Neural Network Project Demo

April 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 was used for content images. It has around 83K training images and 41K training images. 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.

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

3. Network

I have implemented the method proposed in [1]. Here, 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.

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.

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

3.1 Encoder Architecture

The encoder is fixed to the first few layers (until relu 4_1) of pre-trained VGG-19 [2]. Also, reflection padding was used in the encoder as mentioned in the paper. Table 1 shows the architecture of the encoder used.

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 1: 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 2: Architecture of Decoder

3.2 Decoder Architecture

The decoder is mostly a mirroring of the encoder architecture. Nearest upsampling is used. Table 2 shows the architecture of decoder.

4. 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.

$$L = L_c + \lambda L_s$$

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

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

Where ϕ_i 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 1.

5. Training Details

The model was trained for 150K iterations using the Adam optimizer. The model is coded in Pytorch and trained from scratch. Table 3 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 1 shows the train loss with increasing iterations and Figure 2 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. 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 3: Hyper Parameter Search

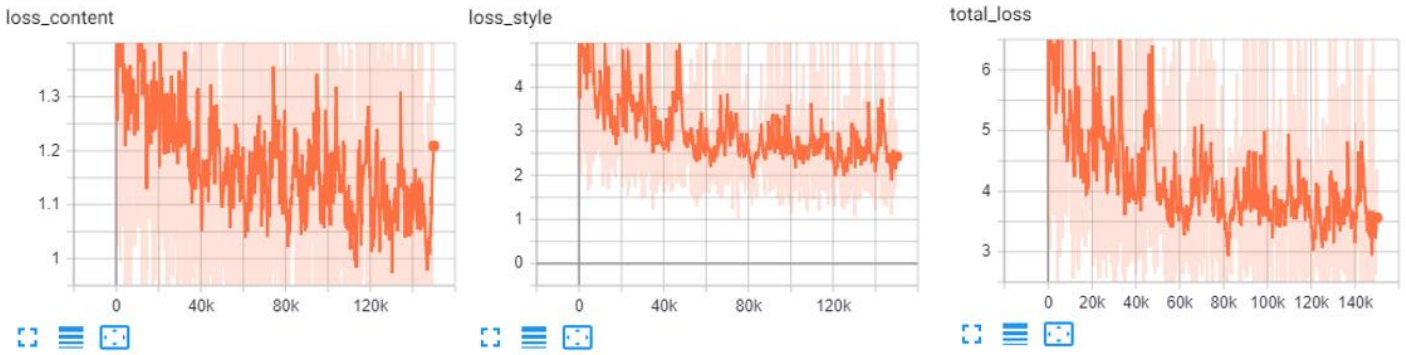

Figure 1: Train Loss – content, style and total

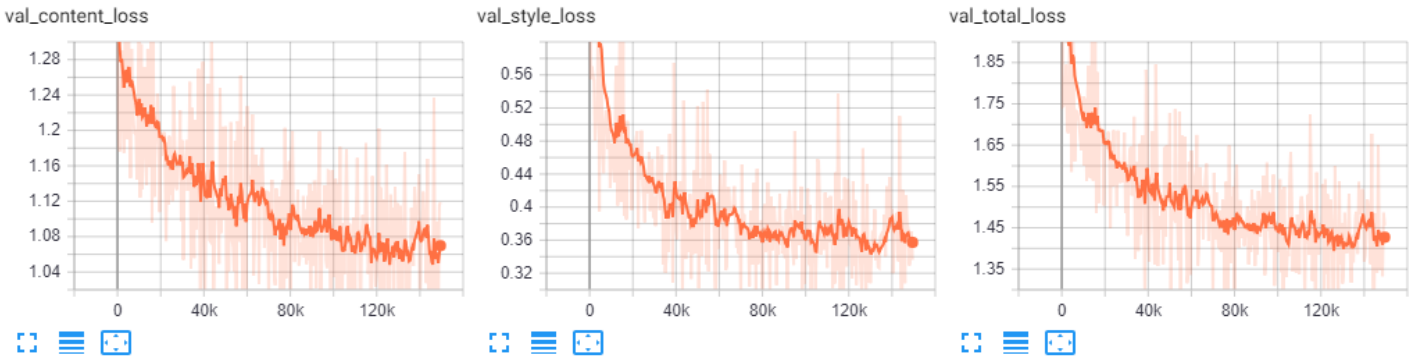

Figure 2: Validation Loss – content, style and total

6. RESULTS

I will present the results in two ways here. Figure 3 shows the output of the network with different content and style images. Figure 4 shows the comparision of the output generated by the model I have trained and the official implementation of the paper. The official code and model for the paper is available in torch. I converted their torch model to pytorch for the comparision. When compared to the official model, the results generated by my model seem to have a certain loss in quality. However, overall, the results are comparable.

Figure 3: The images in the top row are the style images and the images in the leftmost column are the content images

Figure 4: Comparison - the left amongst the style images are the official model's results and the one on the right are the results generated by the model I have trained

7. Instructions to run inference and GUI

7.1 Install Dependencies

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

7.2 Execution

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

7.3 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.

7.4 Video Demo

Link to Video Demo :

8. References

1. Huang, X., Belongie, S.: Arbitrary style transfer in real-time with adaptive instance normalization. In: ICCV. (2017)
2. K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In ICLR, 2015. 4, 5
3. https://github.com/naoto0804/pytorch-AdaIN