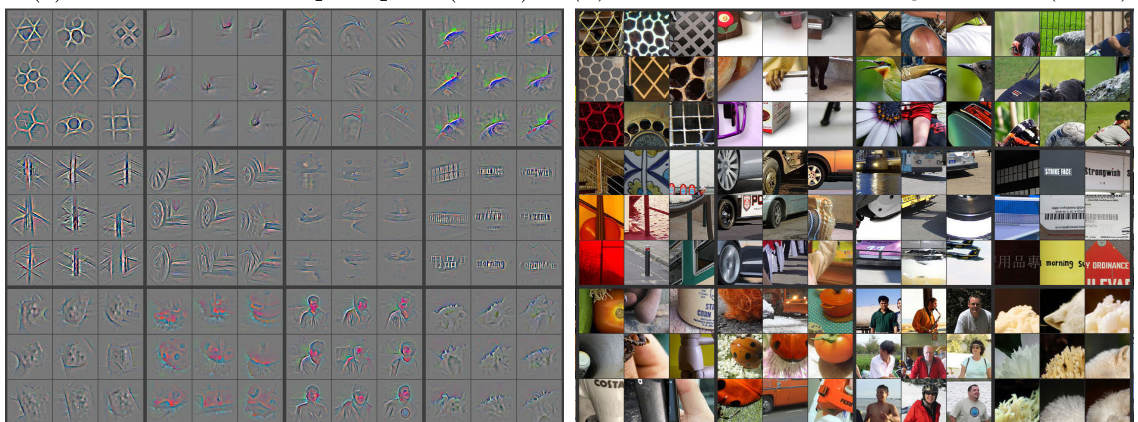
[1] In convolutional networks, there is an equivalence between neurons, channels and numbers of filters on every layer. Therefore, I will use these terms interchangeably.



(a) Visualization on input space ($l = 1$). (b) Correspondence to image patches ($l = 1$).

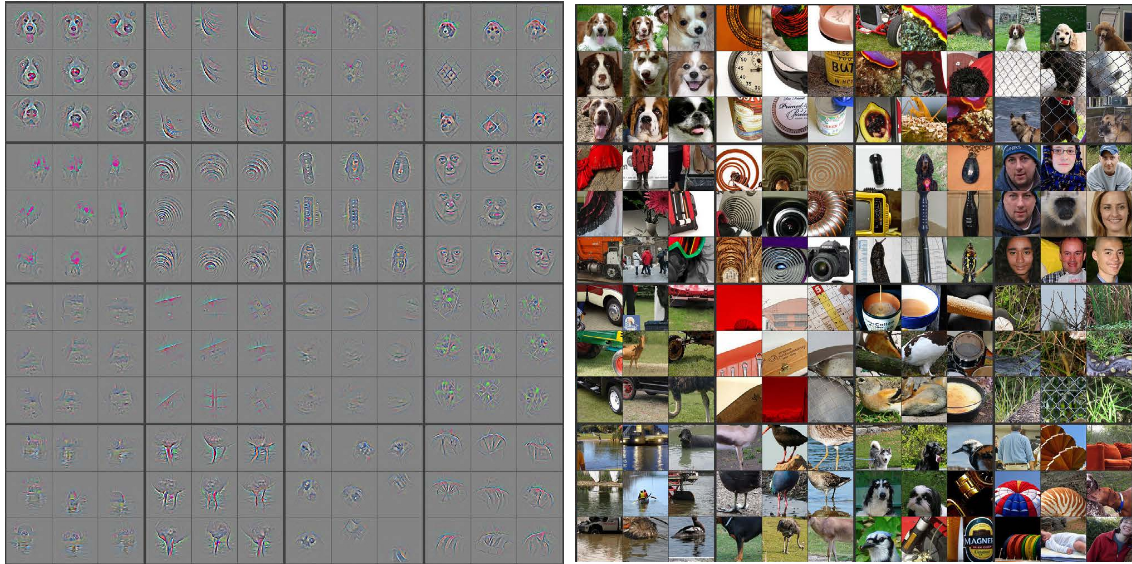


(c) Visualization on input space ($l = 2$). (d) Correspondence to image patches ($l = 2$).

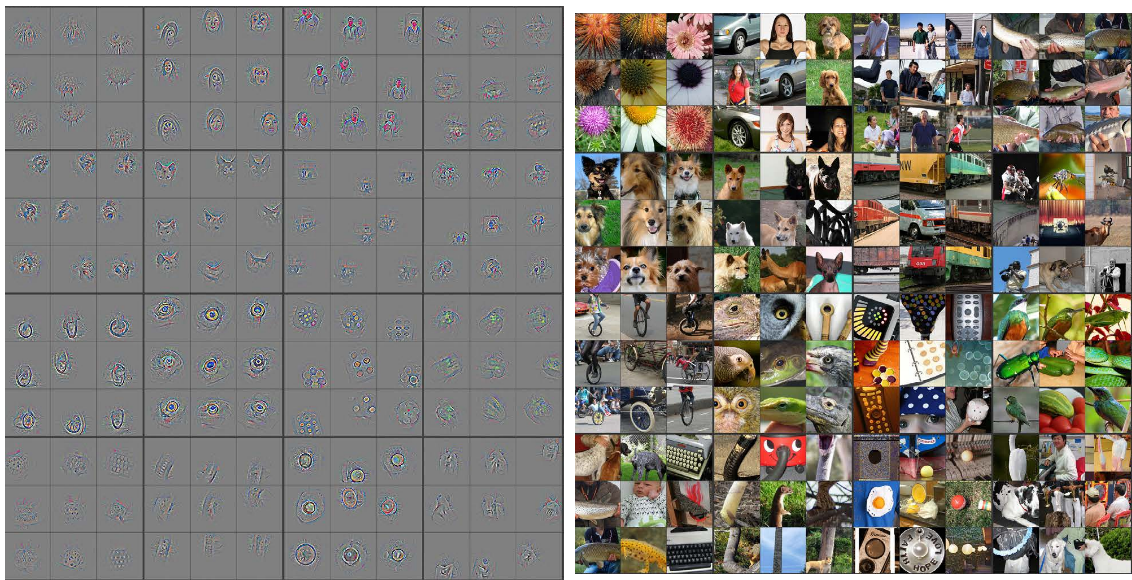


(e) Visualization on input space ($l = 3$). (f) Correspondence to image patches ($l = 3$).

Figure 4: Visualization of neurons from layers 1 to 3.



(a) Visualization on input space ($l = 4$). (b) Correspondence to image patches ($l = 4$).



(c) Visualization on input space ($l = 5$). (d) Correspondence to image patches ($l = 5$).

Figure 5: Visualization of neurons from layers 4 to 5.

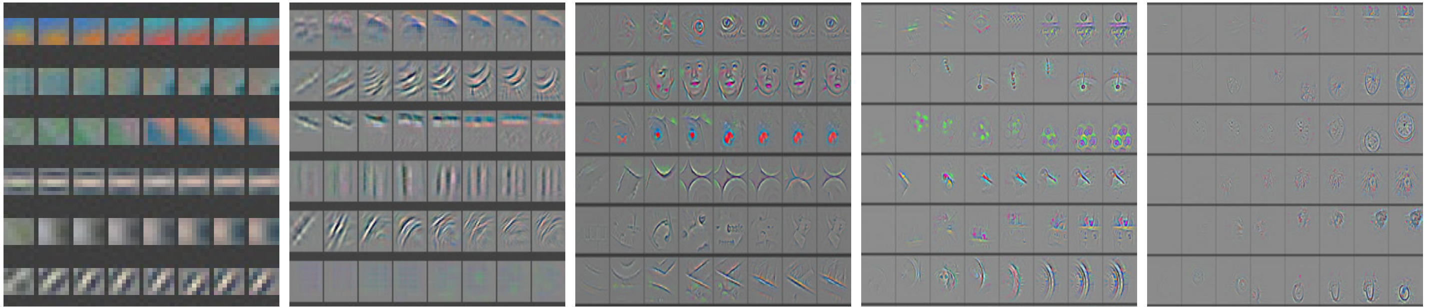


Figure 6: Feature evolution during training for layers from 1 to 5.

1.3 Feature evolution during training

Figure 6 shows the evolution of features during training for 1, 2, 5, 10, 20, 30, 40 and 64 epochs. The picture shows the strongest activation response for some random neurons at all 5 layers, and projected to input space using the deconvnet. As before, when an activation is projected back to input space, all other neuron responses are set to null.

What we can observe in Figure 6 is that low layers converge soon after a few single passes. However, fifth layer does not converge until a very large number of epochs. This visualization indicates the need to verify that all layers have fully converged, and that different layers present different convergence speeds. Furthermore, lower layers may change their feature correspondence after upper layers converge. This is indicated by hard changes between epochs.

1.4 Architecture comparison

Finally, another useful use for visualizing features is to check if different architectures respond similarly or more strongly to the same inputs. For instance, Figure 7 shows the comparison of a modified Alexnet architecture and the original Alexnet network. The architecture used for the picture on the left used filters of size 7×7 instead of 11×11, and reduced the stride from 4 to 2. The visualization provides evidence that there are many less dead units on the modified network than on the original one.

In the case of comparing the second layers, as in Figure 8, the modified network has more defined features, whereas Alexnet has more aliasing effects. This however, is more difficult to observe. Overall, the comparison

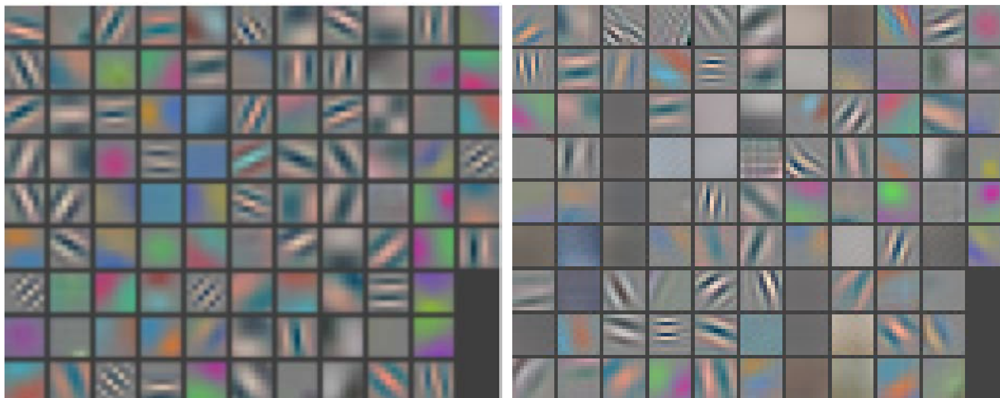


Figure 7: Comparison of different architectures on the first layer.. Left: a modified Alexnet architecture that uses smaller stride and filters. Right: Alexnet original architecture.

provides a way to visually understand why some architectures present better performance metrics than others.

This concludes our exposition about visualization of neural networks. All in all, we have seen how different filters correspond to complex image patterns. We were able to visualize gradients projected to input space of specific activations, and verify how these stimulus correspond to an specific image patch. In the next sections we will talk about image reconstruction and texture synthesis.

2 Image reconstruction

We consider the reconstruction of an image assuming we have the latent features (the encoding) of the image in the neural network. The presented material is based on reference [7]. The authors show, that layers within the network retain an accurate photographical representation about the image, retaining different degrees of geometric and photometric invariance.

We describe the methodology. Assume $a^{[l]}$ corresponds to the latent representation of layer l for some input image $\mathbf{x}$, given the mapping of the neural network $\Phi^{[l]}(\mathbf{x}) = a^{[l](C)}$. We write C to refer to the content image $\mathbf{x}$, and use G to refer to the generated image $\hat{\mathbf{x}}$. The objective is to recover a visual representation of the encoding in the input space (a picture). In order to do

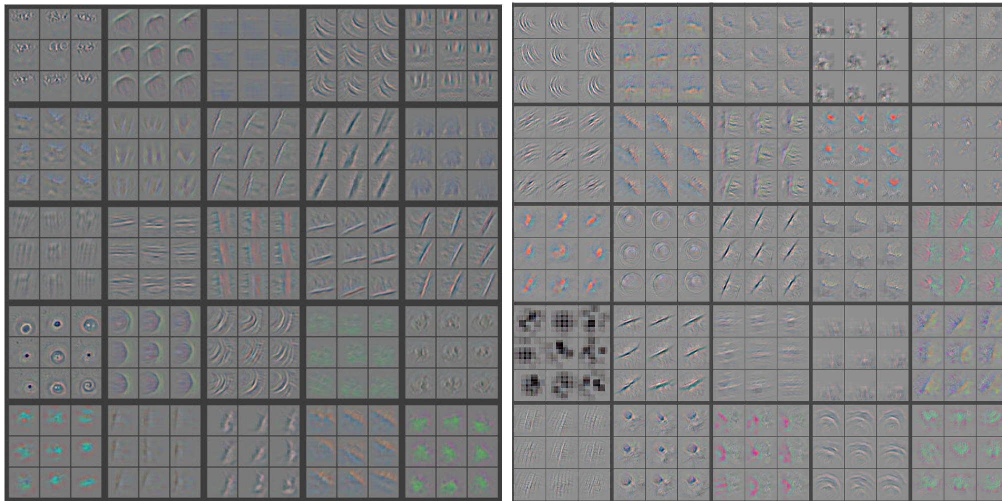


Figure 8: Comparison of different architectures on the second layer. Left: a modified Alexnet architecture that uses smaller stride and filters. Right: Alexnet original architecture.

so, [7] proposes to minimize the following cost function

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{y}} J^{[l](C)}(\mathbf{x}, \mathbf{y}) + \lambda R(\mathbf{y}), \tag{1}$$

where $J_C^{[l]}(\cdot)$ is a loss function for layer l and $R(\cdot)$ a regularizer. An Euclidean loss function gives good results:

$$J_C^{[l]}(\mathbf{x}, \mathbf{y}) = \left\|\Phi^{[l]}(\mathbf{y}) - \Phi^{[l]}(\mathbf{x})\right\|_{\mathcal{F}}^2 = \left\|a^{[l](G)} - a^{[l](C)}\right\|_{\mathcal{F}}^2. \tag{2}$$

As regularizer, the authors recommend to use a combination of α-norm regularizer, and a *total variation* regularizer. The α-norm consists on $R_\alpha(\mathbf{y}) = \lambda_\alpha \|\mathbf{y}\|_\alpha^\alpha$, and the *total variation* encourages images to develop piece-wise constant patches:

$$R_{V_\beta}(\mathbf{y}) = \lambda_{V_\beta} \sum_{i,j,k} \left(\left(a_{i,j+1,k}^{[l](G)} - a_{i,j,k}^{[l](C)}\right)^2 + \left(a_{i+1,j,k}^{[l](G)} - a_{i,j,k}^{[l](C)}\right)^2 \right)^{\beta/2}. \tag{3}$$

The recommended parameters to choose for the regularization terms are specified in [7]. We skip such discussion.

To reconstruct an image, a random noise initialization is computed and assigned to $\mathbf{y}$. Then, gradients for the cost function (2) are determined and

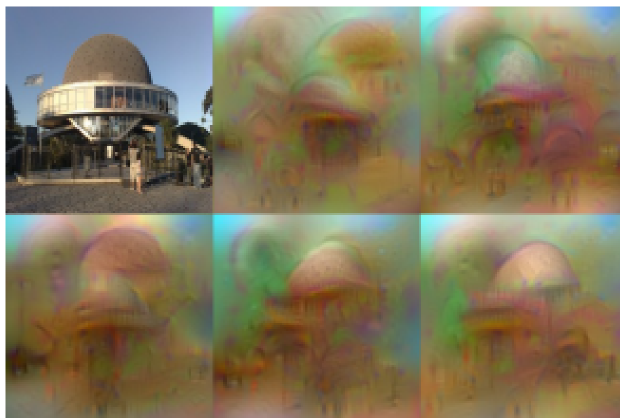


Figure 9: Five possible reconstructions of a reference image extracted from a 1000-dimensional encoding and the penultimate layer of Alexnet [2].

backpropagated to input space. The generated image G is updated with a gradient step, and the process is repeated. Figure 9 presents five possible reconstructions for a reference image. All these reconstructions possess similar activation values at layer l. The effect of spikes is controlled with the regularization parameters.

3 Texture synthesis using convnets

We present results of texture synthesis based on a generative model described in [8]. The method is able to generate high perceptual quality images that imitate a given texture. It uses a trained convolutional network (such as VGG) for object classification, and uses the correlation of features among layers as the generative process to obtain new textures. The authors show that the texture representations are captured best by computing these correlations across layers, eventually making the texture information increasingly explicit.

The output a convolutional layer l is a block with width $n_W^{[l]}$, height $n_H^{[l]}$, and number of channels $n_c^{[l]}$. This is depicted in Figure 10. What we are interested to compute, is the correlation among channels within a layer for some input image (the texture to imitate). In other words, we have seen how different filters are activated by different image patches, and what we want to do now is to determine if these activations are correlated and occur together,

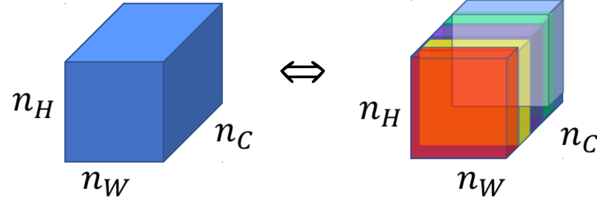


Figure 10: Output sizes of a convolutional layer.

or if they are uncorrelated and do not necessarily occur together. In order to reproduce a texture, we compute the correlation that exists between all the different filters within a layer l, and later compose a different figure that imitates these correlations.

Let's denote the output of a given filter k at layer l with $a^{[l]}_{ijk}$. Indexes i and j refer to the spatial latent features, and k to a specific channel. The cross-correlation between this output and a different channel k' is given by:

$$G^{[l]}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l]}_{ijk} a^{[l]}_{ijk'}. \tag{4}$$

By computing this elements for all k and $k' \in \{1, \ldots, n_C^{[l]}\}$, we obtain the Gram matrix $G^{[l]}$ of size $n_C^{[l]} \times n_C^{[l]}$. Expression (4) can be vectorized as follows. Assume $a^{[l]}_{::k}$ is a vector of size $n_H^{[l]} n_W^{[l]} \times 1$, and $(A^{[l]})^T = (a^{[l]}_{::1}, \ldots, a^{[l]}_{::n_C^{[l]}})$. Then,

$$G^{[l]} = A^{[l]}(A^{[l]})^T \tag{5}$$

is an efficient calculation for the channel correlation matrix.

Remark: Matrix $G^{[l]}$ is a Gram matrix. Strictly speaking, it is not a correlation matrix as it is normally defined in statistics, because when we compute it we are not subtracting the mean of the elements. However, the authors of [8] referred to it as correlation matrix, and it implicitly indicates correlation.

In order to generate a new texture, we will create a new image that has similar correlation as the one we want to reproduce. We introduce a cost function that relates the Gram matrices of the texture of interest, and the new image to generate. We denote with $G^{[l](S)}$ the Gram matrix of the *style* image we want to imitate, and $G^{[l](G)}$ the corresponding one of the newly

generated image. We build these correlations for every layer l, and define the following cost function:

$$J_S^{[l]}(G^{[l](S)}, G^{[l](G)}) = \frac{1}{4(n_W^{[l]} n_H^{[l]})^2} \left\| G^{[l](S)} - G^{[l](G)} \right\|_{\mathcal{F}}^2, \tag{6}$$

where $\|G\|_{\mathcal{F}} = \sqrt{\sum_{ij}(g_{ij})^2}$ corresponds to the Frobenius norm.

Additionally, we combine all of the layer losses into a global cost function, for given weights $\lambda_1, \ldots, \lambda_L$:

$$J_S(\mathbf{x}, \mathbf{y}) = \sum_{l=0}^{L} \lambda_l J_S^{[l]}(G^{[l](S)}, G^{[l](G)}). \tag{7}$$

For notational simplicity, we use $\mathbf{x}$ and $\mathbf{y}$ as the original texture to imitate and the generated image, respectively, for the cost function.

Finally, the synthesis procedure is described in Figure 11. The algorithm is initialized with a random noise picture, and then modified iteratively until convergence. Latent features are computed on the *style* image, and the *random noise* image. We compute the correlations, and determine the Frobenius distances between the Gram matrices. Then, we determine the gradient on the cost function (7) and project it back to input space using backpropagation. Finally, we mofify the *generated* image accordingly using some optimization technique, such as gradient descent, or L-BFGS. We repeat the process iteratively until convergence. Tensorflow and other frameworks have automatic differentiation to compute the gradient descent algorithm and backprop, so in essence, the algorithm is not difficult to implement.

Examples of generated textures are given in Figure 12. The different rows correspond to different cost functions that incorporate more or less layer costs. The last column shows the result for an image without texture.

4 Neural style transfer

Neural style transfer is the artistic generation of high perceptual quality images that combine the style or texture of some input image, and the elements or content from a different one. We describe the idea presented in [9]. Figure 13 gives an example of the results achieved in the original publication. In the following, we refer to the *style* image with S, the *content* image with C, and the *generated* image with G.

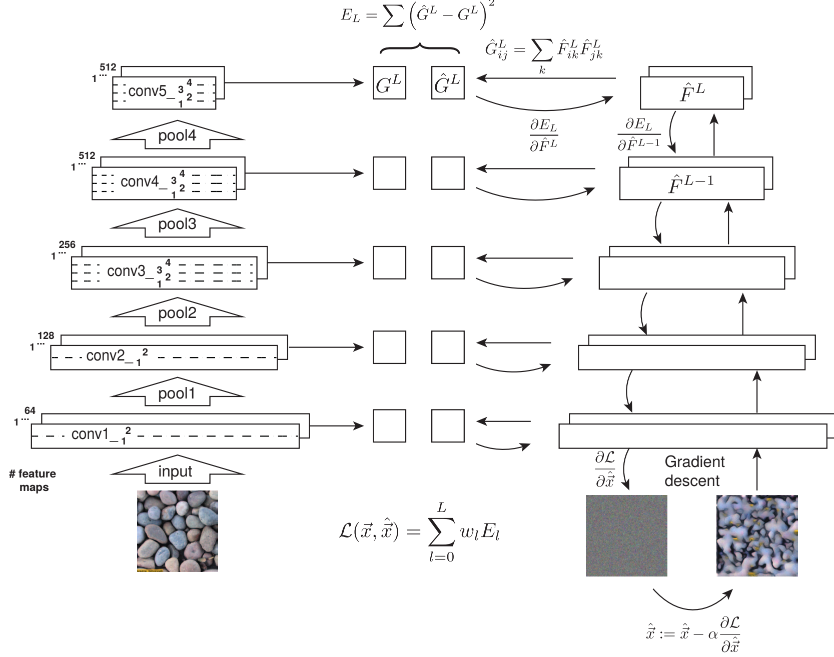


Figure 11: Graphical generation of a texture image.

Every layer in a convnet defines a non-linear filter bank whose complexity increases with the position of the layer. The methodology to combine images consists on optimizing a cost function that uses the activations of the network of the *content* image, and the style correlations of the *style* image. Figure 14 shows how the different activation values in the convnet are used after propagation to reconstruct the content, or the textures, according to Sections 2 and 3, respectively.

After the derivation of content reconstruction and style resemblance, the objective function for neural style transfer simply becomes the combination of both cost functions:

$$J_{\text{total}}(\mathbf{x}, \mathbf{y}) = \alpha J_C^{[l]}(\mathbf{x}, \mathbf{y}) + \beta J_S(\mathbf{x}, \mathbf{y}) \tag{8}$$

where $J_C^{[l]}(\mathbf{x}, \mathbf{y})$ corresponds to (2), and $J_S(\mathbf{x}, \mathbf{y})$ to (7). Note that (2) requires the choice of some layer l to compute the cost. In principle, middle layers are recommended (not too shallow, not too deep) for best results. Furthermore, regularization in (2) can be set to null, and obtain good results.

A few final remarks. Equation (8) is minimized using gradient descent or any other method typical of neural networks combined with backpropagation.

Figure 12: Texture generated examples.

Figure 13: Images that combine the contents and styles of different sources. Picture A credit to Andreas Praefcke. Artwork shown in the bottom left corner of each panel. B *The Shipwreck of the Minotaur* by J.M.W. Turner, 1805. C *The Starry Night* by Vincent van Gogh 1889. D *Der Schrei* by Edvard Munch, 1893. E *Femme nue assise* by Pablo Picasso, 1910. F *Composition VII* by Wassily Kandinsky, 1913.

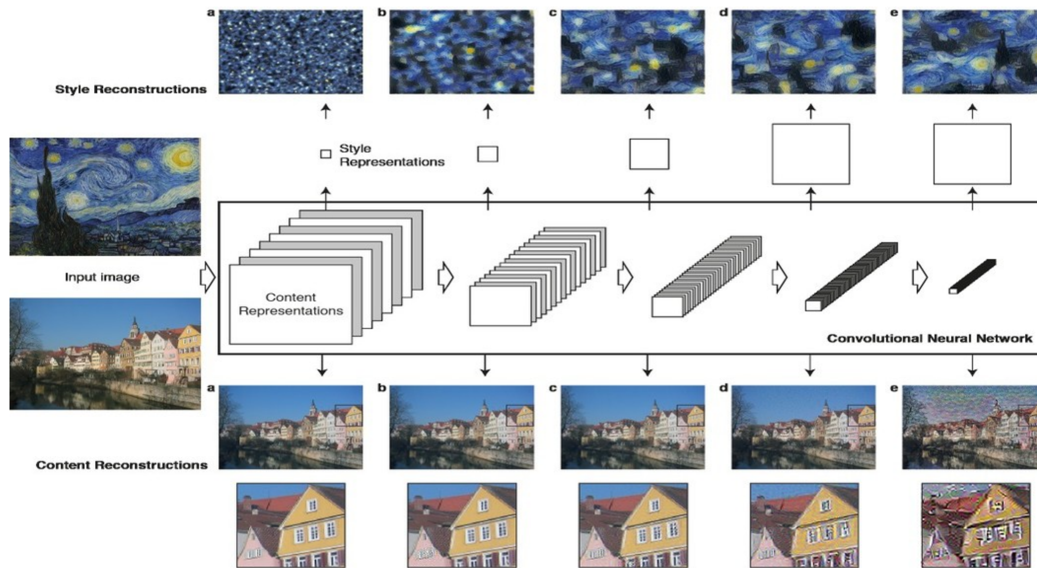


Figure 14: Convolutional activations, content and texture reconstructions.

The input **y** is initialized with random noise. After convergence, we assign the output to $\hat{x}$, obtaining the *generated* image. For image synthesis, we use a discriminating trained convolutional network for image classification, such as VGG. Finally, the authors from [9] found that replacing the max-pooling layers with average pooling improves the gradient flow, and this produces more appealing pictures.

References

[1] Matthew D. Zeiler and Rob Fergus, "Visualizing and understanding convolutional networks," in *Computer Vision*. 2014, pp. 818–833, Springer, doi:10.1007/978-3-319-10590-1_53, arXiv:1311.2901.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105, doi:10.1145/3065386.

[3] Matthew D Zeiler, Graham W Taylor, and Rob Fergus, "Adaptive deconvolutional networks for mid and high level feature learning," in *IEEE International Conference on Computer Vision (ICCV)*, 2011, pp. 2018–2025, doi:10.1109/ICCV.2011.6126474.

[4] "What are deconvolutional layers?," `https://datascience.stackexchange.com/a/12110`.

[5] Vincent Dumoulin and Francesco Visin, "A guide to convolution arithmetic for deep learning," Mar. 2016, arXiv:1603.07285.

[6] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," Dec. 2013, arXiv:1312.6034.

[7] Aravindh Mahendran and Andrea Vedaldi, "Understanding deep image representations by inverting them," Nov. 2014, arXiv:1412.0035.

[8] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge, "Texture synthesis using convolutional neural networks," Nov. 2015, arXiv:1505.07376.

[9] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge, "A neural algorithm of artistic style," Aug. 2015, arXiv:1508.06576.

Acknowledgments

This manuscript was composed by Javier Zazo, and the material was obtained from the cited references. Figures 1 to 8 were obtained from [1]. Figure 9 from [7]. Figure 10 from deeplearning.ai course. Figures 11 and 12 from [8]. Figures 13 and 14 from [9].