Universal Style Transfer via Feature Transforms

E4040.2018Fall.YYZA.report

Shiyun Yang sy2797, Manqi Yang my2577, Yizhi Zhang yz3376

Columbia University

Abstract

This project reproduces the paper 'Universal Style Transfer via Feature Transforms' in TensorFlow. The objective is to transfer arbitrary visual styles onto content images to generate an artistic creation. Our implementation achieves the objective with slightly lower performance than the original paper. The main technical challenge we encountered is the long training time of training five separate networks (decoders), but we overcame this challenge by using TensorFlow record to represent training dataset and by altering the decoder structure to reduce feature sizes.

1. Introduction

Image Style Transfer is an artistic creation process that synthesizes a new image by transferring the style of the style image to the content image. The main challenge of traditional style transfer methods, either optimization-based or using feed-forward networks, is the tradeoff between style generalization, visual quality, and computational efficiency. Different from existing methods, the 'universal style transfer' method proposed by our selected paper does not require training on any predefined style image; instead, it uses trained image reconstruction decoders to generate the synthesized image.

The key task in style transfer is extracting features of the style image so that they could be transferred onto the content image. Gram matrix [3] extracted through trained DNN is good at representing visual styles, so one way to implement style transfer is by minimizing the Gram/covariance matrix based loss function [4]. Similarly, our paper's approach matches the feature covariance matrix of the content image to a given style image, but through a direct and training-less feature transform process called "whitening and coloring transforms" (WCTs).

2. Summary of Original Paper

This section contains two subsections: 2.1 Methodology of the Original Paper, where we discuss the main techniques proposed by the paper; and 2.2 Key Results of the Original Paper, where we briefly summarize the results of the paper.

2.1 Methodology of the Original Paper

Different from using DNN and a designed loss to synthesize style image and content image, WCT is an effortless work - a mathematical formula that implement style synthesis directly. To get a basic understanding of WCT, the formula of WCT will be presented in two parts: F_c and f_c are the VGG feature maps extracted from content and style image through encoder, while decoder will reconstruct the original content image from f_c.

Whitening:

$$\widehat{f_c} = E_c D_c^{-\frac{1}{2}} E_c^T f_c \tag{1}$$

The uncorrelated content feature map $\widehat{f_c}$can be reached: $\widehat{f_c}\widehat{f_c}^T = \mathrm{I}$.

Coloring:

$$\widehat{f_{cs}} = E_s D_s^{\frac{1}{2}} E_s^T \widehat{f_c} \tag{2}$$

We could obtain: $\widehat{f_{cs}}\widehat{f_{cs}}^T = f_s f_s^T$.

Thus re-stylize the content image with the style of style image.

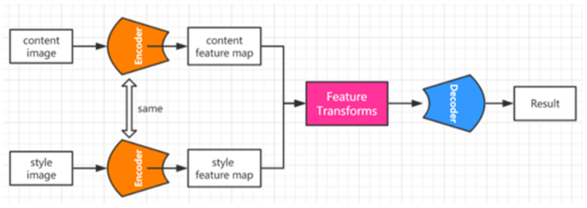


Figure 1 Feed-forward method to realize the fast transferring for arbitrary styles.

This is the intuition about WCT, and the detail will be discussed later.

The main work of this paper is training the decoder (recover the image from features) according to defined loss. This paper employ VGG-19 network as the encoder and train a symmetric decoder of VGG-19 encoder, to reconstruct the image from features. For the encoder(feature extractor), this paper used the pre-trained weights. For the decoder, it trained separately five decoders to capture features of different levels separately. The final model will combine the five encoders and five decoders, and feed the content image and style image in a reverse order to get a better result.

2.2 Key Results of the Original Paper

The original paper firstly proposes to use transforms: WCT. Secondly uses the trained encoder-decoder network to implement the style transform by simple feed-forward operations. The proposed methods achieved effectiveness results and can be extended to other fields like texture synthesis.

3. Methodology

This section discusses our methodology in reproducing the paper and contains the following subsections: 3.1 Objectives and Technical Challenges, where we describe our goals and challenges; 3.2 Problem Formulation and Design, which contains detailed description of our approach.

3.1 Objectives and Technical Challenges

Our *objectives* for this project are the following:

a. Train five decoders to perform feature inversion, i.e. reconstruct image from the feature maps extracted by VGG encoder.
b. Implement WCT process to perform feature transforms from style images to content images.
c. Construct the multi-level coarse-to-fine stylization structure where lower levels capture simpler features and higher levels capture more complex features.

Technical Challenges:

One challenge we faced is understanding and implementing the whitening and coloring feature transforms (WCTs). The WCT process is discussed very briefly in the original paper, so additional resources were needed for us to interpret the detailed implementation of the process.

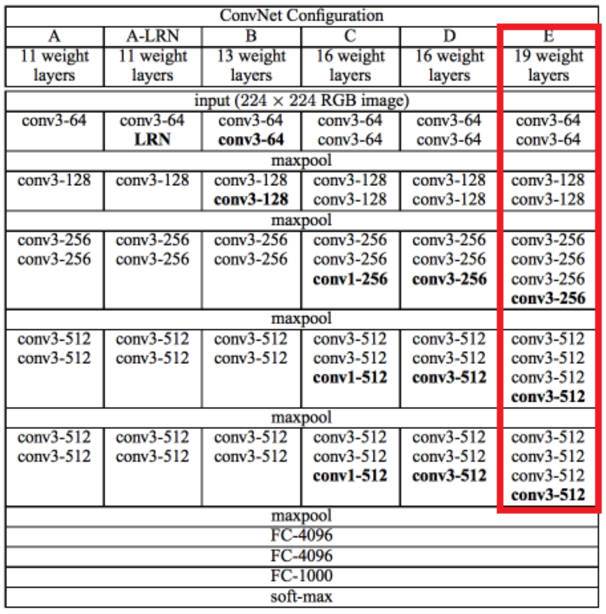
Another challenge is that, in the original paper, the 5 reverse-VGG decoders are trained on the entire Microsoft COCO dataset, which contains over 80,000 images; hence, one major challenge for us is the training process, and we reasonably expect that our results will show worse performance compared to the original paper due to training on smaller dataset and for smaller iterations.

3.2 Problem Formulation and Design

The following subsections provide detailed description of our approach to implement the methods proposed by the paper.

3.2.1 VGG19 Encoder

The first part of the image reconstruction task is to extract features from input images, i.e. we need to encode the images into feature maps. The encoder we use for this part is modified from the VGG19 neural network and has 20 layers, which are the first 20 layers from the original network. The following table shows the original VGG16 and 19 networks built by the Visual Geometry Group [5], and the layers we used is marked with a red rectangle:



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 2 VGG19 network built by Visual Geometry Group.

The structure of our VGG encoder is shown below:

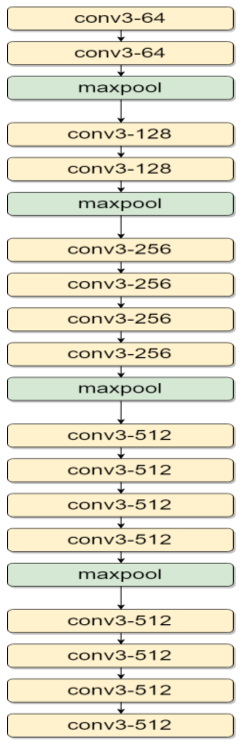


Figure 3 VGG19 Encoder Structure (20 layers)

The input to the VGG encoder is 224x224x3, so we reshape every image to 224x224 pixels and convert to RGB format if the image is in greyscale.

The output of the VGG encoder is the extracted feature map of the image, which has size 14x14x512.

To reduce training time, we downloaded pre-trained VGG19 weights and used it for our encoder. The file we downloaded is called "vgg19.npy".

3.2.2 Reverse VGG Decoder

The second part of the image reconstruction task is to decode the feature maps and revert them back to RGB images. The decoder we use for this part is symmetric to our VGG encoder with the layers reversed. Figure 4 demonstrates the structure of the encoder-decoder pair [6].

To reverse the max-pooling layer, we apply upsampling (unpooling) before the convolutional layers. Denote the input size by (height, width), then the upsampling process resizes the input to (2*height, 2*width) using TensorFlow's nearest neighbor interpolation method. After upsampling, we apply the same convolutional layers as in the encoder but in reverse order.

To train the decoder, we use subsets of the Microsoft COCO dataset, which contains a total of 82,783 images in either RGB or greyscale format.

The loss in training is defined as the sum of pixel loss and feature loss [7], where pixel loss is the mean squared error (squared L2 norm) between the input image and output (reconstructed) image, and feature loss is the mean squared error between the feature map of the input image and the feature map of the output image. The loss equation is as Equation (3):

$$L = \|I_o - I_i\|_2^2 + \|\Phi(I_o) - \Phi(I_i)\|_2^2 \tag{3}$$

, where I_o is the output image, I_i is the input image, and $\Phi(.)$ is the VGG encoder.

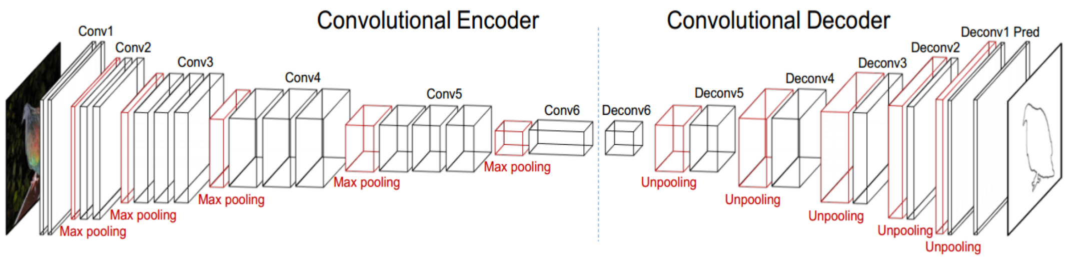


Figure 4 Structure of symmetric encoder-decoder pair; in our implementation, both encoder and decoder have 20 layers.

3.2.3 Whitening and Coloring Transforms (WCTs)

As discussed above in the Introduction section, our style transfer method takes two inputs, a content image and a style image, and outputs a single, synthesized image with the style of the style image and the content of the content image.

WCT is the key part of our style transfer method and consists of two processes: whitening transform, where the “style” features of the content image are removed; and coloring transform, where the features of the content image are transformed to match the features of the style image.

Whitening transform:

To obtain the whitened image, we need to transform the feature map of the content image (C) so that it has no correlation, which requires that the covariance matrix of the feature map is an identity matrix.

Denote the feature map of the content image (computed from VGG encoder) by f_c, then we first need to subtract the mean value from f_c:

$$\boldsymbol{f_c} \in \mathbb{R}^{C \times H_c W_c}$$
$$\mu_c = E[\boldsymbol{f_c}]$$
$$\boldsymbol{f_c} = \boldsymbol{f_c} - \mu_c$$

Denote the transformed feature map we want by f_{c_t} , then we need to find the transform matrix **W** such that:

$$\boldsymbol{f_{c\_t}} = \boldsymbol{W} \cdot \boldsymbol{f_c}$$
$$\boldsymbol{f_{c\_t}} \cdot \boldsymbol{f_{c\_t}}^T = \boldsymbol{I}$$

A common way to find **W** is the following [8]:

$$\boldsymbol{X} = \boldsymbol{f_c} \cdot \boldsymbol{f_c}^T$$
$$\boldsymbol{W} = \boldsymbol{X}^{-\frac{1}{2}}$$

Since **X** is a symmetric matrix with real values, we can use eigen decomposition to represent **X**: $\boldsymbol{X} = \boldsymbol{E_c D_c E_c}^T$, where $\mathbf{E_c}$ is an orthogonal matrix whose columns are the eigenvectors of **X**, and $\mathbf{D_c}$ is a diagonal matrix whose diagonal elements are the corresponding eigenvalues of **X**.

Let **A** be $\boldsymbol{A} = \boldsymbol{X}^{-\frac{1}{2}} = \left(\boldsymbol{E_c D_c E_c}^T\right)^{-\frac{1}{2}}$,

then we can compute **W** by the following steps:

$$\boldsymbol{A}^2 = (\boldsymbol{E_c D_c E_c}^T)^{-1} = \boldsymbol{E_c D_c}^{-\mathbf{1}} \boldsymbol{E_c}^T$$
$$\boldsymbol{A} = \boldsymbol{E_c D_c}^{-\frac{1}{2}} \boldsymbol{E_c}^T$$
$$\boldsymbol{W} = \boldsymbol{X}^{-\frac{1}{2}} = \boldsymbol{A} = \boldsymbol{E_c D_c}^{-\frac{\mathbf{1}}{2}} \boldsymbol{E_c}^T$$

Therefore, the whitening transformation equation is:

$$\boldsymbol{f_{c\_t}} = \boldsymbol{W} \cdot \boldsymbol{f_c} = \boldsymbol{E_c D_c}^{-\frac{1}{2}} \boldsymbol{E_c}^T \boldsymbol{f_c}$$

Coloring transform:

The coloring transform is an inverse version of the whitening transform in which we apply a new style to the style-removed content image feature map. To obtain the synthesized (style-transferred) image, we need to match the covariance matrix of the whitened content image to that of the style image. Hence, we need to find $\mathbf{W_{cs}}$ such that:

$$\boldsymbol{f_s} \in \mathbb{R}^{C \times H_s W_s}$$
$$\boldsymbol{f_{cs\_t}} = \boldsymbol{W_{cs}} \cdot \boldsymbol{f_{c\_t}}$$
$$\boldsymbol{f_{cs\_t}} \cdot \boldsymbol{f_{cs\_t}}^T = \boldsymbol{f_s} \cdot \boldsymbol{f_s}^T$$

Where f_{cs_t} is the feature map of the colored content image that we want, f_{c_t} is the whitened content image from the previous step, and f_s is the mean-subtracted feature map of the style image:

$$\mu_s = E[\boldsymbol{f_s}]$$

$$\boldsymbol{f_s} = \boldsymbol{f_s} - \mu_s$$

Because $f_{c\_t} * f_{c\_t}^T = \mathrm{I}$, we have:

$$\boldsymbol{f_{cs\_t}} \cdot \boldsymbol{f_{cs\_t}}^T = (\boldsymbol{W_{cs}} \cdot \boldsymbol{f_{c\_t}}) \cdot (\boldsymbol{W_{cs}} \cdot \boldsymbol{f_{c\_t}})^T$$

$$= \boldsymbol{W_{cs}} \cdot \boldsymbol{f_{c\_t}} \cdot \boldsymbol{f_{c\_t}}^T \cdot \boldsymbol{W_{cs}}^T = \boldsymbol{W_{cs}} \cdot \boldsymbol{I} \cdot \boldsymbol{W_{cs}}^T$$

$$= \boldsymbol{W_{cs}} \cdot \boldsymbol{W_{cs}}^T$$

Same as in the whitening process, we can compute f_{s_t} by the following:

$$\boldsymbol{f_{s\_t}} = \boldsymbol{W} \cdot \boldsymbol{f_s} = \boldsymbol{E_s D_s}^{-\frac{1}{2}} \boldsymbol{E_s}^T \boldsymbol{f_s}$$

$$\boldsymbol{f_{s\_t}} \cdot \boldsymbol{f_{s\_t}}^T = \boldsymbol{I}$$

We then obtain the expression for fs as below, and derive the equation for $f_s * f_s^T$:

$$\boldsymbol{f_s} = \left(\boldsymbol{E_s D_s}^{-\frac{1}{2}} \boldsymbol{E_s}^T\right)^{-1} \boldsymbol{f_{s\_t}} = \boldsymbol{E_s D_s}^{\frac{1}{2}} \boldsymbol{E_s}^T \boldsymbol{f_{s\_t}}$$

$$\boldsymbol{f_s} \cdot \boldsymbol{f_s}^T = \left(\boldsymbol{E_s D_s}^{\frac{1}{2}} \boldsymbol{E_s}^T \boldsymbol{f_{s\_t}}\right)$$

$$\cdot \left(\boldsymbol{E_s D_s}^{\frac{1}{2}} \boldsymbol{E_s}^T \boldsymbol{f_{s\_t}}\right)^T$$

$$= \left(\boldsymbol{E_s D_s}^{\frac{1}{2}} \boldsymbol{E_s}^T\right) \cdot \boldsymbol{f_{s_t}} \cdot \boldsymbol{f_{s_t}}^{\boldsymbol{T}} \cdot \left(\boldsymbol{E_s D_s}^{\frac{1}{2}} \boldsymbol{E_s}^T\right)^T$$

$$= \left(\boldsymbol{E_s D_s}^{\frac{1}{2}} \boldsymbol{E_s}^T\right) \cdot \boldsymbol{I} \cdot \left(\boldsymbol{E_s D_s}^{\frac{1}{2}} \boldsymbol{E_s}^T\right)^T$$

$$= \left(\boldsymbol{E_s D_s}^{\frac{1}{2}} \boldsymbol{E_s}^T\right) \cdot \left(\boldsymbol{E_s D_s}^{\frac{1}{2}} \boldsymbol{E_s}^T\right)^T$$

Now we can solve the equation for $\mathbf{W_{cs}}$:

$$\boldsymbol{f_{cs\_t}} \cdot \boldsymbol{f_{cs_t}}^T = \boldsymbol{f_s} \cdot \boldsymbol{f_s}^T$$

$$\Downarrow$$

$$\boldsymbol{W_{cs}} \cdot \boldsymbol{W_{cs}}^T = \left(\boldsymbol{E_s D_s}^{\frac{1}{2}} \boldsymbol{E_s}^T\right) \cdot \left(\boldsymbol{E_s D_s}^{\frac{1}{2}} \boldsymbol{E_s}^T\right)^T$$

Therefore, we can choose $\mathbf{W_{cs}}$ to be $\boldsymbol{E_s D_s}^{\frac{1}{2}} \boldsymbol{E_s}^T$, and the coloring transformation equation is:

$$\boldsymbol{f_{cs\_t}} = \boldsymbol{W_{cs}} \cdot \boldsymbol{f_{c\_t}} = \boldsymbol{E_s D_s}^{\frac{1}{2}} \boldsymbol{E_s}^T \boldsymbol{f_{c\_t}}$$

After obtaining f_{cs_t}, we need to add the mean of the style image to f_{cs_t} to get the final image:

$$\boldsymbol{f_{cs\_t}} = \boldsymbol{f_{cs\_t}} + \mu_s$$

User control:

Same as in the paper, we add user control on the strength of stylization effects by adding a parameter alpha to blend f_{cs_t} with the original feature map of the content image fc as shown by the equation below:

$$\boldsymbol{f_{cs\_t}} = \alpha \boldsymbol{f_{cs\_t}} + (1 - \alpha) \boldsymbol{f_c}$$

When $\alpha = 0$, the output image is the content image. When $\alpha = 1$, the output image is the most stylized image.

3.2.4 Multi-Level Stylization

To achieve better result, the paper uses a technique called Multi-level Stylization, in which the style image is applied five times to the content image with the utilization of five encoder-decoder pairs instead of one. The multi-level network structure is shown as Figure 5 [6].

In order to implement this multi-level structure, we need to have 5 separate encoders and 5 separate decoders. The VGG19 encoder pre-trained weights are downloaded from [9]. For the decoders, we have to train each one separately then use the best model for each decoder in our final style transfer process.

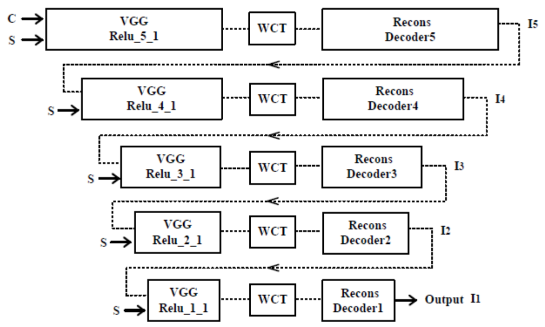


Figure 5 Multi-level stylization process; we reproduce the technique of the original paper and use the same coarse-to-fine (decoder 5 to decoder 1) stylization.

In the original implementation given by the paper, the 5 encoders output the feature maps at the following 5 layers of the VGG19 network: Relu_1_1, Relu_2_1, Relu_3_1, Relu_4_1, Relu_5_1. However, to make training the decoders easier and to reduce train time, we decided to modify the encoders slightly, such that the 5 encoders output the feature maps at the following 5 layers: Maxpool_1, Maxpool_2, Maxpool_3, Maxpool_4, and the final layer (conv3_512) of the VGG19 encoder.

The structure of the 5 encoders and 5 decoders in our implementation is shown in Figure 6.

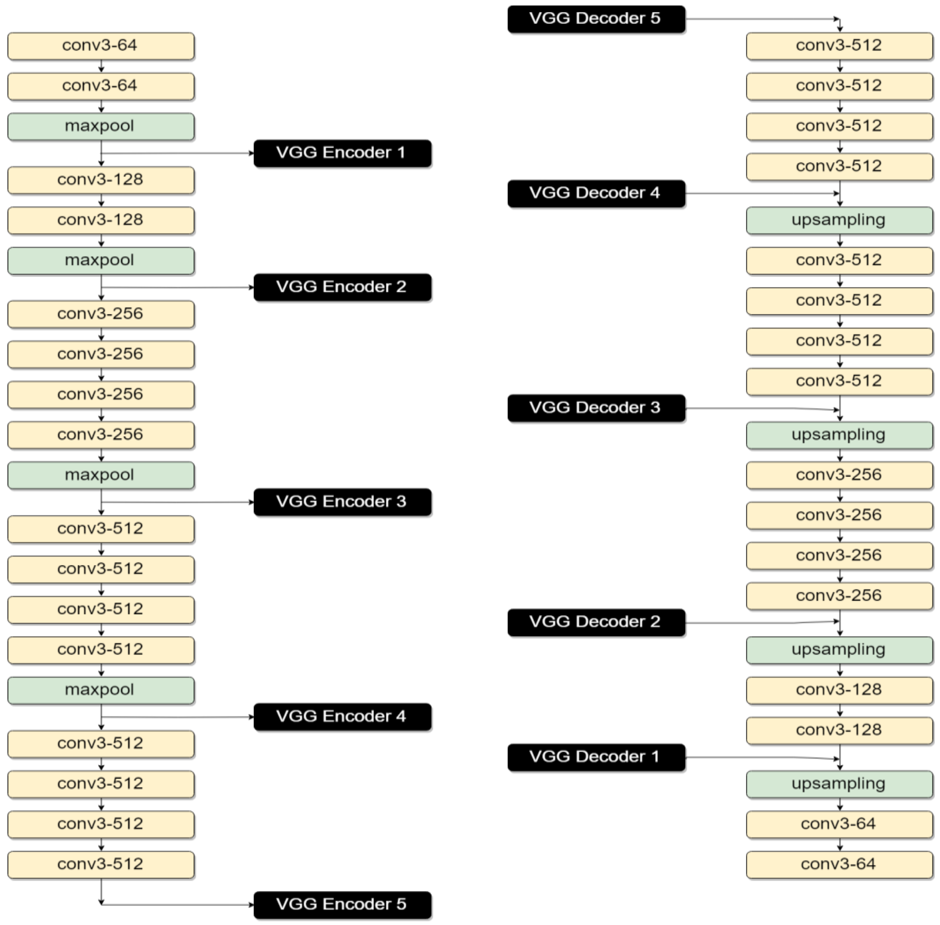


Figure 6 Complete structure of our VGG encoder network and reverse VGG decoder network.

3.2.5 Single-Level Stylization Test

In addition to generating the multi-level stylization result, we also implemented a single level stylization to test the stylization effect of each of the decoders. The single level stylization process is shown in Figure 7.

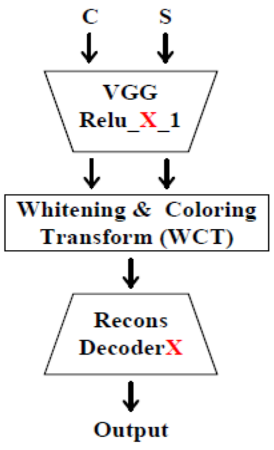


Figure 7 Single level stylization process; this process uses only one encoder-decoder pair to generate a stylized image and is used to test the stylization effect of each encoder-decoder pair.

For example, if we want to test the stylization effect of level 3, we can use encoder 3 and decoder 3 to generate a stylized image of two input images.

As discussed in the paper, the lower levels capture simpler features such as color, while the higher levels can capture more complex features such as content.

4. Implementation

This section contains the detailed implementation of our project and our software design.

4.1 Implementation Overview

In this section, two main flow charts will be introduced: training process and style transfer procedure. The detail of each function will be discussed in detail in the next section.

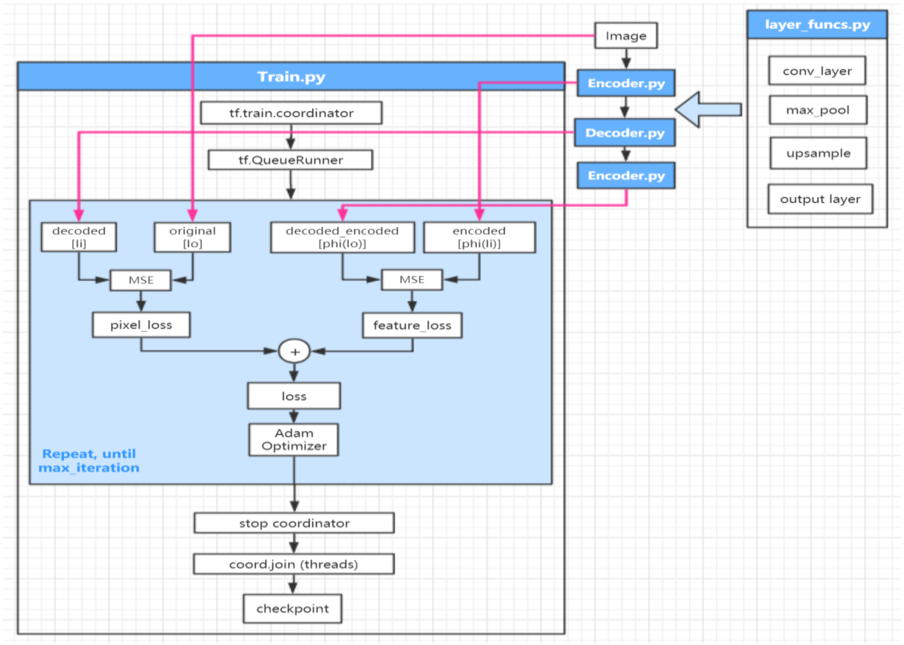


Figure 8 flowchart of training process.Train.py implements the training procedure. Layer_funcs.py is the support of the encoder.py and decoder.py, which is the main structure of the network. Tf.train.coordinator and tf.QueueRunner will implement the multiple threads in tf session, which will help to speed up the training procedure.

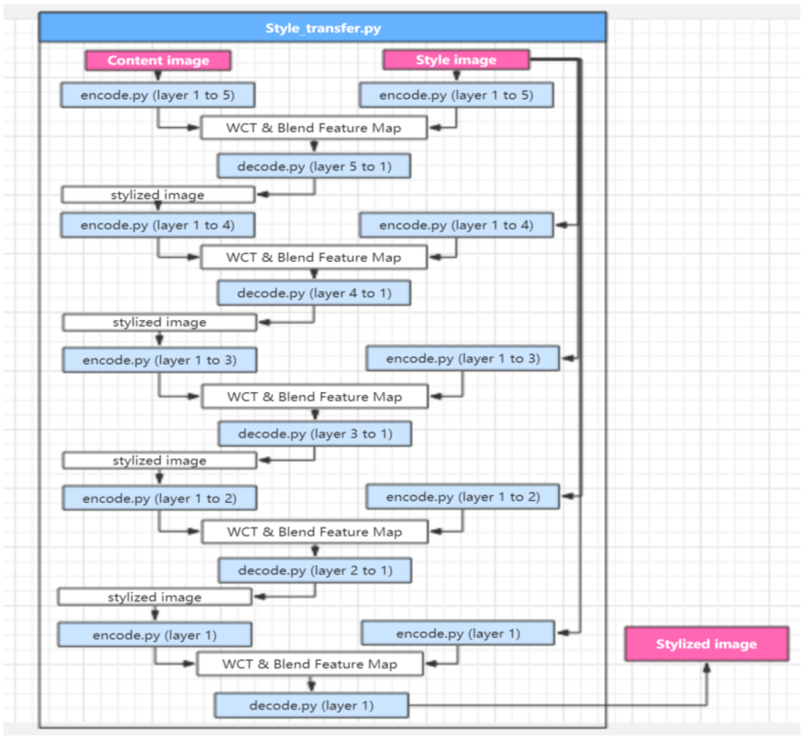


Figure 9 Flowchart of network structure for style transfer. Style_transfer.py implements the final style transfer of the content image given a desired style image. It divides VGG-19 structure into 5 parts and use two symmetric VGG-19 networks in total. Thus, there are 5 encoders and 5 decoders. Between encoder and decoder, we use wct.py to implement WCT and Blend the content feature map and style feature map as we wanted.

4.2 Detailed Description

This subsection describes each file and each function in our code.

4.2.1 Encoder.py

This python file contains the Vgg19 class we use for our encoders which has the following functions:

a. Initialization function
The initialization function loads the pre-trained VGG19 weights in vgg19.npy and imports the convolutional layer and maxpool layer from layer_funcs.py
b. Encoder function
The main encoder function takes an image and a target layer as inputs and generate the encoded feature map of the image. The target layer parameter specifies which encoder we want to use, e.g. if we want to use encoder 3, we would set target layer to 'relu3', then the function would encode the image using only layers in encoder 3. This function returns the encoded feature map of the input image.

4.2.2 Decoder.py

This python file contains the Decoder class, which has the following functions:

a. Initialization function
The initialization function imports layers and loads the corresponding encoder weights from vgg19.npy, e.g. if using decoder 3, we need to load weights of encoder 3.
b. Decoder function

The main decoder function takes a feature map and a target layer as inputs and revert the feature map back to image. Same as for encoder, the target layer parameter specifies which decoder we want to use. This function returns the decoded image of the input feature map.

4.2.3 layer_funcs.py

This python file contains all the neural network layers we used in both the encoders and the decoders. The layers are defined the same as in VGG19.

The encoder layers are the following:

a. Maxpool layer.
uses tf.nn.max_pool; reduces both height and width to ½.
b. Convolutional layer.

This layer function (conv_layer) includes the convolutional layer followed by a Relu layer. Uses tf.nn.conv2d and tf.nn.relu. The input and output sizes are the same.

The decoder layers are the following:

a. Upsample layer.
This is the "inverse" of the maxpool layer in which the input is resized using Nearest Neighbor Interpolation. The output height and width are twice of the input height and width.
b. Convolutional layer (decoder).

This is almost the same as the convolutional for encoders but with two additional parameters: in_channel and out_channel. Since we are decoding in reverse order of VGG19, we have to reduce the convolutional layer channel size. For example, when going from conv3_512 to conv3_256, we set in_channel=512, out_channel=256 so that the result can be inputted again to the next conv3_256.

4.2.4 tfrecords.py

This python file contains the code to generate TensorFlow Record file from our training dataset [10].

4.2.5 train.py

This python file contains the Train_Model class, which has the following functions:

a. Initialization function

The initialization function sets the following input parameters: target layer (which decoder we want to train), number of iterations for training, training batch size. It also loads the the VGG19 model to encoder.

b. Encoder_decoder function

This function takes an input image and computes the following:

Encoded = feature map obtained by processing the original image with the encoder;

Decoded = reconstructed image obtained by processing the original image with both the encoder and decoder;

Decoded_encoded = feature map obtained by processing the reconstructed image with the encoder.

c. Train function

This is the main training function. The training data set is fed to the network using the tfrecord file. Loss is defined as the sum of pixel loss and feature loss. We use Adam optimizer with learning rate of 1e-4.

4.2.6 wct.py

This python file contains the code to perform the Whitening and Coloring Transformations as discussed in 5. The inputs are the content image, style image, and alpha. The output is the feature map of the synthesized (stylized) image.

4.2.7 single_layer.py

This python file contains the single_layer class and is used to generate single level stylization result as discussed in 3.2.5. The single_layer class has the following functions:

a. Initialization function

The initialization function sets the following parameters: target layer (which encoder-decoder pair we want to use), alpha, encoder (loads pre-trained vgg19 encoder), decoder (loads Decoder class), and decoder weights (loads trained decoder weights specified by input path).

b. Test function

The main test function uses one (selected) encoder-decoder pair to generate stylized images. We use the PIL package to load the images and convert them to numpy arrays.

4.2.8 style_transfer.py

This python file contains the style_transfer class and performs the main task of our project: it takes content images and style images as inputs and generate stylized images for each pair of content-style images. The style_transfer class has the following functions:

a. Initialization function

The initialization function sets the following parameters: alpha, encoder (loads pre-trained vgg19 encoder), decoder (loads the Decoder class), and decoder weights (loads trained decoder weights for all decoders). We also need to specify content path, style path, and output path (path to store output images).

b. Test function

This main test function generates a stylized image for each content-style image pair using all 5 decoders. The process structure is shown in 6 multi-level stylization.

4.3 Hardware Specification

GCP Setup:

Machine type: n1-standard-4 (4 vCPUs, 15 GB memory). CPU platform: Intel Haswell. Disk:100GB. GPUs: 4 * NVIDIA Tesla K80

5. Results

5.1. Project Results

We train the five reconstruction decoders for features at the VGG-19 Relu_X_1 (X=1,2,...,5) layer separately as the paper does. Here we denote them as Relu X (X=1,2,3,4,5) in the following for short. The same as the original paper, we use Microsoft COCO dataset [11] and set the λ (parameter which balance the feature loss and pixel loss) to 1.
Here we firstly show the details of training results at each layer ReluX (X = 1,2,…,5) in Figure 10. Then show the stylized images that fed-forward through all layers in Figure 11.

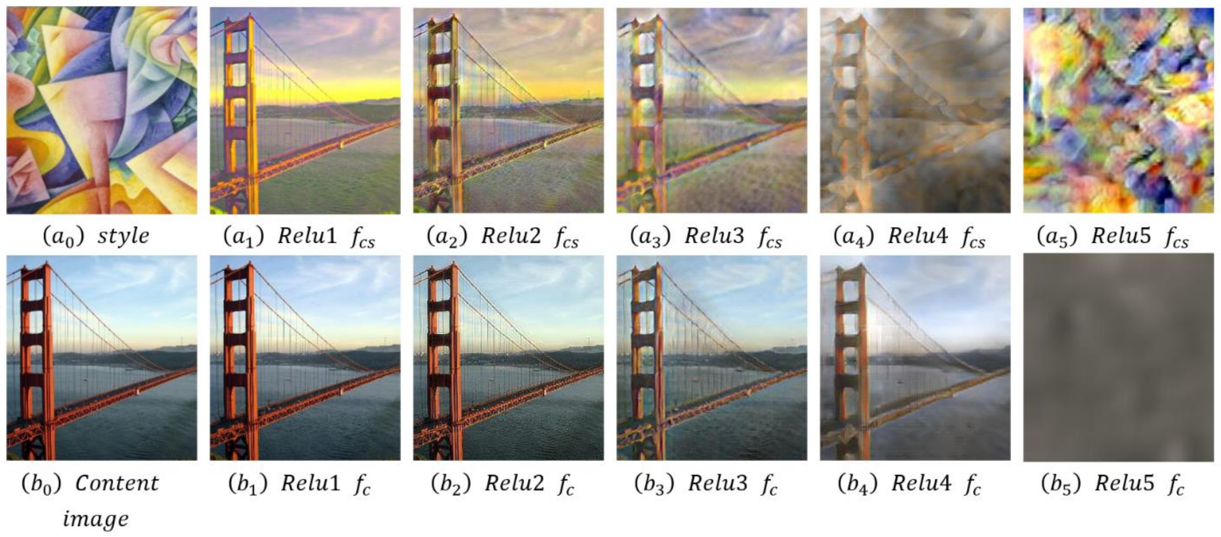


Figure 10 Single-level stylization using different VGG features. Row 1 contains stylized image from each single layer. Row 2 contains recovered image from each decoder.

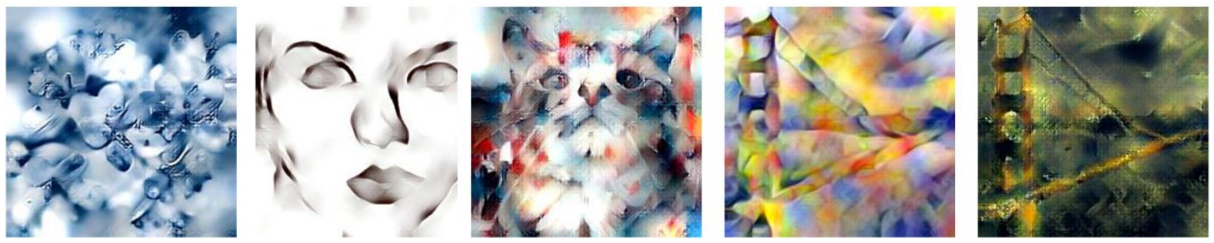


Figure 11 Feed-forward through all layers.

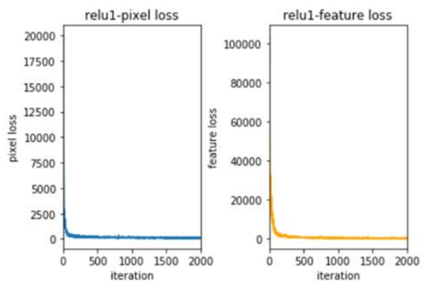
Layer	Relu 1	Relu 2	Relu 3	Relu 4	Relu 5
Iteration	2k	2k	10k	10k	20k
Dataset Size	10k	10k	10k	80k	80k
Training time	30min	35 min	2h 10 min	5h 20 min	6h 30 min

Table 1 Training details.

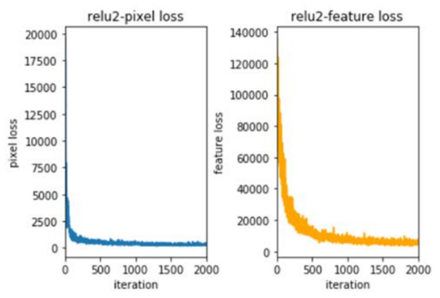
Firstly, we use reduced size of dataset (10k images) to train decoder for 2k iteration. We find that it works for low-level layer, the loss curve converges when iteration reaches 2000 at Relu 1, Relu 2. However, 2000 iteration seems not enough for Relu 3 and higher layers. In Figure 12, we could see the intermediate results for 2000-iteration Relu 3. Thus, we use 80k dataset and 10k iteration to train higher layer.

However, even though we trained 20k iteration with 80k dataset. The final image is still very bad. We have to discard Relu 5 and use only Relu 1-Relu 4 to stylize image. The following results we generated are without Relu 5.

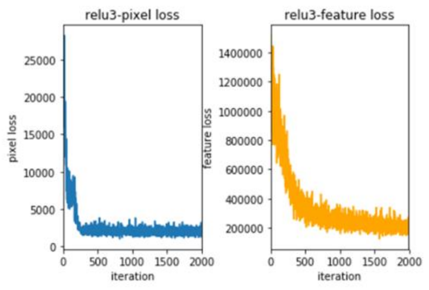
We can see that in general, pixel loss converges faster than feature loss. As well, when layer goes higher, both loss converges slower (especially feature loss) and oscillates more drastically.



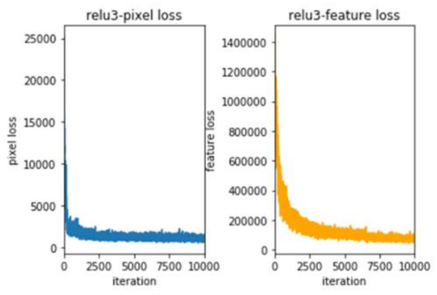
Relu 1 - 2k iterations



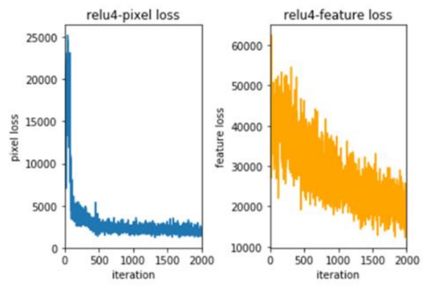
Relu 2 - 2k iterations



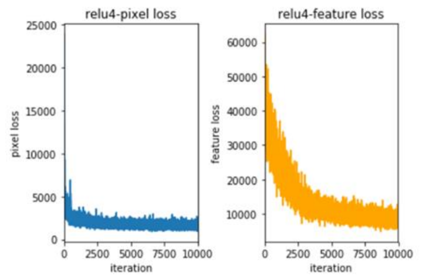
Relu 3 - 2k iterations



Relu 3 - 10k iterations



Relu 4 - 2k iterations



Relu 4 - 10k iterations

Figure 12 Pixel loss and feature loss plot in Relu 1-Relu 4

5.2.Comparison of Results

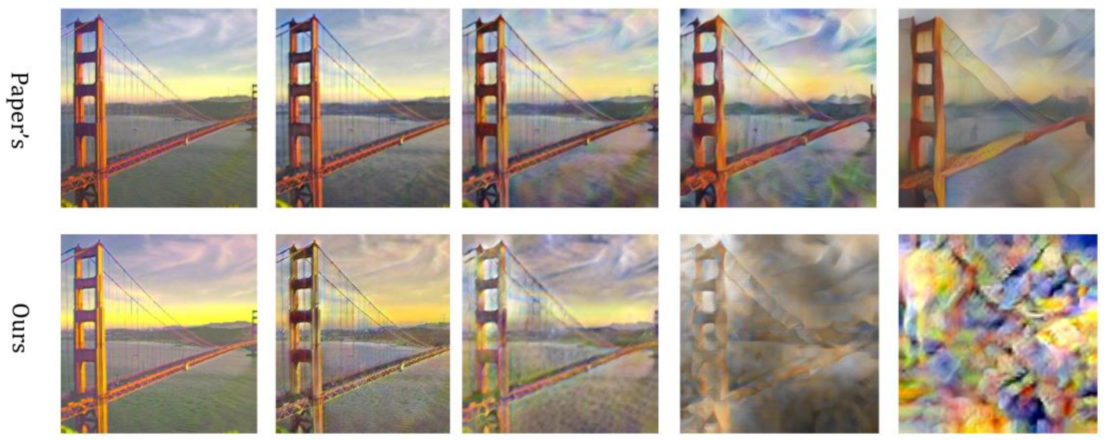


Figure 13 Single layer stylization. Our results are satisfying from Relu 1- Relu 3. In Relu 4, there are distortion.

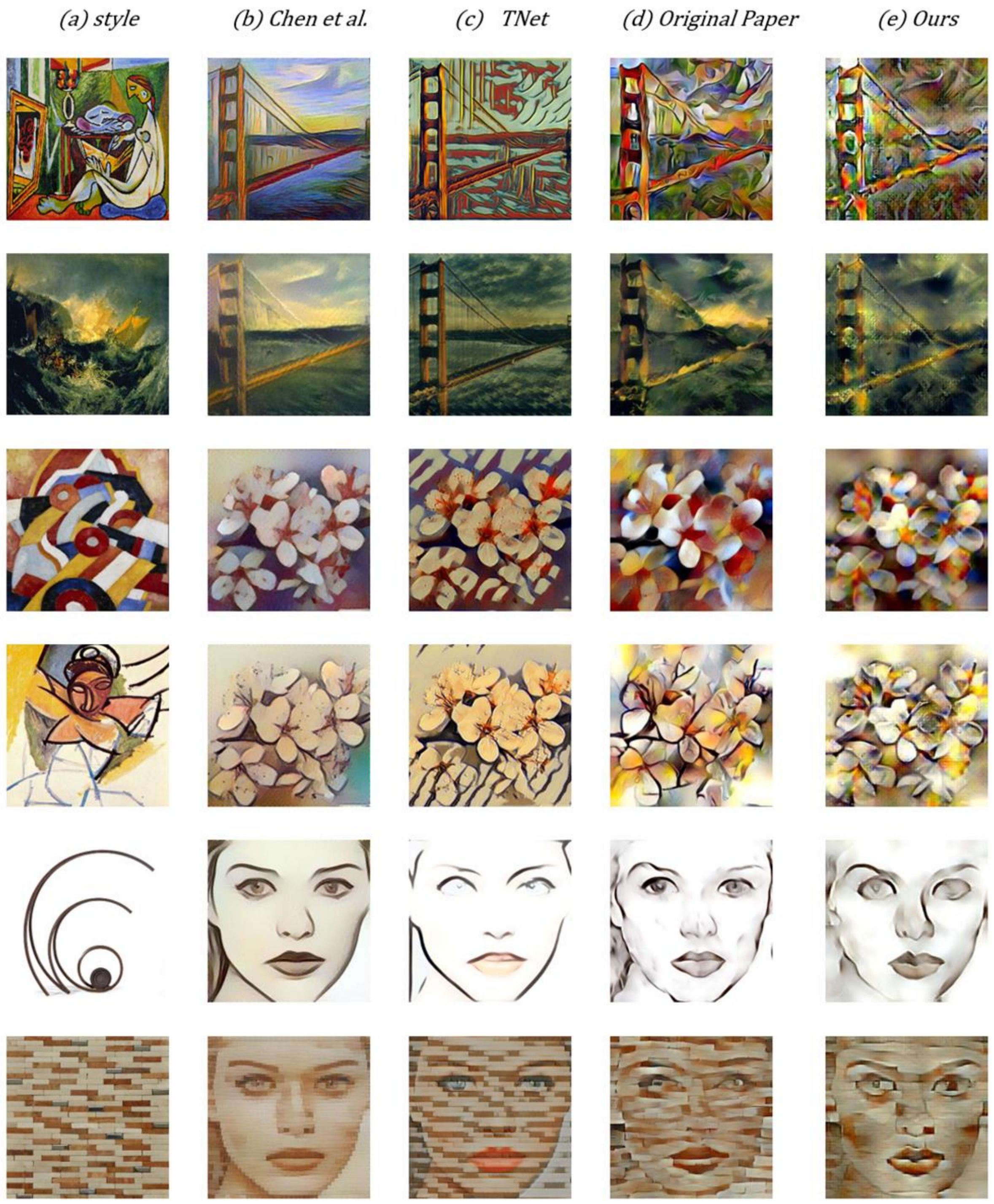


Figure 14 Multilayer stylization. Here we provides comparison with the original paper and other two papers. We set $\alpha = 0.6$ as the paper suggests to obtain the best styling effect.

In this section, firstly the results through 5 separate single layer will be compared in Figure 13.

Then the results fed-forward the whole network will be compared in Figure 14.

5.3. Discussion of Insights Gained and Problems Faced

In the single layer stylization, we find some color distortion in our results compared with the paper's results. One possible reason is inadequate training. Here are some findings during training attempts.

From intermediate results of one layer, we could explore the characters of feature discovering. The structure of an image, e.g. shape of the bridge, will be discovered before the color feature. Figure 15 describes the reconstructed image from the decoder($\alpha = 0$) under different training iterations and training set of Relu 4. We expect the same image as the original content. However, we could see that the structure of the original image is discovered but the color is distorted.

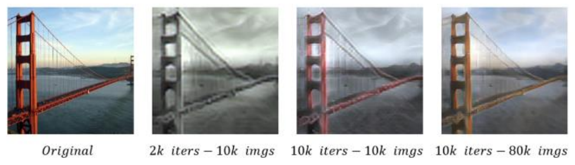


Figure 15 Feature discovering

However, due to hardware limitation, we could not afford more training iteration. The training of 20k iteration and 80k images takes 6 hours, which almost reaches our limit.

Hyperparameters set contains learning rate, training input/output image size, testing image size and other network parameters. We directly use the VGG19 parameters to train our model.

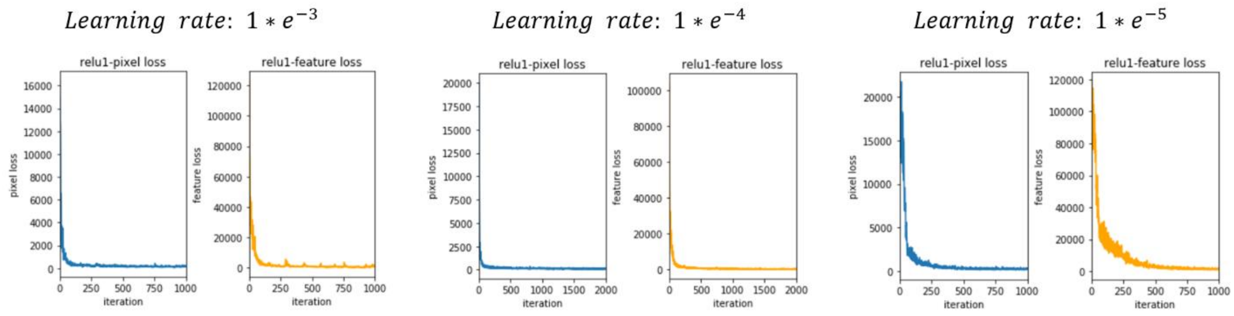


Figure 16 We could see that at layer Relu 1, learning rate = 1e-4 performs best. Because of high computation cost of higher layer, we set learning rate = 1e-4 as an experienced value. The paper does not give the parameter set they use. Thus, we cannot make a comparison. Hyperparameters difference maybe one reason of different results

6. Conclusion

In this project, we correctly reconstruct the network structure, design proper training and testing methods and generate satisfying results in image style transfer. However, we meet some training challenges that make us still a bit distant from the methods in the original paper. During the process, we have deeper understanding of TensorFlow, encoder & decoder and VGG network. What's more, we obtained precious experience in practice in using neural network to solve problems and using TensorFlow to realize a neural network. Possible future improvements include using a better-performing neural network as encoders (e.g. ResNet) or introduce additional hyperparameters (e.g. learning decay).

7. Acknowledgement

We would like to show our gratitude to all the TAs of ECBM4040 for providing us valuable advices and patient assistance throughout the semester. We are also grateful for our professor Zoran Kostic for igniting our interest in machine learning and for being a wonderful instructor.

8. References

[1] BitBucket Links: https://bitbucket.org/ecbm4040/2018_assignment2_sy2797/src/master/

[2] H. Li, "Author Guidelines for CMPE 146/242 Project Report", Lecture Notes of CMPE 146/242, Computer Engineering Department, College of Engineering, San Jose State University, March 6, 2006, pp. 1.

[3] Y. Li, C. Fang, J. Yang, Z. Wang, X. Lu, and M.-H. Yang. Diversified texture synthesis with feed-forward networks. In CVPR, 2017

[4] Desai, Shubhang. "Neural Artistic Style Transfer: A Comprehensive Look." Medium.com, Medium, 14 Sept. 2017, medium.com/artists-and-machine-intelligence/neural-artistic-style-transfer-a-comprehensive-look-f54d8649c199.

[5] Simonyan, K., & Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. CoRR, abs/1409.1556. https://arxiv.org/abs/1409.1556v6

[6] Sarafianos, Nikolaos. ICIP 2016: Deep Learning Trends and Open Problems. 11 Oct. 2016, nsarafianos.github.io/icip16.

[7] Li, Yijun et al. "Universal Style Transfer via Feature Transforms." NIPS (2017). https://arxiv.org/pdf/1705.08086.pdf

[8] Ungrapalli, Vignesh. "Universal Style Transfer – Towards Data Science." Towards Data Science, Medium, 20 Dec. 2017, towardsdatascience.com/universal-style-transfer-b26ba6760040.

[9] Machrisaa, Chris. "Machrisaa/Tensorflow-Vgg." GitHub, 12 June 2017, github.com/machrisaa/tensorflow-vgg.

[10] Gamauf, Thomas. "Tensorflow Records? What They Are and How to Use Them." Medium.com, Medium, 20 Mar. 2018, medium.com/mostly-ai/tensorflow-records-what-they-are-and-how-to-use-them-c46bc4bbb564.

[11] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: Common objects in context. In ECCV, 2014.

[12] T. Q. Chen and M. Schmidt. Fast patch-based style transfer of arbitrary style. arXiv preprint arXiv:1612.04337, 2016.

[13] D. Ulyanov, V. Lebedev, A. Vedaldi, and V. Lempitsky. Texture networks: Feed-forward synthesis of textures and stylized images. In ICML, 2016.

9. Appendix

9.1. Individual Student Contributions in Fractions

uni	**my2577**	**sy2797**	**yz3376**
Last Name	Yang	Yang	Zhang
Fraction of (useful) total contribution	1/3	1/3	1/3
What I did 1	⅓ Code	⅓ Code	⅓ Code
What I did 2	⅓ Report	⅓ Report	⅓ Report