Style Transformer

Zhihao Zhang

12/10/2020

Introduction

For this project, I will explore one of the deep learning applications in the area of Computer Vision, specifically Image Style Transfer, which transfers the style (mostly art painting) from one image into the target image (mostly landscape photo). The synthesized image contains both the object structure in the target image and the style in the art image. Before deep learning became popular, processing such tasks was difficult due to the lack of image representations that explicitly represent the semantic information. By using Convolutional neural networks, we can derive such feature representations and then synthesizes both the structure and style together. And it generalizes very well to handle different scene images and art style images, which is perfect for feature extraction in this project.

Technical details

Dataset: ImageNet. Since I selected a pre-trained model that is trained on this dataset, there is no need to further train the model due to the purpose of this project
Selected Model: VGG-19 pre-trained Convolutional Neural Network, which is trained on ImageNet to perform object recognition and localization. It was used in the paper of A neural algorithm of artistic style[2]. Since I used VGG to extract the feature maps for both content and style images, the Fully Connected layers are not used in this project.
Training Process:

1. Extract feature maps at specific Conv Layers for both Content image and Style image, and store those features for later use. I used the hook mechanism to capture the feature maps.

2. Create an input image (white noise/Copy of Content), and set auto_grad to be true in order to let the optimizer to update its gradients.

3. Train the input image for 300 iterations. Each iteration updates the input image toward to the minimized total loss.

4. Review the result image, and fine tune the hyperparameters to generate some artistic great-looking styled image.

I created two versions of the style transformer. Version 1 implemented torch hook mechanism to generate feature maps and then compute loss function separately. Version 2 created a new model that contains selected Conv layers and loss layers, computing the loss directly through the new model.
I fully developed Version 1 and intend to discuss the methods in Version 1. The idea of Version 2 comes from Pytorch tutorials on Neural Transfer using Pytorch. I used it as my implementation counterpart and will compare the results from both versions in the Results section.

Version 1 Implmentation:

Preprocessing: First of all, The sizes of content and style images must be the same and need to be converted into torch format as (B,C,H,W), which stands for (Batch, Channel, Height, Width). Hence, Some preprocessing steps are needed to re-format the input image.

Hook mechanism: After preprocessing, I used module.register_forward_hook method to capture the feature maps passed through specific layers of the VGG net. The hook function can only obtain the results after the forward call is executed in the Model. See details.
I used vgg19 pre-trained model and froze all weights by setting requires_grad = False, as Training step is not on the model itself but the input image later. Now, I hooked (5, 12, 22) layers for content image and (5, 12, 22, 32, 35) layers for style image. By passing content image to the VGG net, the content hooks will contain feature values after passing those selected layers, I stored those values for later computing the loss function. And this is the same procedure for style image.
Now, we have prepared all components and need to define two loss functions.

Content loss: The content loss function is defined as sum of element-wise difference from a given layer's content feature maps and input feature maps. The mathmatical formula is as follows:

$$\mathcal{L}_{\text{content}}\left(\vec{p}, \vec{x}, l\right) = \frac{1}{2} \sum_{ij} \left(F_{ij}^{l} - P_{ij}^{l}\right)^2$$

$\vec{p}$ is the original image, $\vec{x}$ is the generated image, l is the layer. P_{ij}^{l} is the activation of the i^{th} filter at position j in layer l for original image. F_{ij}^{l} is the activation of the i^{th} filter at position j in layer l for generated image.
F^l and P^l are dimension of $\mathcal{R}^{N_l \times M_l}$, where N_l is number of distinct filters in layer l, and M_l is the height times the width of the feature map. For example, an intermediate layer has a feature map with size (64, 20, 20) where 64 is Channel, 20 is height and weight. then F^l is dimension of (64, 400)
This is only for one layer, the final content loss is the sum of all layers' losses.

Style loss: Instead of directly computing the feature difference, we need to include the feature correlations of multiple layers to obtain a stationary, multi-scale representation of the input image, which captures its texture information but not the global arrangement[1]. 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:

$$G_{ij}^{l} = \sum_{k} F_{ik}^{l} F_{jk}^{l}$$

For example, an intermediate layer has a featur map with size (64, 20, 20). we flatten it by combining height and weight as we did in content loss and get a (64, 400) dimensional matrix, where 64 is Channel, 400 is vectorised feature map. Lastly, we compute the inner product to get the feature correlation matrix: (64, 64).

Next, we compute the difference between style image Gram matrix and content image Gram matrix for a given layer:

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^{l} - A_{ij}^{l}\right)^2$$

The final step is to add E_l for all selected layers as total style loss:

$$\mathcal{L}_{\text{style}}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l$$

Total loss:

$$\mathcal{L}_{\text{total}}\left(\vec{p}, \vec{a}, \vec{x}\right) = \alpha \mathcal{L}_{\text{content}}\left(\vec{p}, \vec{x}\right) + \beta \mathcal{L}_{\text{style}}\left(\vec{a}, \vec{x}\right)$$

$\vec{p}$ is the original image, $\vec{a}$ is the style image, $\vec{x}$ is the generated image.
α is the weight parameter for content loss, saying how much we care about the content.
β is the weight parameter for style loss, saying how much we care about the style.

Training details: I choose LBFGS optimizer with default learning rate = 1 as suggested in the paper[1]; however, other optimizer such as Adam can also be utilized. One thing about LBFGS is that it requires a closure when calling optimizer.step(). A closure is a callable function that performs the training step once and reevaluates the model and return the loss; then the optimizer updates the gradient as usual.
Mean Square Error is my selected loss function and 300 is total number of steps to train. The most work of hyperparameter selection is to choose α (content weight) and β (style weight).
$\alpha = 100$ and $\beta = 1e6$: test on using the copy of content image as input image
$\alpha = 1e7$ and $\beta = 10$: test on using the white noise image as input image.

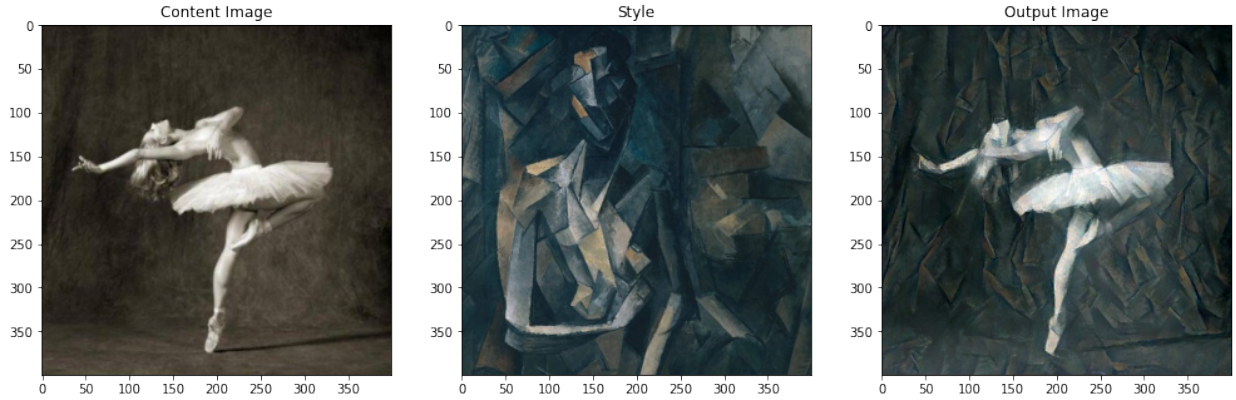
Results

Under this section, the illustration of results are from both Version 1 Implmentation (mine) and Version 2 Implmentation (Pytorch tutorial). At the end, I show a couple of different results from different content and style combinations.

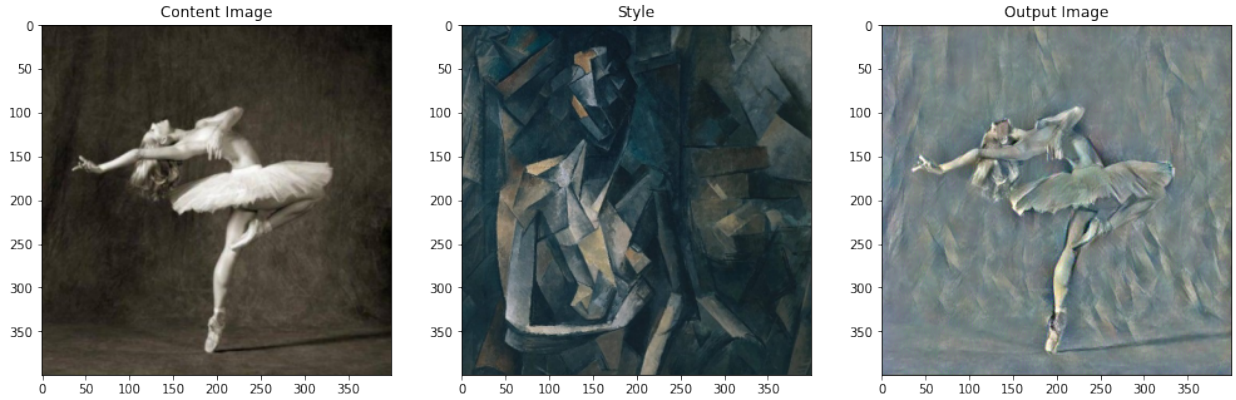
Both versions are using the LBFGS optimizer with default learning rate = 1 and MSEloss.

Version 1:

1. use copy of content image, $\alpha = 100$ and $\beta = 1e6$



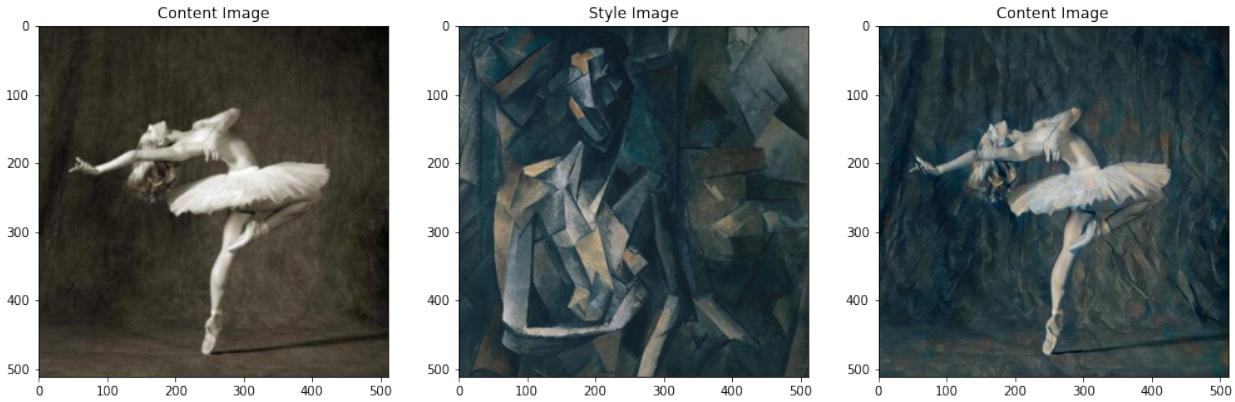
2. use white noise image, $\alpha = 1e7$ and $\beta = 10$



Generally speaking, using content image as input did pretty well on the result; however, using white noise image could not capture the style texture very well. The obvious reason is the hyperparameter choice. After done several experiments, I realized that α value need to be large in order to capture more about the structure of content, which enforces us to focus more on content instead of style. This is the best achievable result so far. Thus, if using white noise image, we need to pay more attention on the content rather than style.

Version 2:

1. use copy of content image, $\alpha = 1$ and $\beta = 1e6$



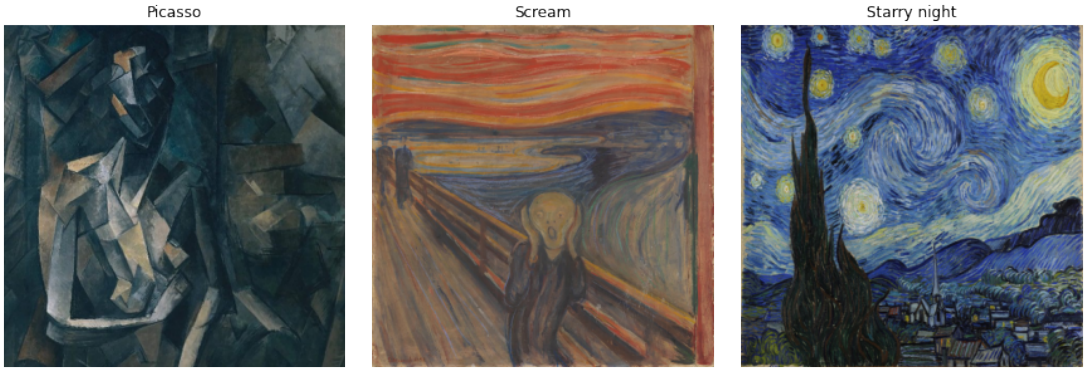
2. use white noise image, $\alpha = 10$ and $\beta = 1e6$

Clearly, we can see the Version 2 did better than Version 1. When using the content image as input image, Both versions created very well-styled results, but Version 2 kept more texture colors than Version 1. When using the white noise image as input image, Version 2 kept much more texture information than the counterpart in Version 1. Notice the hyperparameter choice for Version 2 with white noise image. It posed $\alpha = 10$ and can capture enough content structure and thus enforced more weight on style loss, which Version 1 cannot acheive the same effect with the same α and β.

3. More examples:

Content:

Style:



Version 1 style:

Version 2 style:

Observation:
Version 2 implemnetation is generally better than Version 1's. Version 2 could obtain as many texture colors as possible and produce vivid styles. However, Version 1 seems to be dim dark overall and produce seemingly unnatural balance over content and style. But some particular styles are well-crafted in Version 1 implementation such as the horse with picasso style: the horse itself is segemented as the same style as the picasso's painting!

Conclusion

For this project, I intend to build a Deep Learning model that combines the style (texture features) and content (locational features) together to generate a new image with balanced mixture of both. The most important component of this project is to define the loss function and optimize the loss. At the end, I generated several style-crafted images and demonstrated that my implmentation (Version 1) could reproduce the result shown in the

paper[1]. However, it still has some issues with respect to the image fidelity and visual effect. Then I developed Version 2 under Pytorch tutorials on style transfer, which used a different approach to build the system, and it generated advanced well-looking style-crafted images.

Source Code: https://github.com/zzyrd/Style-Transformer

Reference

1. Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. "Image style transfer using convolutional neural networks." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

2. Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. "A neural algorithm of artistic style." arXiv preprint arXiv:1508.06576 (2015).

3. Ghiasi, Golnaz, et al. "Exploring the structure of a real-time, arbitrary neural artistic stylization network." arXiv preprint arXiv:1705.06830 (2017).

4. Ruder, Manuel, Alexey Dosovitskiy, and Thomas Brox. "Artistic style transfer for videos." German Conference on Pattern Recognition. Springer, Cham, 2016.