A Comparison Study on Two Neural Style Transfer Methods

Denglin Jiang, Mengyang Zhang

New York University

dj1369@nyu.edu, mz2325@nyu.edu

1. Introduction

Neural style transfer refers to algorithms that manipulate the original images and render a brand-new appearance with a different style. People have grown interested in this topic because it has wide applications in fields like transfer learning and domain adaptation.

In this project, we plan to do a comparative study on neural style transfer techniques. We applied the knowledge we learned throughout the semester: 1) Applied CNN-based neural networks, such as CNN and CycleGAN. 2) Utilizing cloud services for deep learning model training. 3) Deploy the trained model on IBM Cloud using Kubernetes services. 4) Evaluating the ML systems using the performance factors covered in class. 5) Comparing two different neural style transfer methods.

2. Related Work

In this section, we will discuss some of the popular neural style transfer methods. Among them, we studied the top 2 most popular methods and gave a short introduction about them in the following subsections.

2.1. Neural Style Transfer Methods

Before the booming of deep neural networks, people use a class of algorithms called image analogy [1] for style transfer, which focus on lower level image features like edges, and these algorithms often fail to capture image structures effectively. [2] As hardware and computing power develop, people then switch to neural networks for style transfer. [2] categorized neural style transfer methods into basically two categories: image-optimization-based and model-optimization-based.

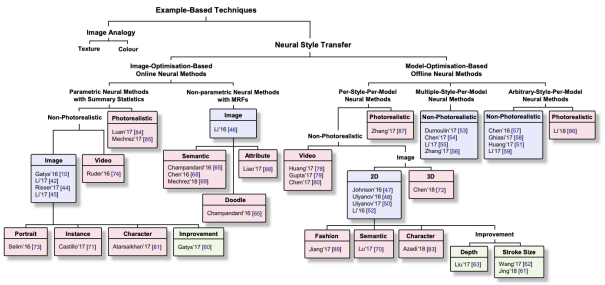

Figure 1: *A taxonomy of NST techniques [2]*

2.2. A Neural Algorithm of Artistic Style by Gatys al.

To transfer the style of an artwork onto a photo, Gatys al. [3] synthesised a new image that simultaneously matches the content representation of the photo and the style representation of the artwork.

To separately capture the content and style representations from an image, they used a generic representation learned by high-performance VGG16 [4] to modify an white-noise image I into the desired output (artistic style added to the phote) through back-propagation. During backpropagration, they updated I using a specially designed loss function, which is a linear combination of the content loss and the style loss:

$$\mathcal{L}_{total} = \alpha\mathcal{L}_{content} + \beta\mathcal{L}_{style}$$

$$\mathcal{L}_{content} = \frac{1}{2}\sum_{l=0}^{L}(F^l(X) - F(I)^l)^2$$

$$\mathcal{L}_{style} = \frac{1}{constant}\sum_{l=0}^{L}(G(F(X) - G(F(I)))^2$$

where X represents the photo, I represents the white noise image, l represents each layer of VGG16 (there are altogether L layers), F represents the feature mapping, G represents the gram-matirx, and α and β control the ratio beween content and style. The update rule could be written as

$$I_t = I_{t-1} - \eta\frac{\partial\mathcal{L}_{total}}{\partial I_{t-1}}$$

This way, they were able to independently process and manipulate the content and style of natural images.

2.3. Cycle-GAN

Generative adversarial network(GAN) [5] refers to a class of unsupervised machine learning algorithms that used to synthesize new data from learning the patterns from old real data. It is said to be the most exciting idea in the last 10 years in the field of machine learning according to Yann Lecun. It is now widly used in Computer Vision, Natural Language Processing and many more fields. Its applications include improving image and video quality, data synthesis, domain adaptation and etc.

It is basically made up of two components: a generator and a discriminator. The generator is an Auto-encoder used to generator new data points and the discrtiminator is used to judge how 'real' this new data point is. In other words, discriminator is basically a birnary calssifier, whose two classes are 'real' and 'fake'. We will keep training both the generator and the discrtiminator until the discrimintator cannot tell real data points from fake, or should we say synthetic, data points. Intuitively, instead of hand-engineer the loss function, say in Auto-encoders the reconstruction loss (Mean Square Error), we use the discriminator of GAN to learn real and fake patterns and then automatically measure the distance between fake and real data. To train both generator and discriminator, we have a standard GAN loss function, min-max loss:

$$\mathcal{L}_{gan}(G) = E_x(\log D(x) + E_z(\log 1 - D(G(z))$$

where x represents real data, z represents the latent variable used to create fake synthetic data, G represents the Generator, and D represents the discriminator. During training, the generator tries to minimize the loss while discriminator tries to maximize the loss. To avoid generator saturation, people add log to the discriminator. In practice, it is challenging to train GANs, and the choices of hyper-parameters may easily affect GAN convergence. [5].

A number of variations have been proposed to solve these challenges. Wasserstein Generative Adversarial Network (WGAN) [6], which swtich the sigmoid functions for linear functions for activation layers, is proposed to address the gradient vanishing and mode collapse issues for vanilla GAN. Conditional Generative Adversarial Network (CGAN) is designed to generate synthetic data points associated with a particular type of descriptive labels. [7] We also have Cycle GAN [8] designed for image-to-image translation.

Cycle GAN aims to learn the mappings from source domain to target domain using a set of aligned image pairs. It adopts an idea of cycle consistency, which couples the original mapping from source domain X to target domain Y with an inverse mapping from target domain Y to source domain X. Therefore, compared with standard GAN loss, Cycle GAN loss has an additional GAN loss and a cycle consistency loss:

$$\mathcal{L}_{cycgan}\left(G,F\right)=\mathcal{L}_{gan}(G)+\mathcal{L}_{gan}(F)+\mathcal{L}_{cyc}(F,G)$$

where G represents the generator that maps images from source to target domain, and F represents the generator that maps images back from target to source domain. Cycle consistency loss could be written into:

$$\mathcal{L}_{cyc}(F,G)=\mathbb{E}\left[||F\left(G\left(x\right)\right)-x||_1\right]+\mathbb{E}\left[||G\left(F\left(y\right)\right)-y||_1\right]$$

where we use L1 distance to measure the distance between synthetic data and real data.

With this design, Cycle GAN allows translation of images without training data sets, while other GANs need to manually convert horse images to zebra images and use these aligned pairs as training data. Using Cycle GAN, we could translate photos of zebras to horses using only a collections of horse and zebra images and back again.

3. Experiment

In this section, we will explain how we did the experiments. We have conducted training and inference separately using 3 different environment set up. For training, we used NVIDIA Tesla V100 and NVIDIA Tesla P100. For inference, apart from using the same GPUs, we also built an app using Python Flask, containerized it and deployed it on IBM Cloud using Kubernetes services.

3.1. Data

For NST, we used 6 pairs of colored images (1 content and 1 style) cropped into size 512 by 512.
For Cycle GAN, we used toy data 'horse2zebra' by the CycleGAN author, which is made up of 939 horse images and 1177 zebra images of size 256 by 256 downloaded from ImageNet [9] using keywords wild horse and zebra.

3.2. Environment

We set up 3 different system for training and inference: 1) Tesla V100 on Google Colab; 2) P100-CIE from Google Colab; and 3) a CPU from IBM Cloud.

3.3. Parameters Setup

For NST, we set content weight $\alpha = 1000000$, style weight $\beta = 1$, normalized mean $= (0.485, 0.456, 0.406)$, normalized std $= (0.229, 0.224, 0.225)$, and we used a LBFGS [10] optimizer with default parameters from Pytorch. During NST process (inference), we did not start with a white noise image. Instead we initialized with the content image, and then updated the weights for 300 epochs to get satisfactory results.
For Cycle GAN, learning rate is 0.0002, and β1=0.5. No dropout. Batch size is set to 1. We cropped and normalized images and set training epochs to 100.

3.4. Training

The average training time for both algorithms are as follows.

Algorithm	NST by Gatys al.	Cycle GAN
Tesla V100	0	402 s/epoch
Tesla P100	0	408 s/epoch
CPU	0	-

To get satisfying results for CycleGAN, we trained for 200 epochs and it almost takes a total of 22 hours, which is every long period of time, so we did not proceed to experiment with CPU for CycleGAN.

3.5. Inference

For NST by Gatys al., we get the following results using a customized input image.

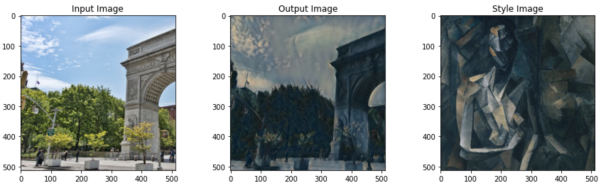

Figure 2: *A sample output of NST algorithm by Gatys al.*

For CycleGAN, we successfully reproduce the results from the original paper.
Then we compare the inference time for a single image for both algorithms:

Algorithm	NST by Gatys al.	Cycle GAN
Tesla V100 (s)	9.05	7.31
Tesla P100 (s)	14.73	6.98
CPU (s)	1295.08	-

3.6. Deployment

We later deployed the trained models on IBM cloud using Kubernetes service. There are two ways to deploy our model.
1) The first way is to follow the instructions in the second homework to create an object storage and a Watson machine learning service on IBM where we just import our notebooks.
2) The other way is to convert the notebook to a python file, create a docker image and deploy it onto IBM cloud using kubernetes.

In the actual implementation, we took the second approach. We realized that while we could import the Colab notebook into IBM Watson, we could directly run it as the notebook utilizes some libraries provided by Google Cloud. Secondly, we had to

add additional codes to connect the IBM Object storage with our machine learning service. After many tries, we failed to get it working. As a result, we switched to the second approach to deploy our model.

This is how we did it. We downloaded the Colab notebook and converted it into a python file neural_style_transfer.py. In the file, we wrapped the demo code within a flask app that runs on port 8002. When the user visits the corresponding IP address, the demo code would be triggered to start running. In the end, it would return the executation time as the result. After we finializing the python file, we wrote a dockerfile to let us to create a docker image. Based on this, we then utilized the kubectl tools to create a kubernetes deployment called proj2. The next step is to expose our app's port 8002 and to create a NodePort type of service, which is currently running on IBM cloud at http://192.168.0.135:8002. Because we were only able to request 2 CPU and 4GB memory for our IBM cluster for free, the response time of our training service is quite long. If it was on GPU, the service would return the result immediately.

4. Discussion

In this section, we compared these two methods, Neural Style Transfer Algorithm by Gatys al. (referred to as NST in later sections) and Cycle GAN algorithms, using several metrics we learned throughout the semester:

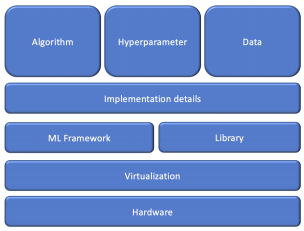

Figure 3: *Performance Measures for ML systems, from lecture 4 slides by Dr. Chung*

4.1. Data

NST only requires a single pair of content and style images, while Cycle GAN requires two groups of source and target domain images. Additionally, almost all example data sets from Cycle GAN paper contain at least 800-1000 images for either group. So, to use Cycle GAN, you have to spend time preparing data. If your source or target objects are in Image Net database, you could easily use Image Net data. However, if your source or target are not in the Image Net categories, you will have to collect all images from scratch.

4.2. Algorithm

Algorithm Complexity
We first measured the time complexity for each models by count of FLOPS.

Time Complexity	NST by Gatys al.	Cycle GAN
# Flops	15.32×10^9	117.5×10^9

Then, we calculated the total number of parameters of two algorithm's network architecture and found a total of 20.024 millions of parameters in VGG16 backbone, while there are 14.143 millions of parameters in Cycle GAN: 11.378 millions parameters in the generator plus another 2.765 millions of parameter the discriminator.

Memory Complexity	NST by Gatys al.	Cycle GAN
# Params (million)	20.024	14.143
Model Size (MB)	548	28.29

We got CycleGAN model size of 28.29 M by doubling the sum of the generator and the discriminator sizes.

Model Architecture
These numbers are coherent with the number of layers in two algorithms. The NST model used VGG16 which is made up of 16 layers of CNN blocks, while the generator of CycleGAN is made up of only 6 blocks of conv-instance_norm-relu modules plus a PatchGAN discriminator.

4.3. Time

Training time
There are generally more hyper-parameter options for Cycle GAN due to its generator-discriminator architecture, which makes hyper-parameter tuning more challenging. Also, GAN itself suffer from problems of 1) gradient vanishing 2) model collapse 3) non-convergence. Since cycle-GAN has need to train two GANs (source-to-target domain mapping and inversely target-to-source domain), it could be more fragile to problems above. Compared with Cycle GAN, NST itself requires no training, and thus doesn't suffer from these headaches.

Inference time
Cycle GAN generator is much smaller than VGG16, so it used only 6.98 seconds on a Tesla P100 for a single image, beating 14 seconds for NST on the same device.

5. Conclusion

Which algorithm is better? It depends.

Cycle GAN could deal with far more domain adaptations, including both appearance and style transfer, while NST, as its name suggested, is only able to transfer image styles (basically color and texture). Cycle GAN is able to tweak location/lower-level features of objects, if provided with 'appropriate' training data. For example, both NST and CycleGAN are able to 1) transfer artistic style of an image, 2) change season setting of an image, but only Cycle GAN could 3) change apple into orange. Both algorithms cannot change object sizes.

Meanwhile, Cycle GAN requires more time resources as well as data resources for training. NST requires no training, while you need to prepare 2 groups of images from both source and target domains in order to train a Cycle GAN model. The GAN training process could be challenging, as GANs are sensitive to hyper-parameter options, harder to converge.

Additional concern of NST is that source (content) and target (style) image have to be in the same size, since NST uses pixel-to-pixel L2 loss to measure distance between the content and input image.

In general, Cycle GAN is more capable but more demanding in resources. Therefore, you may choose these algorithms according to your needs.

6. Lessons learned

In this project, we connected the dots of machine learning and deep learning basics, and ML system knowledge. We taught

ourselves an advanced topic in Machine Learning 'neural style transfer' and learned the methodologies of two popular algorithms in the field. We utilized the ML system knowledge in class to compare these methods and experimented with some cloud services.

Particularly, we benefited from several **training tricks**: **1)** carefully selecting learning rate for different combinations of ML model and optimizer; **2)** carefully initializing the weight. For example, for faster convergence, we replace the white noise image by the content image. **3)** normalization mean and standard deviation would affect training a lot more than we previously anticipated. For example, normalizing using our set up works better than using the exact mean standard deviation of your data set.

7. Future Work

If time permitted, we would continue experimenting with the following questions.
First, would the residual connection improve NST performance? We could experiment with that by replacing VGG 16 with a Resnet 18 [11]. Second, why the CycleGAN paper authors choose instance normalization for each conv-norm-relu module instead of batch normalization? Third, we might want to answer many questions remained uncovered. For example, is the cycle consistency loss really working? What will happen if we throw away the Cycle Consistency term in the Cycle GAN loss function?
We might design new experiments and conduct further hyperparameter tuning for these purposes.

8. Appendix

8.1. Complexity Estimation and Measurements

We used NVIDIA nvprof and torch.profiler to get GPU performance details for the models, and torchsummary to get model memory complexity.

8.2. IBM cloud Kubernetes Deployment

We later deployed both models on IBM Cloud Kubernetes Service. To see the code and reproduce results, please refer to our code here: Project Github Repository

9. References

[1] A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin, "Image analogies," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 2001, pp. 327–340.

[2] Y. Jing, Y. Yang, Z. Feng, J. Ye, Y. Yu, and M. Song, "Neural style transfer: A review," *IEEE transactions on visualization and computer graphics*, vol. 26, no. 11, pp. 3365–3385, 2019.

[3] L. A. Gatys, A. S. Ecker, and M. Bethge, "Image style transfer using convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2414–2423.

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[5] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *arXiv preprint arXiv:1406.2661*, 2014.

[6] J. Adler and S. Lunz, "Banach wasserstein gan," *arXiv preprint arXiv:1806.06621*, 2018.

[7] W. Fang, F. Zhang, V. S. Sheng, and Y. Ding, "A method for improving cnn-based image recognition using dcgan," *Computers, Materials and Continua*, vol. 57, no. 1, pp. 167–178, 2018.

[8] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2223–2232.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[10] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal, "Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization," *ACM Transactions on Mathematical Software (TOMS)*, vol. 23, no. 4, pp. 550–560, 1997.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.