Dissecting Neural Style Transfer

Xi Du

Australian National University, Australia
u6559090@anu.edu.au

Abstract. [?]

Keywords: Neural network, Deep learning, Small data, Interpretability
.

1 Introduction

2 Literature Review

3 Method

3.1 Deep Neural Network

For the purpose of implementing and even extending neural style transfer algorithms, it is not necessary to understand how a neural network is trained, because we can use pretrained model for example vgg19 [?].

It is necessary to point out that a deep neural network that neural style transfer is concerend about is just a sequence of functions $F_1, F_2, F_3, ...$ whose inputs and outputs are all multi-dimensional arrays, as illustrated in Figure 1. Because

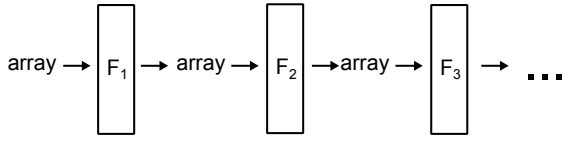

Fig. 1. A deep neural network

the arrays have usually more than two numbers of dimensions they are not really matrices. The arrays are called tensors (as in TensorFlow or torch.Tensor in PyTorch) but actual tensor algebra in a mathematical sense is rarely relevant. The functions need to be differentiable, which is handled automatically by modern deep learning frameworks. Other than that we can treat the functions as blackboxes because we are not concerned of training the model. A "layer" in a deep neural network refers to a few consecutive functions together. It is not particularly necessary to consider "layers" in this work. Though it is important to note that in our illustrations there are functions instead of layers.

3.2 vgg19 model

For the vgg19 model that most neural style transfer implementations available are based on, The inputs and outpus of each function are all arrays with 4 dimensions. The word "dimension" here may mean something slightly different from what "dimension" means in for example "3-dimension vector". Some people call the number of dimensions "rank" which would then raise another confusion with the rank of matrices. We give an illustration for the vgg19 model in Figure 2

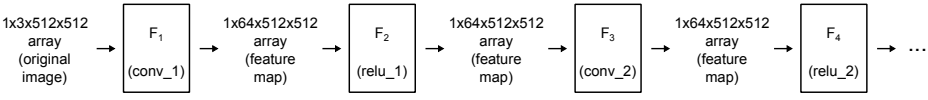

Fig. 2. The VGG19 model with an 512×512 RGB image as its input

The 1st dimension is the "batch" dimension, which means that you can put several images through the series of functions. In neural style transfer implementations it is usually just one image each time. So the 1st dimension is always 1, even between the F functions, throughout this work.

The 2nd dimension is the "feature" dimension. For a raw RGB image, it is 3. Intermediate arrays between, for example, F_3 and F_4, usually have a size much larger than 3, such as 64 or 128.

The 3nd and 4th dimensions are just spatial locations. For a 512×512 image they are 512 and 512. For intermediate arrays, these 2 dimensions are sometimes scaled down to half or even $\frac{1}{4}$ of the sizes of the image.

The functions $F_1, F_2, F_3, ...$ now have meaningful names such as relu_1, conv_2 now. [[[[Not actually necessary to know but insert your elaborations here]]]]

The arrays after the initial image are called *feature maps*, because their value means "how much a feature x exists at position y and x". Take a $1 \times 64 \times 512 \times 512$ feature map for example. There are 64 types of features. Its [1, 17, 111, 222] element, would then refer to the extent that the 17th type feature exists at the position of the 222th column of the 111th row. This "extent" could also be negative.

3.3 Vanilla neural style transfer

The vanillay neural style transfer algorithm [**?**] is structured as iteratively solving an optimisation problem. The argument to optimise is the image as a multidimensional array, of size $1 \times 3 \times 512 \times 512$ for example. The interative solver can be many, but in our case it is L-BFGS [[[[Maybe elaborate on L-BFGS here]]]]. The initial value of the argument could be either white noise or the content image, but the latter appeared to make the optimisation much easier.

The key issue here is still how to structure the optimisation target. The value to minimise is a linear combination of a "style loss" and "content loss". Although

these appeared to be two weights, actually only their ratio mattered. We simply fixed the weight of the content loss to 1 and leave the weight of the style loss as a adjustable hyperparameter k.

$$\underset{\text{image}}{\operatorname{argmin}}(kL_{\text{style}} + L_{\text{content}}) \tag{1}$$

The content loss L_{content} is the easier one to explain of the two losses. It is the mean square error between the outputs of a certain function in the vgg19 taking the current argument image and the content image as inputs respectively. We say "a certain" function because the output of which function to choose is adjustable. A typical choice is the output of `conv_4` function. See Figure 3.

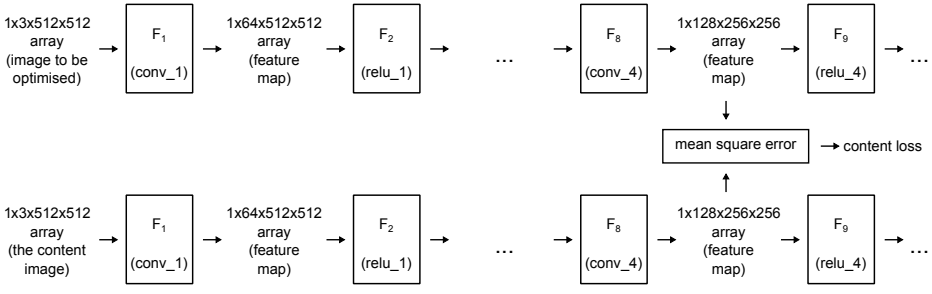

Fig. 3. The content loss

The style loss L_{content} can actually be a sum of several style losses, or sometimes just one. For each style loss, both the current image under optimisation and the style image are passed through the vgg19 model. The two outputs of a certain function in the model are extracted, and converted to Gram matrices. We will elabrate on what Gram matrices are later. For now a Gram matrix is just a square matrix computed from a multidimensional array (i.e. a feature map). For example, the Gram matrix (in the style transfer algorithm) of a $1 \times 64 \times 512 \times 512$ array is a 64×64 matrix. The point is that a Gram matrix describes the distribution of features in the image without the spatial information of where the features are. Gram matrices describe the distribution with correlation between features. The several style losses are computed with the outputs of different functions. Otherwise, each of the several style losses are computed in exactly the same way. Again, the outputs of which functions in the model to derive style losses from is really adjustable. A nice default we used was `conv_1`,`conv_2`,...,`conv_5`. We illustrate the style losses derived from the output of `conv_1` in Figure 4.

3.4 Gram matrix

[[[[Elaborate on gram matrix here]]]]

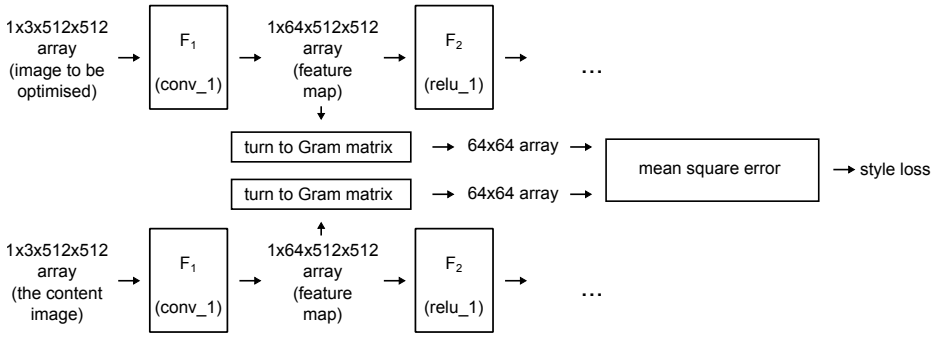

Fig. 4. The style loss derived from the output of conv_1

3.5 Removing the spatial information in other ways

We think that the most brilliant aspect of the neural style transfer algorithm, apart from the general structure, is the use of Gram matrices to compare distributions of features in two images, so that where the features are does not matter. It gives really nice insight on what "artistic style" means.

Now we wonder, could we achieve similar results with other ways to compare distribution of features? One approach would be to consider the *comparison* as a whole to find substitutes. For example, minimising the mean square errors between two Gram matrices could be seen as a special case for minimising a Maximum Mean Discrepancy (MMD), which means other Maximum Mean Discrepacies could be used [**?**].

[[[[Maybe Describe MMD here]]]]

For us, it seemed more natural to consider substitutes for the process itself of transforming a feature map into a Gram matrix. That is, we will consider other mappings that remove the spatial information or where features are in a feature map but retain the distribution of features, like a histogram. Such functions need to be differentiable as well, otherwise they would render the optimisation infeasible.

We will call such functions "*dislocators*" in the remaing part of this work. Unfortunately, the word "dislocate" would give the connotation of moving something to somewhere else. Nevertheless, the word "locator" means something that provides spatial information, so calling somthing that removes spatial information a "dislocator" is still reasonable.

Other "dislocator" functions whose output somehow describe the distribution of features in their input should then work as substitutes for computing Gram matrices. The first few simplistic "dislocator" functions we tried are listed here.

1. Scaled sum of the responses all over the place for each feature. That is, for an $1 \times 64 \times 256 \times 256$ feature map, we compute an 1×64 array where each element is the sum of the 256×256 elements from the input. The whole array is then scaled to one over the total number of elements of the input feature map.

2. Scaled sum of element-wise squares of the responses over the place for each feature. Other than an additional element-wise squaring, it is identical to the previous one.
3. Scaled sum of absolute values of the responses over the place for each feature.

To find more dislocators, let us consider *how to describe a distribution* in general? The obvious answer is then sample statistics or estimators, like (sample) mean, (sample) standard deviation, (sample) skewness, (sample) kurtosis and any moments. Unfortunately, we found that higher moments are too slow to compute for experiments, so we ended up with the following (combinations) of sample statistics.

4 Experiments and Discussion

4.1 Reproducing the original result

4.2 Varying the dislocator function

4.3 Varying the extraction points for losses

We will also show that other factors, such that the outputs of which functions to derive the style losses from are not very essential and somewhat belongs to tuning.

5 Conclusion and Future Work

References

1. Gatys, L., Ecker, A., & Bethge, M. (2016). A neural algorithm of artistic style. Journal of Vision, 16(12), 326. doi:10.1167/16.12.326
2. Li, Y., Wang, N., Liu, J., & Hou, X. (2017). Demystifying Neural Style Transfer. IJCAI.
3. Doshi-Velez, F., & Kim, B. (2017). Towards A Rigorous Science of Interpretable Machine Learning.
4. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., & Lerer, A. (2017). Automatic differentiation in PyTorch.