Neural Image Style Transfer in Real Time

Rami Matar
Computer Science
Columbia University
rhm2142@columbia.edu

I. Introduction

Neural (Image) Style Transfer (NST) is the interesting task of imposing the style of one image to the content of another image, producing a totally new and original image. NST is fascinating for multiple reasons, one of which is its artistic capabilities which allow it to mimic painting style, create non-photorealistic photo and video such as animation, and much more; another is it being one of the first extremely powerful examples of learned creativity from computers.

Style transfer had previously been a subject of research interest prior to the popularity of convolutional neural networks (CNNs) and deep learning [11]. Non-neural network methods faced a great number of problems; with limitations to the types of style transfers, those algorithms could not find more general approaches to style transfer. As CNNs gained popularity after 2012, following papers such as "Visualizing and Understanding Convolutional Networks" by Zeiler and Fergus and "Texture synthesis using convolutional neural networks" by Gatys, et. al. [8][11][10], it was understood that convolutional networks convey a lot of visual information in ways that agree with our human perception and understanding of vision. Careful analysis in both papers showed that the learned filters and effects of the depth of the network on the receptive field allow the network to learn to identify very complex shapes and objects.

The first paper to introduce NST, in 2015, was "A Neural Algorithm of Artistic Style" by Leon A. Gatys, et. al [1]. The paper proposed a neural-network based optimization approach that guides the input content image to a stylized image using a weighted combination of a content and a style loss function to backpropagate the pixel values of the input image. This approach used a pretrained convolutional neural network, VGG [4], in order to compute the perceptual features of the image. Those perceptual features are deterministically computable since the model is pretrained, and they represent the content and style of the image. The perceptual features are a sort of latent space to which we will constrict our handling of the images, since NST requires us to step away from direct pixel values and to consider patterns and perceptual features overall. This approach was extremely versatile and powerful and worked for any two arbitrary images. The only problem, a big one at that, is that was devastatingly slow, taking minutes to generate even one image.

After the seminal paper by Gatys, et. al., NST was birthed and naturally gained a lot of attention. It was exciting to see the potential of CNNs and their latent space representation of images, but it was very apparent that the applications of this approach were very limited due to computation. The next papers that followed fixed that problem by reframing the problem as a feedforward trainable network, rather than an optimization process [5][9][7][2]. In this approach, a decoder learns to decode a combination of the features of both the content and the style images into an image that minimizes the combined perceptual loss. That loss was also measured using the same pretrained CNN. This approach achieved NST in real-time with very promising results, yet the models were typically capable of stylizing a finite number of styles on which they were trained, and would not generalize well [2]. Moreover, there was at least a linear cost associated with supporting more costs in the network, making it unfavorable for training with very big datasets, and for generalizing to real-world examples [2].

An extremely important finding was that of the importance of instance normalization in the style transfer task. In 2016, Ulyanov, et. al. showed in "Instance Normalization: The Missing Ingredient for Fast Stylization" that instance normalization layers allow the feedforward decoder to learn to transfer style features much better than batch normalization layers [9]. This observation was surprising yet further research was able to help connect the dots on why instance normalization turns out to be much more effective [2]. Instance normalization allows the network to isolate the individual stylistic elements of each image, and it also turns out that most of the information on an images style is contained within the statistics of the individual channels [2]. It turns out that instance normalization was indeed the missing element in fast style transfer!

In 2017, "Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization" (AdaIN) by Huang and Belongie showed an implementation that can generalize to arbitrary inputs in real-time [2]. The authors proposed to swap the learnable layer that was responsible for learning to process the perceptual features to be input to the decoder with a non-learnable instance normalization layer where the perceptual features of the content feature are normalized around the statistics of the style features. This on its own allowed for a model capable of learning to generate stylized images using any content image and any style image. Importantly, this approach maintains the real-time speed of feedforward models, produces high quality results, and generalizes to any style! In this project, I implemented the proposed model in AdaIN to generate arbitrary image style transfers in real-time.

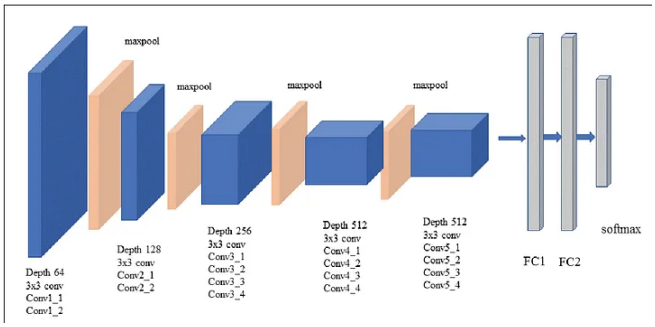

Fig. 1. The VGG-19 model [4]. In this implementation, we use the encoder up to the relu activation after Conv4-2 layer. We also extract one of the relu activations from each convolution block (blue) to gather features of different levels of complexity and to use in our loss computation

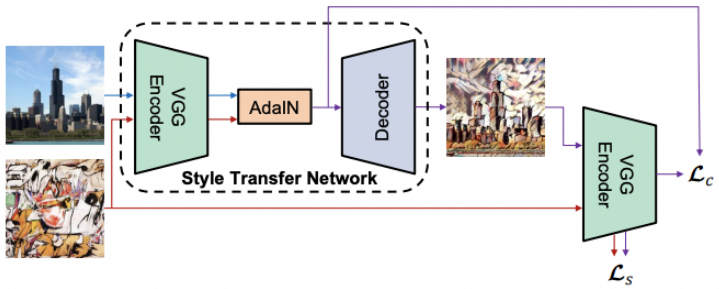

Fig. 2. The architecture proposed by Huang, et. al. consists of a VGG encoder that calculates perceptual features in its first few layers followed by the AdaIN layer to perform style transfer in the latent space. Then, a decoder is trained to turn that representation into a stylized output image, which gets fed through the VGG encoder to calculate its features and compute the loss.

II. Architecture

The AdaIN model architecture consists of three components [2]:

(1) pretrained VGG encoder (shown in Fig. 1), which we use to calculate the perceptual features and losses [4]. Initially, the model receives the content image and style image as input; it will evaluate the perceptual features of both images using the pretrained encoder. The encoder was introduced in the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition" by Simonyan and Zisserman and is a standard network of convolution layers followed by maxpooling and relu layers. In the forward pass, we store the output of specific layers as our perceptual features. These are simply a latent space representation of our image that carries perceptual information. It is possible to experiment with many different choices for the output layers, but generally, most implementations choose one relu layer from each of the first 4 convolution blocks in the VGG encoder.

(2) The Adaptive Instance Normalization layer is a simple function:

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

where x and y represent the content and style images respectively, and the first order statistics μ, σ are calculated channel-wise [2]. The AdaIN layer solves all of the problems with the initial implementations of instance normalization in NST. It both works for any arbitrary x, y and completely removes the need to learn any additional parameters. The output of the AdaIN layers goes through (3) the decoder, which is the only learnable part in this architecture. The decoder is used to predict the stylized image and follows an inverted shape of the VGG encoder. My implementation replaces the pooling layers with upsampling and reflection pad layers.

III. Training

A. Loss Function

We calculate individually a style loss and a content loss for our stylized output image. The content loss is calculated as the mean squared error (MSE) the AdaIN layer output and the perceptual features of the stylized image. The style loss is similary a MSE loss, but applied between the first order statistics of the style image and the stylized output image in each of the four intermediate layers [2].

Let $f_i(x)$ denote our VGG encoder's output for input x at the ith output layer we chose to obtain our perceptual features, t be the AdaIN output, and let g be the decoder we're training. Then the content loss using the 4th (last) encoder layer activation is calculated as

$$L_c = ||t - f_4(g(t)||_2$$

and the style loss is calculated as

$$L_s = \sum_{i=1}^{4} ||\mu(f_i(g(t))) - \mu(f_i(y))||_2 + \sum_{i=1}^{4} ||\sigma(f_i(g(t))) - \sigma(f_i(y))||_2$$

Then we write the general loss function

$$L = L_s + \lambda L_c$$

with λ as a hyperparameter for the network which controls the intensity of style preservation relative to content preservation.

B. Data

To train this model, I used the MSCOCO dataset [12] and a small music album covers dataset for my content image datasets, and the WikiArt dataset [13] for style images. During training, each epoch saw 1 example of each content image in those datasets paired with a random style image from the WikiArt dataset. I highly recommend WikiArt for training models capable of artistic image transformations.

C. Hyperparameters

For the loss calculation, $\lambda = 2.5$ was chosen. Training was done using an Adam optimizer with an initial learning rate, r_i of 1e-4; the authors further implement learning rate decay, so the learning rate is calculated as

$$r = \frac{r_i}{1 + d * t}$$

where d is the learning rate decay parameter and t is the number of iterations of backpropagation elapsed so far. The

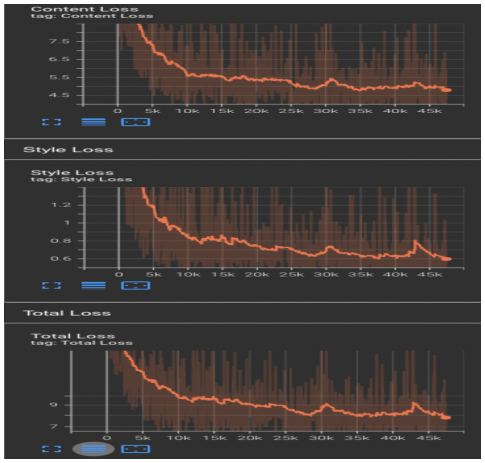

Fig. 3. Observe the content and style loss both going down but exhibiting the high variance of our data.

authors chose a learning rate decay of 5e-5 to help the model make smaller changes and converge more precisely after the initial stages of training, although this is not an extreme decay by any means. I ran my training on 2 RTX 4090s with a batch size of 8 until the model had seen approximately 700,000 images.

D. Convergence

During training, we observe the content and style loss to ensure that our model is behaving well and to further understand our results. The style loss and content loss both go down considerably (Fig 3.), but the loss still is a high enough value (I did not apply scaling so the style loss for example is a MSE of first order statistics between the different channels), leading me to believe that for any content and style image pair, there is a natural limit to the amount of content and style that we can conserve while representing both. Because of this, it makes sense that while a very smoothed version of the loss function is clearly showing that the model is learning, the natural graph looks much more noisy and arbitrary.

E. Observing The Training Process

I used Tensorboard to log images and torchvision to save image grids, so I will share some of the examples observed at different points during training. The examples shown are after approximately 1,000 (Fig. 4), 10,000 (Fig.5), 100,000 (Fig. 6), 250,000 (Fig. 7) images being observed by the model. Each figure is a grid of the content image (left), style image(middle), stylized output image (right).

IV. Technical Issues and Implementation Details

The entire project code is available on https://github.com/RamiMatar/AdaIN_NST.git, as well as a walkthrough notebook and a pretrained model. I will use this section to discuss some things I had to implement or learn for this project and give a brief summary and potentially useful code snippets. My code was written using PyTorch

Fig. 4. After 500 examples

Fig. 5. After 5000 examples

Fig. 6. After 50,000 examples

Fig. 7. After 100,000 examples seen

```
def load_ddp_checkpoint(self, PATH, from_DDP = False):
    if torch.cuda.is_available():
        state_dict = torch.load(PATH)['model']
    else:
        state_dict = torch.load(PATH, map_location = torch.device('cpu'))['model']
    # This is essential if we save the model during DDP training and want to do inference on a single GPU or CPU.
    if from_DDP:
        new_state_dict = OrderedDict()
        for k, v in state_dict.items():
            name = k[7:] # remove 'module.' of DataParallel/DistributedDataParallel
            new_state_dict[name] = v
        state_dict = new_state_dict
    self.model.load_state_dict(new_state_dict)
```

Fig. 8. Code block to enable using DDP model for inference, essentially updates the weight names to be suitable for a non DDP-wrapped instance.

[14] and utilized DDP for multi-GPU training, torchvision, tensorboard, and some other utilities.

A. Preprocessing and Postprocessing

After initially setting up the model, it was not capable of learning until I added image preprocessing and postprocessing to work with the pretrained VGG encoder I used. Otherwise, the encoder will not produce sensible results. To fix this, I used the following suggested VGG normalization transform: torchvision.transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) Importantly, we also have to denormalize the output based on the same values when we receive an output image. It was also important to make sure we convert all the images to the same format; one problem I ran into was with images that included an alpha channel, which would cause my model to crash in training, so it was important to ensure everything is represented in RGB to ensure it was input to the CNN encoder properly.

B. Multi-GPU Training

In order to train the model on multiple GPUs, we use the DDP wrapper class in PyTorch to wrap our model and prepare it for training. We initialize the training with a Pytorch wrapper of the python multiprocessing library. After the DDP processes are running and training, we might need to load from a checkpoint. I didn't face any literal errors when loading the model, but when training, the models on the non-parent GPU process typically produced nan outputs, and it was only fixed when I added this line: torch.load(PATH, map_location = "cuda:".format(self.rank)) when loading the model to DDP after being trained using DDP. self.rank here is the value given to the specific GPU by the multiprocessing library in PyTorch. Moreover, I needed to handle the option of loading the model on CPU or on a single GPU not using DDP for inference. To handle that, I had to change my checkpoint loading to that shown in Fig 3.

C. Interpolation Across Styles

After our model is trained, it is very simple to use it to implement a style transfer with a certain level of intensity by combining the output of the AdaIN layer and the output of the encoder on the content as such:

$$t(c, s, \alpha) = \alpha AdaIN(f(c), f(s)) + (1 - \alpha) f(c)$$

It is also possible to apply the same principle to interpolate a weighted combination of styles.

Fig. 9.

Fig. 10.

D. Results

In Fig. 9-12, we can see results from the model on unseen images during training. The results are promising and relatively high quality. Generally speaking, most results are of at least good quality, however, the results when the style image was monochromatic did not end up following the style closely, and still included various layers of colors. An example of this can be seen in Fig. 9, 13. It is possible to address issues like this by increasing the style loss coefficient during

Fig. 11.

Fig. 12.

Fig. 13.

training and allowing flexibility during inference by setting the α parameter discussed above. Other approaches can try to ensure color preservation from the style or content image in future works.

V. Conclusion

In this implementation, I was able to closely follow the paper by Huang and Belongie and reproduce similarly high-quality stylized images as their study. The potential of AdaIN layer did not stop after this paper, but saw many applications for stylistic and conditional generative models, such as StyleGAN, and remains a very useful tool for similar tasks. After this project, it is my goal to explore further the possibilities with stylistic generation, such as video rendering, and how that may be improved with a different loss function that accounts for temporal loss using this approach. It is also possible to pair this with other powerful models in computer vision to produce incredible results, such as segmentation models to produce animated backgrounds or multiple styles across the image.

References

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

[2] Huang, Xun, and Serge Belongie. "Arbitrary style transfer in real-time with adaptive instance normalization." Proceedings of the IEEE international conference on computer vision. 2017.

[3] Dosovitskiy, Alexey, and Thomas Brox. "Inverting visual representations with convolutional networks." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[4] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[5] Johnson, Justin, Alexandre Alahi, and Li Fei-Fei. "Perceptual losses for real-time style transfer and super-resolution." Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part II 14. Springer International Publishing, 2016.

[6] Karras, Tero, Samuli Laine, and Timo Aila. "A style-based generator architecture for generative adversarial networks." Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2019.

[7] Ulyanov, Dmitry, et al. "Texture networks: Feed-forward synthesis of textures and stylized images." arXiv preprint arXiv:1603.03417 (2016).

[8] Gatys, Leon, Alexander S. Ecker, and Matthias Bethge. "Texture synthesis using convolutional neural networks." Advances in neural information processing systems 28 (2015).

[9] Ulyanov, Dmitry, Andrea Vedaldi, and Victor Lempitsky. "Instance normalization: The missing ingredient for fast stylization." arXiv preprint arXiv:1607.08022 (2016).

[10] Jing, Yongcheng, et al. "Neural style transfer: A review." IEEE transactions on visualization and computer graphics 26.11 (2019): 3365-3385.

[11] Zeiler, Matthew D., and Rob Fergus. "Visualizing and understanding convolutional networks." Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13. Springer International Publishing, 2014.

[12] Tsung-Yi Lin, Maire, M., Belongie, S. J., Bourdev, L. D., Girshick, R. B., Hays, J., ... Zitnick, C. L. (2014). Microsoft COCO: Common Objects in Context. CoRR, abs/1405.0312. Retrieved from http://arxiv.org/abs/1405.0312

[13] WikiArt: "WikiArt: A Large-Scale Dataset for Artistic Content Analysis." Kaggle, 2018, https://www.kaggle.com/c/painter-by-numbers/data.

[14] Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." Advances in neural information processing systems 32 (2019).