Neural Network Project Demo

July 2019

1. Topic

In this project I built a neural network model to transform a given image in the style of another image. At the end I made a GUI in Tkinter to show my results.

2. Dataset

The COCO dataset (2014) was used for content images. It has around 83K training images and 41K validation images. The dataset is available at http://cocodataset.org/#download. Some examples of the training images from this dataset is shown in Figure 1.

The dataset provided by Kaggle for the competition 'Artists by Numbers' was used for style images. It has around 79K training images and 23K validation images. The dataset is available at https://www.kaggle.com/c/painter-by-numbers/data. Some example of the training images from this dataset is shown in Figure 2.

During training, random crops of size (256, 256) were taken for both content and style images. During inference any image size is accepted.

Figure 1: Sample Images from COCO dataset

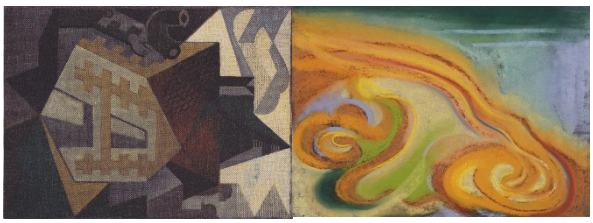

Figure 2: Sample Images from Kaggle dataset

3. Baseline

As a baseline model I trained a neural network which stylizes a given content image to a fixed style. This style image has to be pre-decided while training the network.

3.1 Network

The simple encoder-decoder architecture proposed in [1] is trained here. The network takes a content image of size of 256 x 256 x 3 as input and outputs the stylized image of size 256 x 256 x 3. We represent the input content image as c, the fixed style image as s and the network output image as o. The network uses residual blocks shown in Figure 3. Both the convolution blocks have the same parameters. The architecture of the network is shown in Table 1.

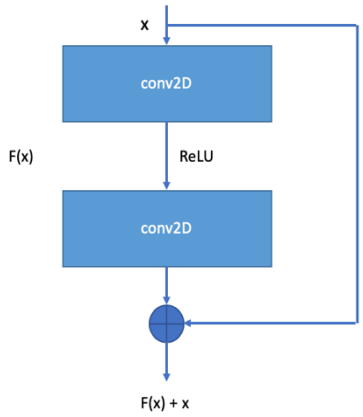

Figure 3: Residual Block Architecture

LAYER	ACTIVATION SIZE	KERNEL	STRIDE
Input(Image)	3 x 256 x 256		
Conv2D_1	32 x 256 x 256	9 x 9	1
ReLU_1	32 x 256 x 256		
Conv2D_2	64 x 128 x 128	3 x 3	2
ReLU_2	64 x 128 x 128		
Conv2D_3	128 x 64 x64	3 x 3	2
ReLU_3	128 x 64 x 64		
Residual_1	128 x 64 x 64	3 x 3	1
Residual_2	128 x 64 x 64	3 x 3	1
Residual_3	128 x 64 x 64	3 x 3	1
Residual_4	128 x 64 x 64	3 x 3	1
Residual_5	128 x 64 x 64	3 x 3	1
Conv2D_3	64 x 128 x 128	3 x 3	0.5
ReLU_4	64 x 128 x128		
Conv2D_4	32 x 256 x 256	3 x 3	0.5
ReLU_5	32 x 256 x 256		
Conv2D_5	3 x 256 x 256	9 x 9	1
ReLU_6	3 x 256 x 256		
Output (Image)	3 x 256 x 256		

Table 1: Architecture of baseline model

3.2 Loss

The loss is a combination of two loss terms. We wish the output image to have a style similar to the style image and content same as the content image. This gives us two loss terms, style loss and content loss.

We use perceptual loss terms to define style and content loss. We use the VGG -16 network [2] pretrained on the image-net dataset [3] to evaluate these losses. The VGG network is denoted by ϕ and $\phi_i(x)$ denotes activations of the i^{th} layer of the network ϕ when processing the image x. We denote the size of $\phi_i(x)$ as C_i x H_i x W_i.

The content loss is expressed in the equation below. The i^{th} layer here corresponds to relu2_2 of VGG16.

$$L_c = \frac{1}{C_i H_i W_i} ||\emptyset_i(c) - \emptyset_i(o)||_2^2$$

To evaluate the style loss, firstly $G_i(x)$, the gram matrix is obtained from the i^{th} activation layer ϕ_i. Gram matrix is evaluated efficiently by reshaping $\phi_i(x)$ into $\varphi_i(x)$ of size C_i x (H_i*W_i) and then using the equations below.

$$G_i(x) = \varphi(x)\varphi^T(x)/C_i H_i W_i$$

The MSE loss evaluated across various activations of VGG and summed to obtain the final style loss as can be seen in the equation below. Here, activations from relu_1_2, relu_2_2, relu_3_3, relu_4_3 of VGG16 are taken.

$$L_s = \sum_i ||G_i(s) - G_i(o)||_2^2$$

The total loss is a weighted sum of these two loss terms. λ_c and λ_s are the weights for content and style loss respectively.

$$L = \lambda_c L_c + \lambda_s L_s$$

3.3 Training Details

The network I trained was trained for the mosaic image shown in Figure 3. The model was trained for 20K iterations using the Adam optimizer. The model is coded in Pytorch and trained from scratch. Table 2 shows the hyper-parameters tried and the final parameters selected. I used tensorboard to plot and keep account of the train and validation loss. Figure 5 shows the train loss with increasing iterations and Figure 6 shows the same for validation. Train Loss was saved every 100 iterations and Validation Loss was saved every 500 iterations. Also, the model checkpoints were saved every 10000 iterations.

Figure 4: Mosaic Image – Style Image for baseline model

Parameter	*Range*	*Final*
batch size	4 to 8	4
learning rate	0.001 to 0.0005	0.001
weight (style loss)	0.0005 - 0.005	0.001
weight(content loss)	0.5 - 2	1

Table 2: Hyper Parameter Search for baseline

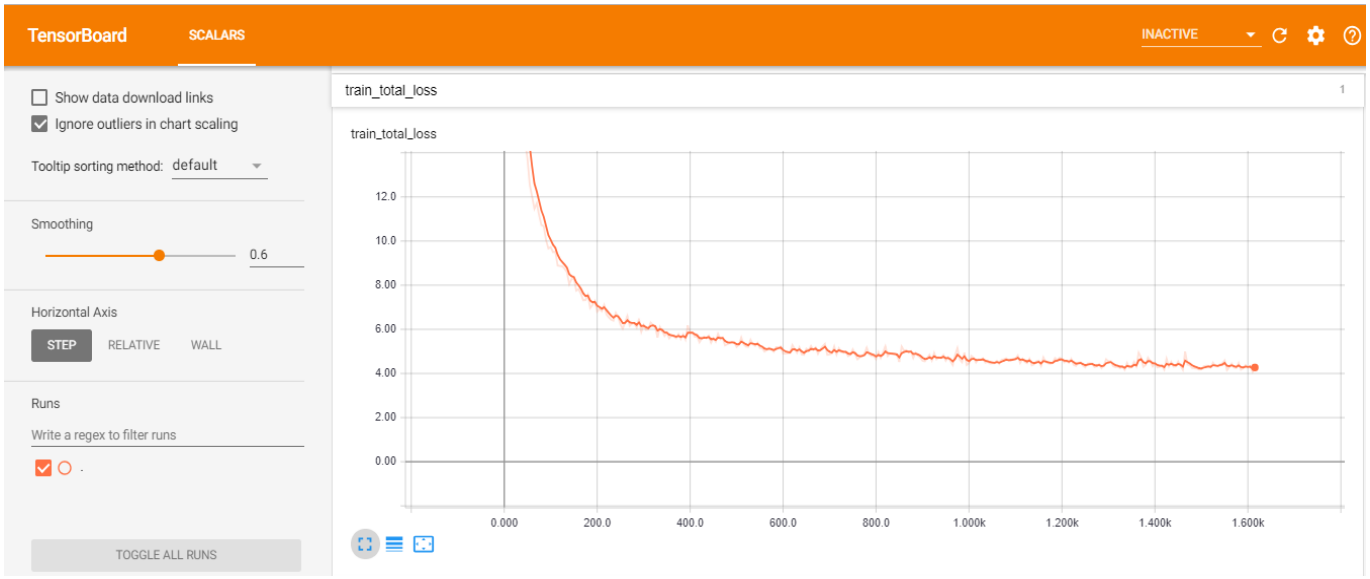

Figure 5: Train loss – Total

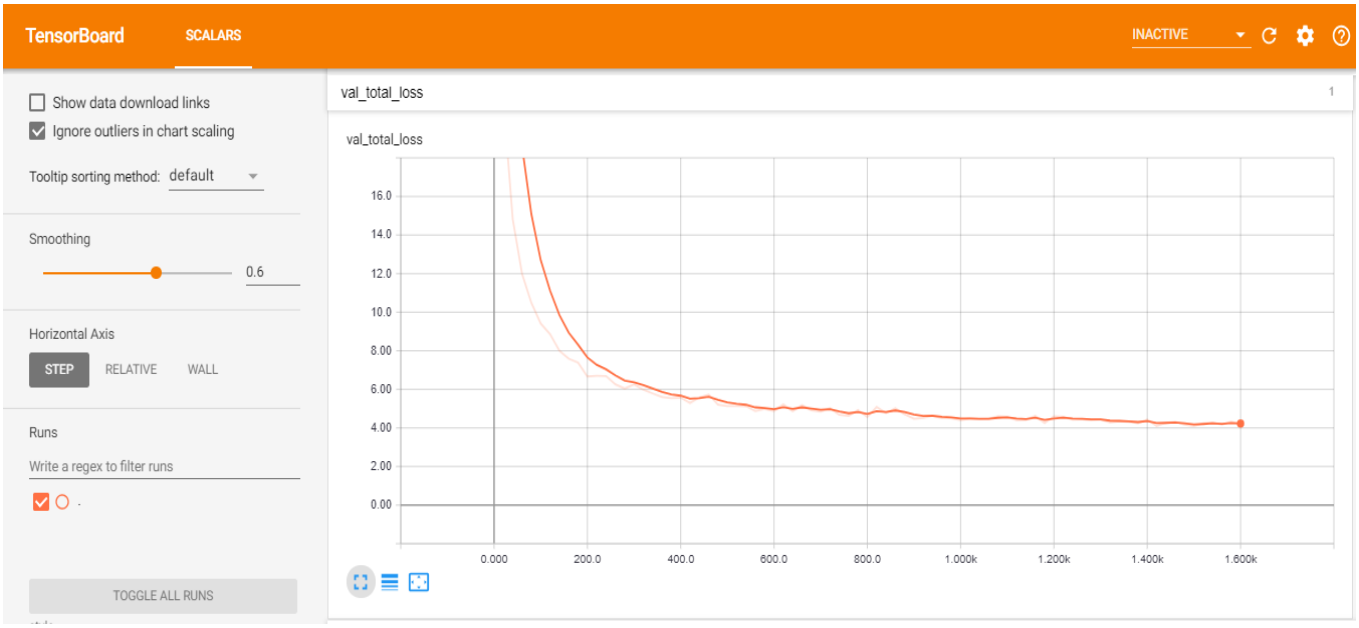

Figure 6: Validation loss - Total

3.4 Results

Figure 7 shows the results obtained with various input images for the mosaic style image for the baseline model.

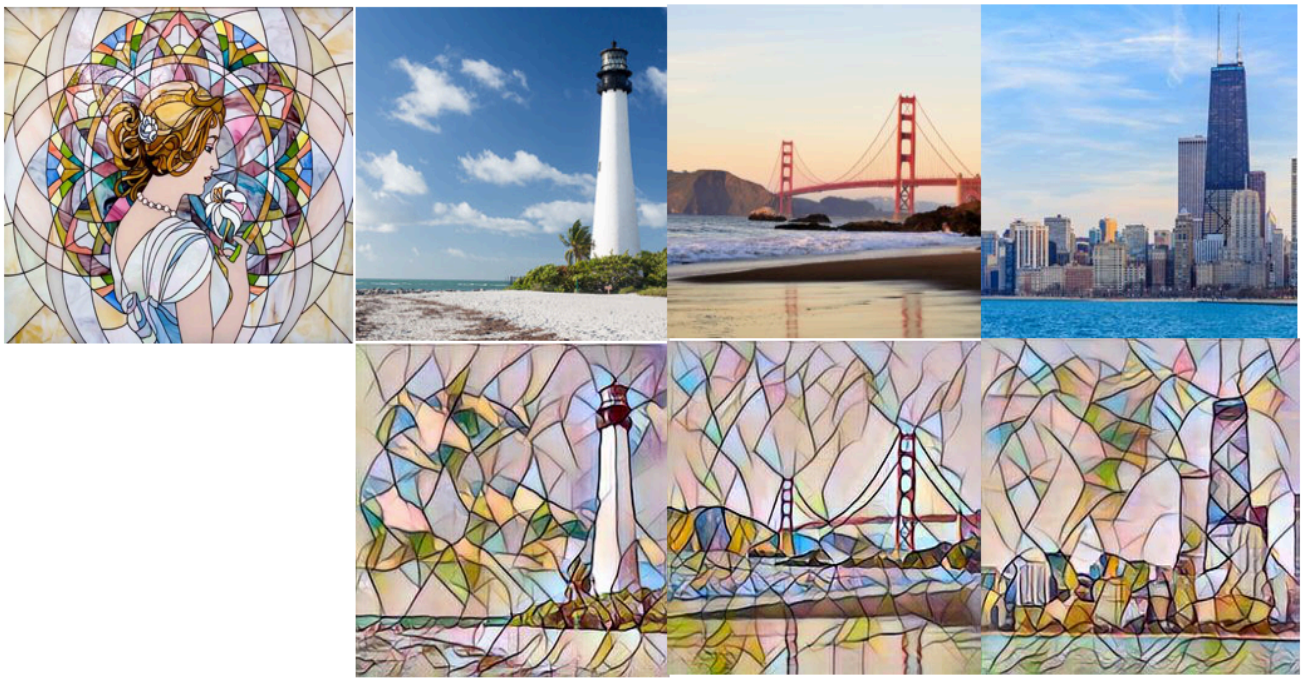

Figure 7: Baseline Results - First picture in the first row is the style image and the other images in the first row are the input content images. The bottom row showS the output stylized images.

3.5 Drawbacks

The implemented network requires to fix a style image before hand. This means that we need to train a new network for each new style image. This is a huge restriction of the network. In the second part of the project, I focused on removing this restriction from the network.

4. Improved Network

In the Part-II of the project, I focused on removing the constraint of a fixed style image from the network. Several changes were made to the network as well as the loss function in order to incorporate an arbitrary style image. Main motivation is taken from the instance normalization layer proposed in [4]. In the sections below, I explain the network architecture, the loss functions and the training procedure in detail.

4.1 Network

We can give the neural network an arbitrary style image and content image as input and can generate the desired stylized output. The proposed network has 3 main components: Encoder, Decoder and Adaptive Instance normalization. I have taken motivation from the network architecture of [6] for this.

The Encoder is used to extract the features from both the style and content images. Adaptive Instance Normalization is used to align channel wise mean and variance of content features to those of style features. Finally, the decoder is used to get the transformed features back to the image domain and thus generate the stylized image.

The Adaptive Instance Normalization layer is defined as follows. It takes input features of both content and style images, x and y respectively. It returns the mean and variance corrected content feature.

$$AdaIN(x,y) = \sigma(y)\left(\frac{x - \mu(x)}{\sigma(x)}\right) + \mu(y)$$

Let c and s be the content and style image respectively. We represent Encoder and Decoder with functions E and D respectively.

$$x = E(c)$$
$$y = E(s)$$
$$t = AdaIN(x,y)$$
$$output = D(t)$$

Figure 8 shows the overall flow of the network. Please note that, as the network is pretty huge, showing it in a single diagram is very difficult. Also, all the network details like kernel size, activation, etc. are also available in the individual tables.

4.1.1 Encoder Architecture

The encoder is fixed to the first few layers (until relu 4_1) of pre-trained VGG-19 [5]. Also, reflection padding was used in the encoder as mentioned in the paper. Table 3 shows the architecture of the encoder used.

4.1.2 Decoder Architecture

The decoder is mostly a mirroring of the encoder architecture. Nearest up-sampling is used. Table 3 shows the architecture of decoder.

LAYER	Activation Size	Kernel	Padding	Stride
Input	3 x 256 x256	-	-	-
Conv2d	3 x 256 x 256	(1,1)	0	1
Conv2d	64 x 256 x 256	(3,3)	1	1
ReLU_1	64 x 256 x 256	-	-	-
Conv2d	64 x 256 x 256	(3,3)	1	1
ReLU_1_2	64 x 256 x 256	-	-	-
MaxPool2d	64 x 128 x 128	(2,2)	0	2
Conv2d	128 x 128 x 128	(3,3)	1	1
ReLU_2_1	128 x 128 x 128	-	-	-
Conv2d	128 x 128 x 128	(3,3)	1	1
ReLU_2_2	128 x 128 x 128	-	-	-
MaxPool2d	128 X64 X 64	(2,2)	0	2
Conv2d	256 X 64 X 64	(3,3)	1	1
ReLU_3_1	256 X 64 X 64	-	-	-
Conv2d	256 X 64 X 64	(3,3)	1	1
ReLU_3_2	256 X 64 X 64	-	-	-
Conv2d	256 X 64 X 64	(3,3)	1	1
ReLU_3_3	256 X 64 X 64	(2,2)	0	2
Conv2d	256 X 64 X 64	(3,3)	1	1
ReLU_3_4	256 X 64 X 64	-	-	-
MaxPool2d	256 X 32 X32	(2,2)	0	2
Conv2d	512 X 32 X32	(3,3)	1	1
ReLU_4_1	512 X 32 X32	-	-	-
Output	512 x 32 x32			

Table 3: Architecture of Encoder

LAYER	Activation Size	Kernel	Padding	Scale
Input	256 x 32 x32			
Conv2d	256 x 32 x 32	(3,3)	1	-
ReLU	256 x 32 x 32	-	-	-
Upsample	256 x 64 x 64	-	-	2
Conv2d	256 x 64 x 64	(3,3)	1	-
ReLU	256 x 64 x 64	-	-	-
Conv2d	256 x 64 x 64	(3,3)	1	-
ReLU	256 x 64 x 64	-	-	-
Conv2d	256 x 64 x 64	(3,3)	1	-
ReLU	256 x 64 x 64	-	-	-
Conv2d	128 x 64 x 64	(3,3)	1	-
ReLU	128 x 64 x 64	-	-	-
Upsample	128 x 128 x128	-	-	2
Conv2d	128 x 128 x128	(3,3)	1	-
ReLU	128 x 128 x128	-	-	-
Conv2d	64 x 128 x 128	(3,3)	1	-
ReLU	64 x 128 x 128	-	-	-
Upsample	64 x 256 x 256	-	-	2
Conv2d	64 x 256 x 256	(3,3)	1	-
Conv2d	64 x 256 x256	(3,3)	1	-
ReLU	64 x 256 x 256	-	-	-
Conv2d	3 x 256 x 256	(3,3)	1	-
Output	3 x 256 x 256			

Table 4: Architecture of Decoder

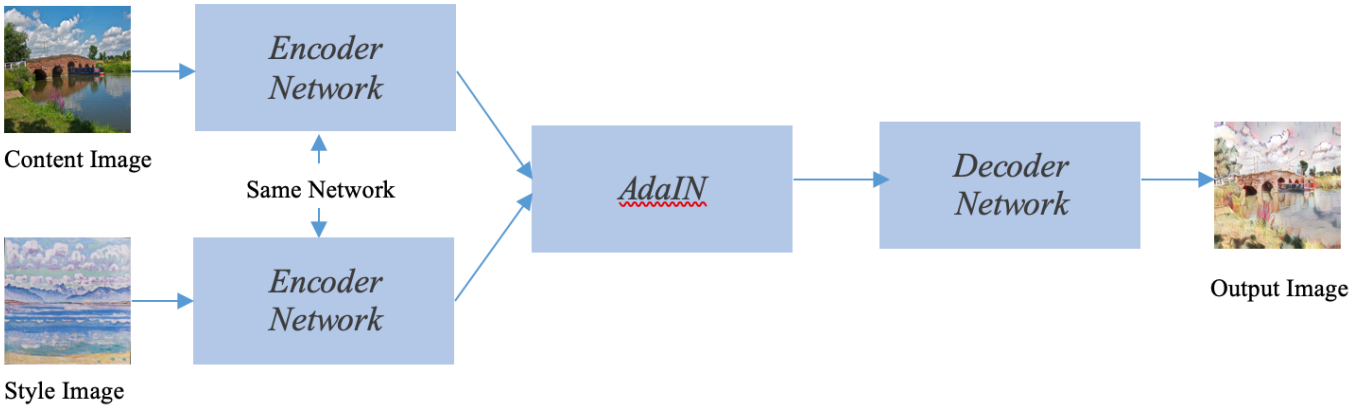

Figure 8: Overall flow of the networ

4.2 Loss

The aim would be to obtain an image which has content from the content image while have the style of style image. The loss term is a sum of both content loss and style loss which is defined as follows.

$$L = L_c + \lambda L_s$$

$$L_c = \| E(D(c)) - t \|_2$$

$$L_s = \sum_{i=1}^{L} \| \mu(\phi_i(D(t)) - \mu(\phi_i(s)) \|_2 + \sum_{i=1}^{L} \| \sigma(\phi_i(D(t)) - \sigma(\phi_i(s)) \|_2$$

Where Φ_l denotes a layer in VGG-19 used to compute style loss. These layers correspond to relu_1_1, relu_2_1, relu_3_1 and relu_4_1 shown in Table 2.

4.3 Training Details

The model was trained for 150K iterations using the Adam optimizer. The model is coded in Pytorch and trained from scratch. Table 5 shows the hyper-parameters tried and the final parameters selected. I used tensorboard to plot and keep account of the train and validation loss. Figure 9 shows the train loss with increasing iterations and Figure 10 shows the same for validation. Train Loss was saved every 100 iteration and Validation Loss was saved every 500 iterations. Also, the model checkpoints were saved every 10000 iterations. Note that the train loss is more fluctuations than the validation loss. This is because the train loss is recorded at a different batch every time, while the validation loss is recorded for the same set of images every time.

Parameter	*Range*	*Final*
batch size	4 to 8	8
learning rate	5e-5 to 2e-5	1.00E-05
lambda	6 to 10	7

Table 5: Hyper Parameter Search

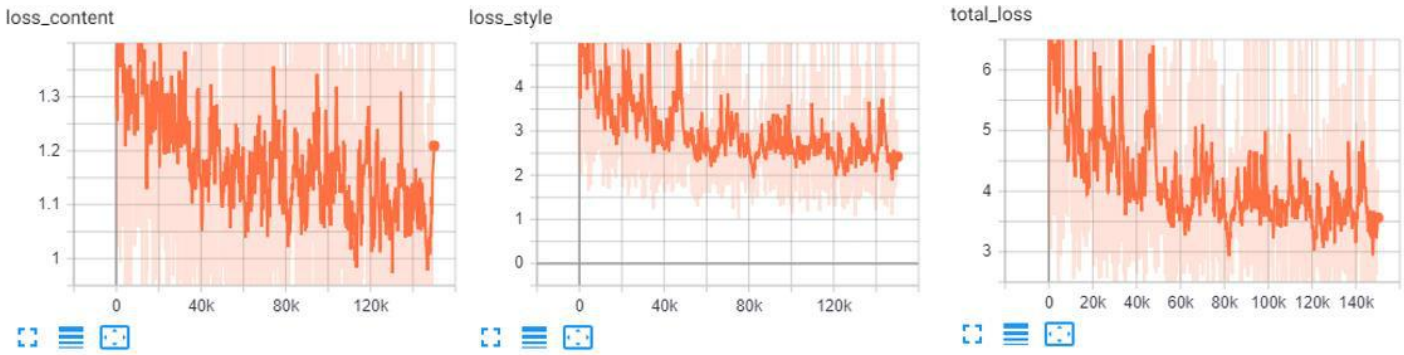

Figure 9: Train Loss – Content, Style and Total

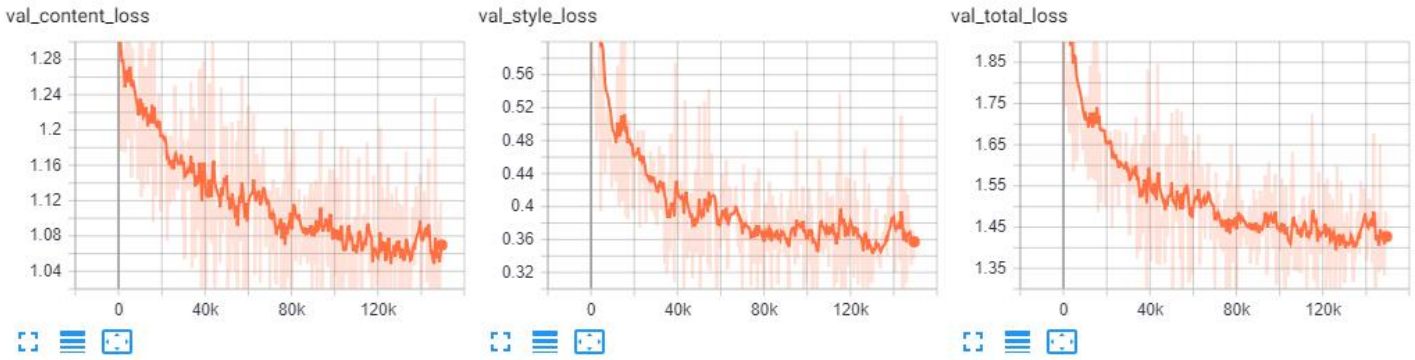

Figure 10: Validation Loss – Content, Style and Total

4.6 RESULTS

Figure 11 shows the output of the network with different content and style images.

Figure 9: The images in the top row are the style images and the images in the leftmost column are the content images

Also, as asked in the feedback, I ran the style transfer network for style images by 'Shiori Matsumota'. Figure 10 shows the result of image stylization with Shiori Matsumoto's painting as the style image.

Figure 10: Output images when a painting by Shiori Matsumoto (bottom left)is taken as style image. All images in the first row are the content images and the images bottom row are the styled images.

5 Instructions to run inference and GUI

5.1 Install Dependencies

- Pytorch 1.0.1
- TensorboardX 1.6
- tqdm 4.28.1
- PIL 5.3.0
- torchvision 0.2.2
- Tkinter

5.2 Execution: Need to download the model file from the link given in github. Please follow github for detailed description on running the code.

- GUI : python gui.py
- Inference : python inference.py –content_path * --style_path *
- Train : python train.py

5.3 Code

GitHub Link: https://github.tamu.edu/stuti/DeepStyleTransfer

Note - More details on running the code, folder structure and data is available in the readme file.

5.4 Video Demo

Link to Video Demo: https://www.youtube.com/watch?v=gdYi0k3Yp8g&feature=youtu.be

5.5 References

1. J. Johnson, A. Alahi, and L. Fei-Fei, “Perceptual losses for realtime style transfer and super-resolution,” in European Conference on Computer Vision, 2016, pp. 694–711.
2. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
3. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L.: ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV) 115(3) (2015) 211–252
4. D. Ulyanov, A. Vedaldi, and V. Lempitsky. Instance normalization: The missing ingredient for fast stylization. arXiv preprint arXiv:1607.08022, 2016. 4
5. K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In ICLR, 2015. 4, 5
6. Huang, X., Belongie, S.: Arbitrary style transfer in real-time with adaptive instance normalization. In: ICCV. (2017)
7. https://github.com/naoto0804/pytorch-AdaIN
8. https://github.com/pytorch/examples/tree/master/fast_neural_style