REPORT: ARTISTIC STYLE TRANSFER

Tuan Tran – A20357888
Tung Nguyen – A20350345

I. Introduction:

With the recent advancement in Deep Learning, machines have achieved significant, near-human performances on a lot of key areas of visual related problems such as object recognition. However, a visual perception area on algorithmically investigating how humans create and perceive artistic imagery has only emerged in recent years. More specifically, in fine arts like painting, even though humans are able to create unique visual experiences through composing a complex interplay between the content and style of an image, there haven't been any algorithms which demonstrate the same capabilities. The term "style transfer" has been coined to describe such problem of training machines to "roughly" mimic human's ability to interplay different image styles to create unique and complex visual experiences. To put it simply, style transfer is the problem of recomposing images in the style of other images; that is, given an original image, and a second image with specific style, we would like to create a new image with the style of the second image applied to the content of the original image. Even though it does not fully explain how humans create unique art pieces, style transfer is definitely the right logical step towards solving the more complex problem of understanding how humans create and perceive arts.

We'll use the algorithm introduced by Gatys et al. as the basis of this project. With a pre-trained VGG model and a clever loss setup, Gatys et al. algorithm is able to achieve high-quality results and also provides insights into image representations learned by Convolutional Neural Networks and empirically demonstrate CNNs' potential for high level image synthesis and manipulation [1]. We'll also implement an extension of this method proposed by Johnson et al. in order to speed up styling process. This extension allows for fast real time style transfer.

II. Base Model Approach:

In order to capture image representations, Gatys et al. proposed a training algorithm that uses the feature space of the pre-trained and very competent CNN called VGG network. Even though Gatys et al. used VGG19, we decided to use VGG16 since we found that VGG16's generated images are more coherent and not as noisy as those generated by

VGG19. Thus, we'll use VGG16 for both our base model results (proposed by Gatys et al.) and for the extension model proposed by Johnson et al.

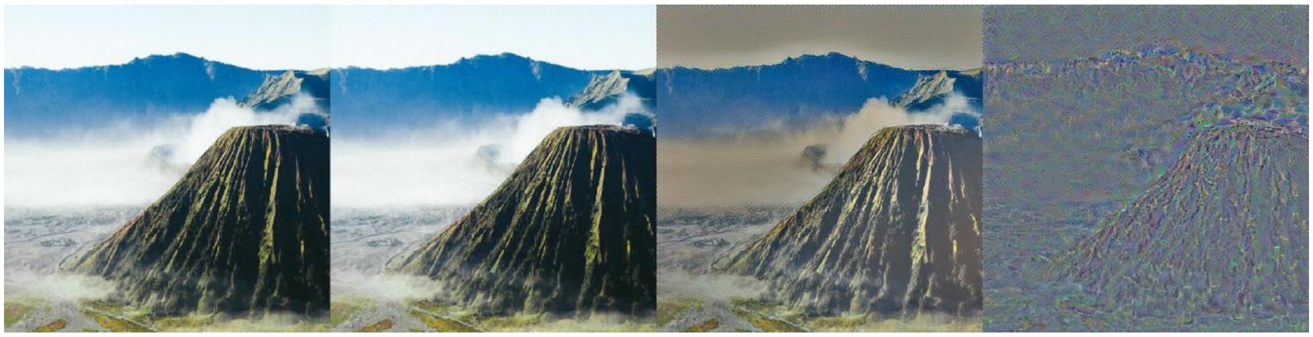
The general approach can be laid out systematically:

(i) Define content image and style image and resize such that they are of same size
(ii) Define a white noise image as the optimization image (also same size with the content and style image). The model will optimize this image such that at the final iteration, this image will be the output with the style (from input style image) applied to the content from the input content image. Normalize the 3 images
(iii) Define content loss by taking mean squared error of the content representation of content image and that of optimization image
(iv) Define style loss using gram matrices (for style representation) and difference between style representation of style image and that of optimization image
(v) Define total loss = content_weight * content loss + style_weight * style_loss
(vi) Update optimization image and minimize total loss until convergence or stopping criteria. Style transferred
(vii) To transfer style for a new pair of content + style image, repeat from step (i)

1. **Content Representation**:

VGG16, when trained on object recognition, will develop a representation of the image that become increasingly explicit along the processing hierarchy [2]. When an input image is passed into VGG16 (or any CNN), along the deeper layers, the feature maps will encode image informations that are increasingly sensitive to the **actual content** of the image, but become **invariant** to the precise appearance. For example, the content of a photo of a dog face is very similar to the content of all other dog face images, even though the color or facial structure may change slightly. In other words, the high-level content of dog face image remains the same, despite the exact pixel values may differ. Thus, VGG16 deeper layers are able to encode such high-level content while the shallower layers simply produce representations that reproduce the exact pixel values [1]. To see that this claim holds, we reconstructed a white noise image to match an input content image and examine the representations produced by using layer 'conv1_2', 'conv3_2', and 'conv5_2:

Original conv1_2 conv3_2 conv5_2



Indeed, the lower layers like 'conv1_2' simply reproduce the original image (so the exact pixel values), the reconstruction from the middle layer 'conv3_2' starts to deviates away from exact reconstruction, while the deeper layers like 'conv5_2' capture the high-level **content** in terms of objects and their arrangement/features, even though pixel values are now very different. The feature maps from these middle to higher layers are therefore called **content representation** of an input image since these layers **encode** the content information of the image [1]. We then proceed to use the content representation to compute content loss described below.

Let us define a content loss. For a layer l, let F^l and P^l be the feature map/response when passing the optimization image and content image as input, respectively. Thus $F^l_{i,j}/P^l_{i,j}$ will be the activation (ReLU) of the i^{th} filter at position j in layer l. In order to transform the optimization image such that it contains the content of the content image, Gatys et al. proposed a content loss function between the content representation of the optimization image (x) and the content representation of the content image (c):

$$L_{content}(c, x, l) = \frac{1}{2}\sum_{i,j}\left(F^l_{i,j} - P^l_{i,j}\right)^2$$

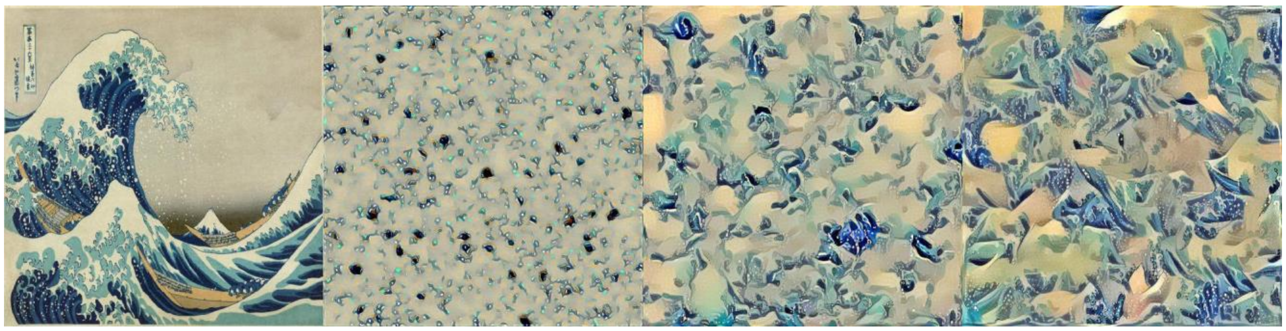
Afterwards, we can calculate $\frac{\partial L_{content}(c,x,l)}{\partial x}$ using back propagation (since $\frac{\partial L_{content}(c,x,l)}{\partial F^l_{i,j}}$ is well defined [1]) and update our initially random/ white noise image x until its feature map at some layer l matches the content image p's feature map at that same layer l. Note that we only update the image x directly, instead of updating any parameters in VGG network. Gatys et al. suggested matching the content representation on only layer "conv4_2"; however, since we use VGG16, it's suggested that we use 'conv2_2', so l = conv2_2 [3].

2. Style representation:

To obtain a style representation from the style input image, we construct a Gram matrix which can be built on top of the feature map in any layer of VGG16 [1]. The Gram matrix consists of the correlations between different filter responses, and thus captures the texture information of an image [2]. Let $G^l \in R^{N_l * N_l}$ be the gram matrix built from feature maps of layer l where N_l is the number of distinct filters. Then, G_{ij}^l is the inner product between the vectorized feature maps i and j in layer l [1]:

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

The basic intuition behind Gram matrix is that by multiplying elements of each combination of feature maps together, we are essentially checking if the elements "overlap" at their location in the image. The summation then discards all the information's spatial relevance. As in the previous part, we can obtain a style reconstruction (for the picture, 1_2 means layer conv1_2):



In contrast to content part, where we produced the matched representation only using 1 layer, for style, Gatys et al. suggested matching/reconstructing using multiple layers. This is justified when we look at the style reconstruction. When we reconstruct using only one layer 'conv1_2', the style produced/matched is still abstract and does not represent the style in the original image. But as we use more layers, for example when we use 'conv1_2', 'conv2_2', 'conv3_3', 'conv4_3', and 'conv5_3', the reconstructed style seems to be more in line with the original image's artistic style.

Let us now define a style loss. Let G^l be the gram matrix of layer l feature maps when white noise image x is passed in and let A^l be the gram matrix of layer l feature maps when input style image a is passed in. With the gram matrix, we can now define our style representation loss for a single layer l:

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

Given the loss per layer, we can obtain a total loss:

$$L_{style}(a,x) = \sum_{l=0}^{L} w_l * L_l$$

where w_l are weight factors of the contribution of each layer to the total loss [1]. We can then compute the gradient $\frac{\partial L_l}{\partial x}$ using back propagation as $\frac{\partial L_l}{\partial F_{ij}^l}$ can be computed easily [1]. Afterwards, we update our initially random/ white noise image x until its feature maps at several (or all) layers $l = 0 \dots L$ matches the style image p's feature maps at that corresponding layers. Gatys et al. suggested using only layers 'conv1_1', 'conv2_1', 'conv3_1', 'conv4_1', and 'conv5_1' to match the style. However, since we used VGG16, layers 'conv1_2', 'conv2_2', 'conv3_3', and 'conv4_3' were used instead [3]. Weight factors w_l is set to 1/(number of active layers) (so = 1/5) for those layers (and 0 for all other layers).

3. Simultaneous content and style matching for style transfer:

Given a content input image p and a style input image a, we can gradually update an image x (originally white noise) such that at the final iteration, x is our output image with the style of image a applied to the content of p. This simultaneous style and content representation matchings can be achieved by minimizing a loss function which is a linear combination of the previously defined content and style loss:

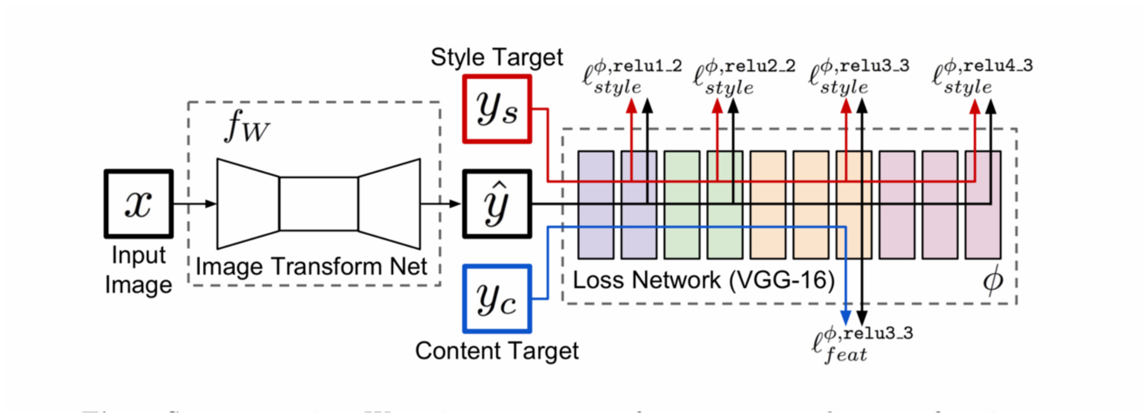
$$L_{total}(p,a,x) = \alpha L_{content}(p,x) + \beta L_{style}(a,x)$$

where α, β are the weight factors for content and style reconstruction, respectively [1].

III. Extension Model – Fast Neural Network Style Transfer:

With the current base model, even though the output images perform well, the key problem with it is image generation speed and model generalization. In other words, because for each image and a corresponding style, we will optimize a particular model for it, it will take a long time to refine the results. It will be a problem when we want to transfer the style of a large number of images. Furthermore, processing style transfer for a video will take a long time to complete. Hence, a model that can perform style transfer quickly as well as generalizable is desirable. In this experiment, we attempted to recreate the model proposed in Johnson et al., utilize it for real-time video style transferring.

The main idea of this method is instead of relying on one model for each input image, we will train a feed-forward network which perform image transformation. After training, this network will perform style transferring for an image by predict the output of a given input image. The network is called Image Transformation Network. Because the goal of the model is to optimize the image style transformation, we need to define a loss function that can help mimic both the style and the content. This is where the perceptual loss proposed by Gatys et al. comes in handy. In the proposed extension model, the output of Image Transformation Network will be used to compute *perceptual loss* of the model. This loss function will be use to optimize the model and update the weights.


Style Target
$\ell_{style}^{\phi,relu1\_2}$
$\ell_{style}^{\phi,relu2\_2}$
$\ell_{style}^{\phi,relu3\_3}$
$\ell_{style}^{\phi,relu4\_3}$
y_s
f_W
x
$\hat{y}$
Input Image
Image Transform Net
y_c
Loss Network (VGG-16)
ϕ
Content Target
$\ell_{feat}^{\phi,relu3\_3}$

In the end, our trained model will be a general model that can transfer one image to a particular style. Each trained model will be responsible for 1 style. We thus find W* such that:

$$W^* = argmin_W \boldsymbol{E}_{x,\{y_i\}}\left[\sum_{i=1} \lambda_i l_i(f_W(x), y_i)\right]$$

where $f_W(x)$ is the transformation output of input x. l_i is the loss with respect to the target output at layer i (y_i). λ_i is the weight of the losses. The objective is to minimize the perceptual loss given by an output image. The losses we use are the Style loss, Content loss and Total variation regularizer.

1. Image Transformation Model:

The image transformation model proposed in Johnson et al. consists of 3 blocks of layers

a. Convolution and Deconvolution block:

The convolution block consists of 3 layers. The first one is a Convolution2D, then a Batch Normalization layer and last layer will be the Activation layer of the block. The activation layer will be ReLU for input and hidden blocks, and Tanh for output block. We use Tanh for the last block because the input is a normalized image, thus we'd like the output to also be a normalized image with value ranges [-1,1] which can then be denormalized back into a RGB image of pixel range [0,255]. In summary, the convolution block is:

Conv2D -> Batch Normalization -> Activation (ReLU or Tanh)

The deconvolution block has the same structure as the convolution block, but we use Conv2DTranspose instead of Conv2D. Thus the block structure is:

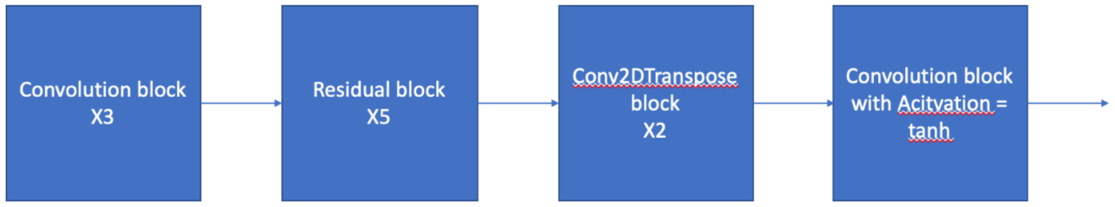
Conv2DTranspose -> Batch Normalization -> Activation (ReLU)

b. Residual block:

Residual blocks are used as the middle layers of the model, with the following structure:

Input -> Conv2D -> Batch Norm -> Activation (ReLU) -> Conv2D -> Batch Norm -> Add(input, Batch Norm output)

Overall, the final model will first down sample the input with 2 convolution block, push it through 5 residual blocks and finally up sample through 2 deconvolution blocks. The reason behind down sampling is that this helps increases the training speed of the model and yields a better performance. Note that the first and last Convolution blocks are for input and outputting the image, while the hidden layers are used for image transformation. The network representation is as follows:



2. Loss Network:

Following the algorithm proposed by Gatys et al., we will use the same content loss and style loss as defined in the base model. The only exception is that Johnson et al. model will use VGG16 instead of VGG19. Since we need the VGG network to compute content and style loss, we can think of Johnson et al. model as an image transformation model combined with the VGG network (figure above). Because Johnson et al. used VGG16, they suggested matching the content representation on layer 'conv3_3' (instead of 'conv4_2') and matching the style representation on layers 'conv1_2', 'conv2_2', 'conv3_3', 'conv4_3'. Additionally, Johnson et al. also introduced a total variation regularization L_{TV}. This regularizer is used to smooth out and de-noise the output image of the model:

$$L_{TV} = \sum_{i,j} \sqrt{\left|y_{i+1,j} - y_{i,j}\right|^2 + \left|y_{i,j+1} - y_{i,j}\right|^2}$$

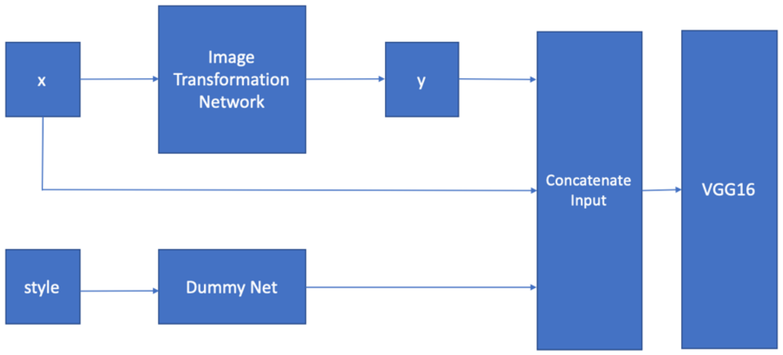
where y is the output image array and $y_{i,j}$ is the corresponding pixel value at i,j.

3. Combining the networks:

As discussed above, since we need VGG network to compute style and content loss, we can think of Johnson et al. model as a combination of image transformation network and a VGG network. However, in order to cooperate the image transformation net with VGG network, there are a few key points:

(i) In our model design, image transformation network has input x and output y. Input x is the image we want to transfer the style, which means we want to our output y to match the "content" of x. We then combine transformation net with the loss network using a Concatenate layer that concatenate x and y and feed the concatenation into VGG16/loss network. By concatenation, we have a continuous model from input image x to the end of VGG16.

(ii) Since our model also takes into consideration the ***style*** of an image, we create a **style dummy net** where its output is the same as its input. The justification for this is that we want to create a continuous model. Hence, this dummy net will output the normalized style image and concatenate it to the Concatenate layer. Thus the overall architecture when implemented is:



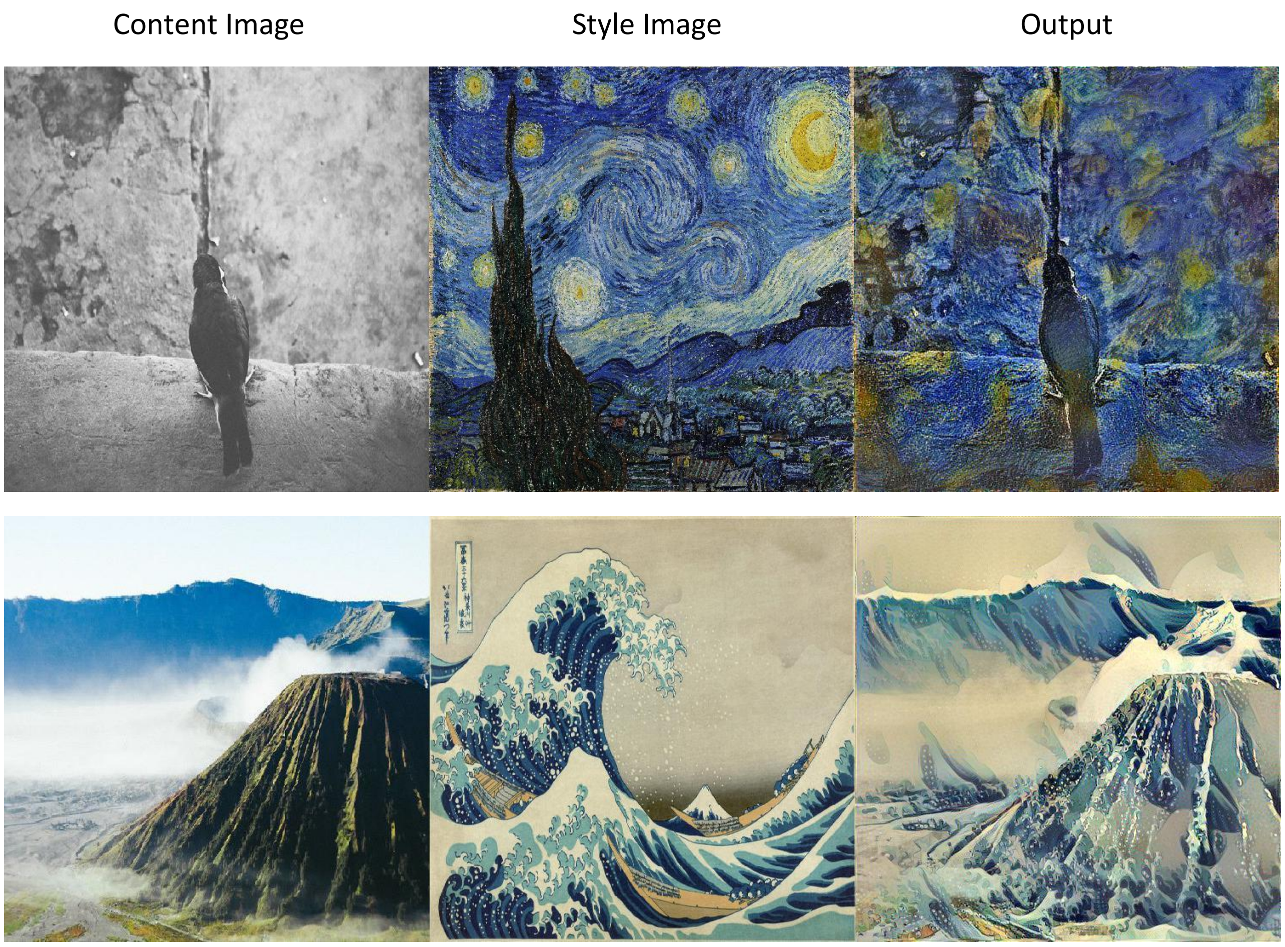
(iii) Since our model has no ground truth and rather an optimization problem, our target value for the Loss Network will be a dummy target with 0 value. We also create a dummy loss function which returns 0.0 while compiling the Loss Net using keras. We then implement the loss by adding a regularization on each of the designated content and style layer. By doing so, the loss in the end is still cumulated correctly while keeping the model implementation easier [5]

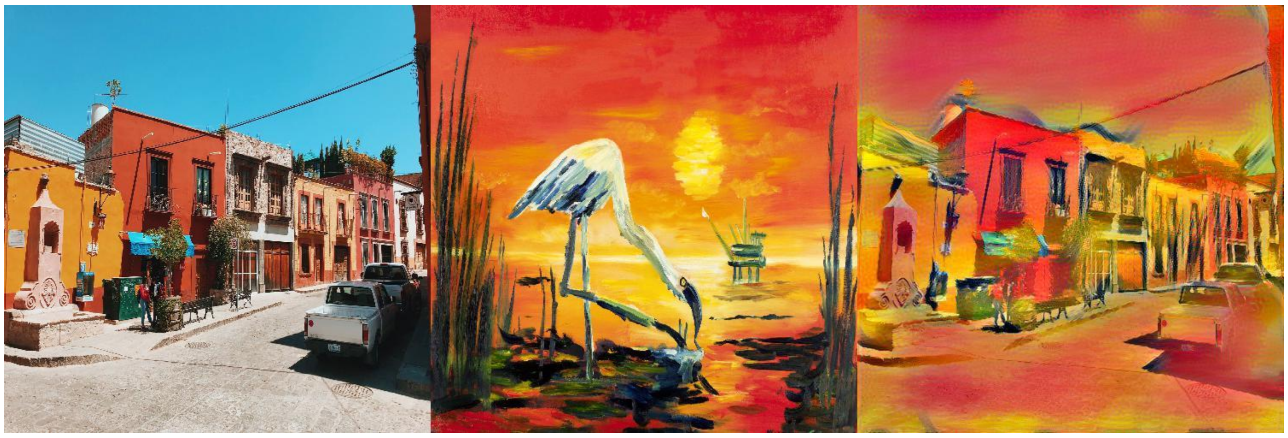
(iv) Weights from VGG16 will not be updated, but rather only the weights from image transformation net will be updated

IV. Results:

1. Base Model Results:

Our implementation included helpers such as normalizing/denormalizing images retrieved from [6]. We used both gray-scale and colored images. We set content weight $\alpha = 0.1$, and style weight $\beta = 5000$ for all inputs. We resized all images to 500x500(x3) for consistency and computation speed. We used L-BFGS as optimization algorithm as suggested by Gatys et al. and set the number iterations to 20. We thus obtained the following results:



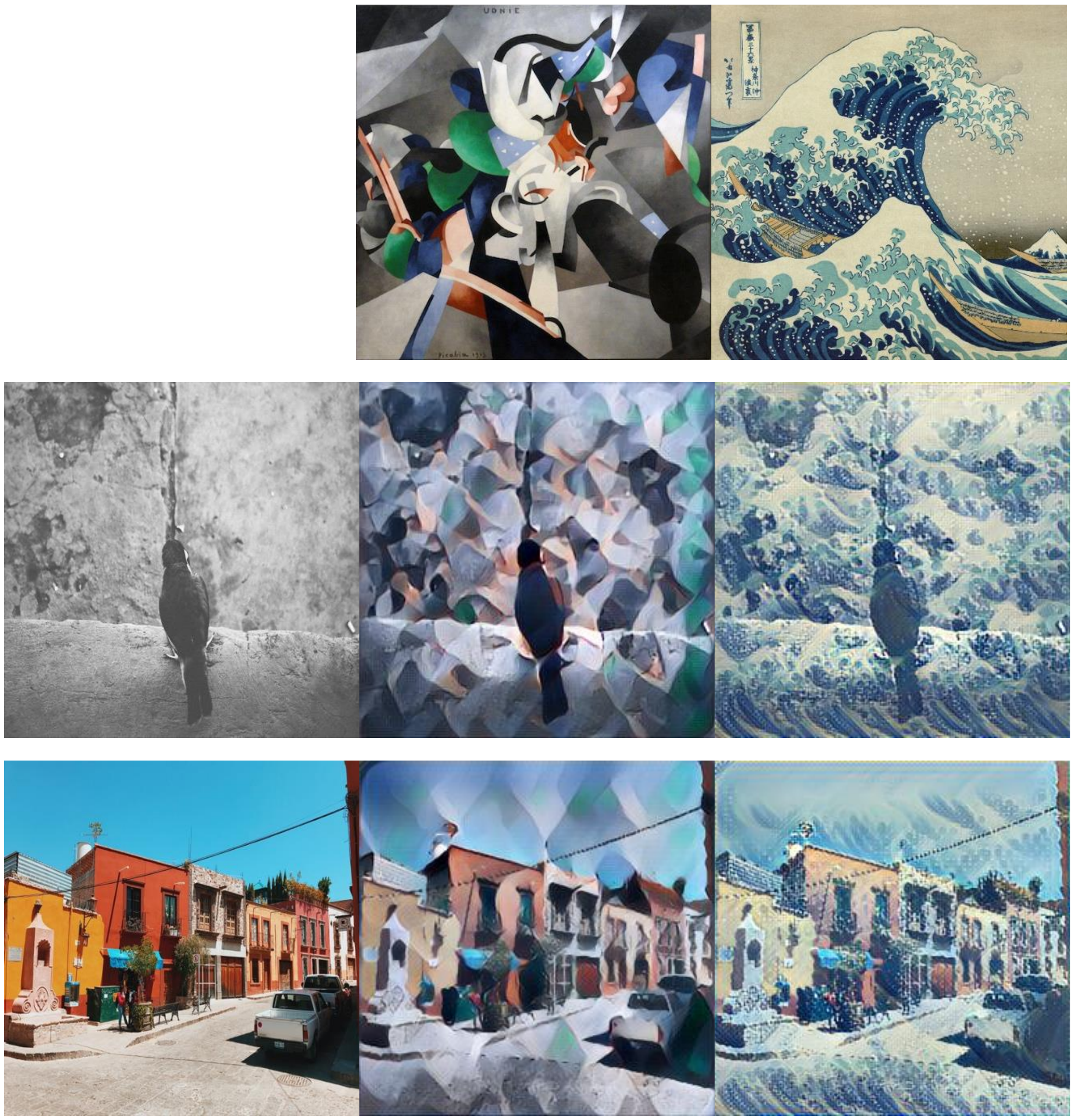


An important thing to note is that we decided to choose VGG16 over VGG19 because VGG16 seemed to be able to separate style and content better and the resulting images were smoother and more coherent than those generated when using VGG19. For VGG19, even though having followed the specifications of the authors, we could not reproduce the style transferred images with the same quality as those in the paper. Additionally, we found, through experimenting, that VGG16 produced consistent quality results with the same content and style weight (0.1 and 5000 respectively). While with VGG19, we had to change the content and style weight around for each new pair of content + style image in order to achieve reasonably good style transfer.

2. Extension Model Results:

The model was trained using MS COCO dataset which consists of more than 82,000 images for training. For faster computation, all images will be resized to 256x256, though, the model can be generalized for larger images as well as producing a higher resolution output. The optimizer used was Adam with the learning rate of 0.001. The default weight of total variation regularizer ranges from 10^{-3} to 10^{-6} and the weights of style and content loss vary depending on the desired style and content synthesis [3]. According to Johnson et al., the model was trained for roughly 2 epochs per image with a batch size of 4. Each style image will be trained on a different model.

Due to resource and time constraint, we cannot train the full model on the dataset which is roughly 12GB. Instead, we will still construct the image transformation model, but then we will load the pre-trained weights provided by [4] to the implemented model. The set of images below shows the result from using the pre-trained weights of the style from Udnie and the Wave. The model runs much faster than the base model, transferring at the rate of 0.5 second per image:


UDNIE

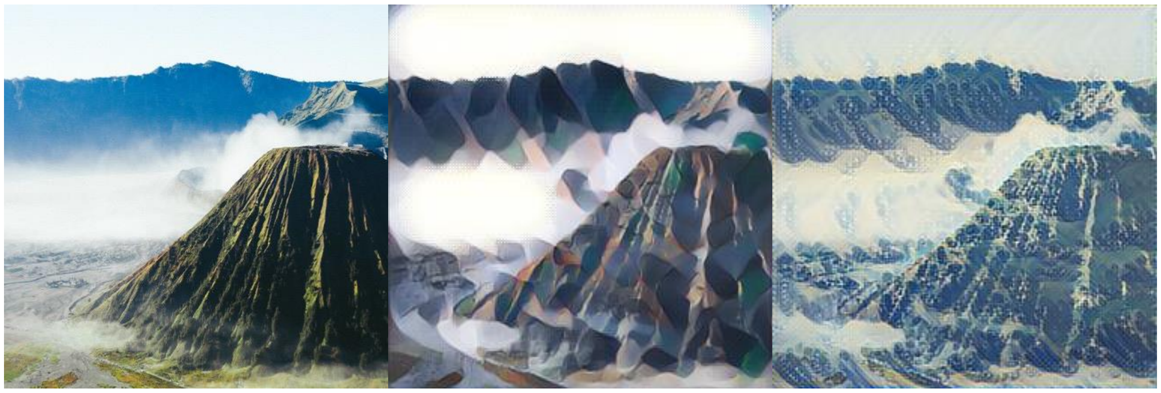


Image: Results from the pre-trained model

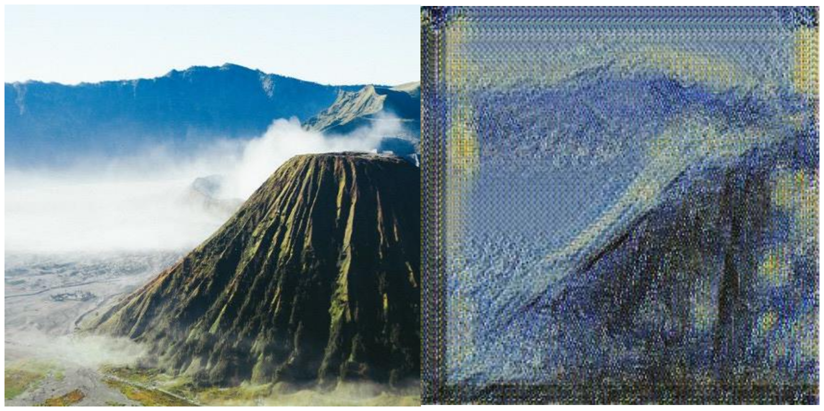
Firstly, qualitatively speaking, compared to the base model, the output styled images are of the same perceptual quality. It is clear, from the results produced above, that the trained model is aware of the separation between semantic content and texture information of images [3]. Additionally, the images produced by the extension model are actually a bit smoother than those produced by base model, due to the presence of a total variation regularizer that helps smooth out and denoise.

Secondly, when using 1 GTX 1080 ti on pictures of size 256x256x3, the extension model finished style transferring in 0.4 seconds while the base model needed 2 minutes and 14 seconds. We can see that the extension model is ~300 times faster than the base model which is crucial since we are able to offset the slow training time by extremely fast prediction time, making this suitable for video as well as real time style transfer. For video, we separate each frame, perform style transferring and convert it back to a 30 fps video:

Image: Sample frame from video

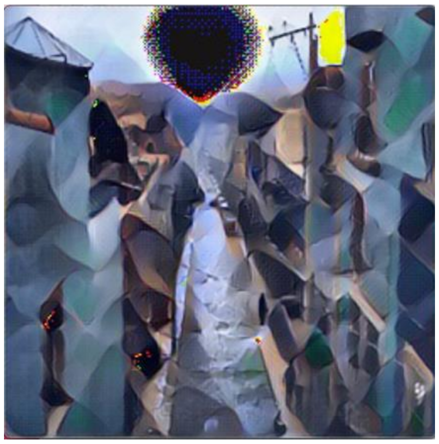
Due to the extremely fast style transferring, we can also perform real-time style transfer via webcam. Similar to how we processed video, only this time, we perform real-time style transfer using the local computer webcam. Since we ran on a local computer with limited computing power (using only CPU), style transfer was pretty slow. However, if forward computation is performed on GPU or using more powerful resources, real-time style transfer can be achieved. The webcam demo resides in **Pretrained-Fast-Neural** notebook.

On an additional note, as mentioned above, due to resource and time constraint, we could not train the full model on all of the training set. However, we did implement the full model and trained on a training set of 10 images, running for roughly 100 epochs. The resulting image for the matching of the mountain picture and the art piece "Starry Night" is below:



Though not significant, we can clearly see that the output image matched (right hand side) can both preserve the content as well as capturing some of the style from "Starry Night". The code for the full model and its training is available in **Fast-Neural-Net** notebook.

On the last note in this result section, we'd like to address a problem when evaluating the model which is the way we handle image processing. The last layer of the image transformation net is a Tanh activation layer, so the value will range from -1 to 1. However, we want to convert the array back to RGB in range of [0,255] and depending on the calculations, the picture can be too dark or overexposed, like this example below:



To have the best consistent results, let x be our predicted arrays, we can transform and denormalize x into the final image X:

$$X = floor\left(x * 127.5 + \frac{255}{2}\right)$$

To recap, the extension model exhibited similar qualitative result to Gatys et al. algorithm and is much faster at style transferring, allowing for video and real-time style transfer.

V. References:

[1] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. 2016. Image Style Transfer Using Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[2] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. 2015. Texture Synthesis Using Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*.

[3] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual Losses for Real-Time Style Transfer and Super-Resolution. 2016. In *The European Conference on Computer Vision*.

[4] Overflocat. 2018. A Keras Implementation of Fast-Neural-Style. Github. Retrieved from: https://github.com/overflocat/fast-neural-style-keras

[5] Sam Lee. 2017. A Fast Neural Style Transfer Implement with Keras 2. Github. Retrieved from: https://github.com/misgod/fast-neural-style-keras

[6] Greg Surma. 2018. Style Transfer – Styling Images with Convolutional Neural Networks. Towards Data Science. Retrieved from: https://towardsdatascience.com/style-transfer-styling-images-with-convolutional-neural-networks-7d215b58f461