**Final Project: Parallelized Neural Network Photo Style Transfer**

Team: Manxi Lu, Shulin Cao, Chao Fan

## 1. Introduction

Burning social media ignites our passions to share life moments with friends all over the world. Compared to traditional filters like 'vivid', photo style transfer has become a hot topic for its highly fun and interactive properties. In this project, we will firstly explore and get comprehensive understanding on how deep learning neural network combines content image with certain style. Besides, another objective of the project is to practice several parallel computing tools and techniques that helps accelerate the transformation. Specifically, there are three models: multithreading after data decomposition, PyTorch with CUDA and Keras with multi-GPU that we formulated to reduce the running time. Meanwhile, three content images with different feature textures like curves, blank etc are tested with the same style image to show the different performance of the models.

## 2. Methodology

The application skeleton is mainly from Gatys, et al 2015 [1]. The main idea of the model is to reduce the sum of errors which caused by comparing the generate image to content image and style image. What is special is the way they use Gram matrix to represent the style. Gram matrix of a feature map is calculated by multiplying its transpose, so that a correlation matrix within that channel is formulated. For example a curve on the right corner will be recorded with respect to its location, and this style could be learnt by Gram matrix. Here are the basic structures of the model (after loading content image and style image):

1. Feed the content image and style image towards pretrained model (eg. VGG16)
2. Compute content cost: $J_{content}(C, G) = \frac{1}{4n_H n_w n_c} \sum_{all\ entries} (a^{(C)} - a^{(G)})^2$
3. Reshape the filter matrix from 3D tensor $n_H * n_W * n_c$ into 2D tensor $(n_H * n_W) * n_c$
4. Compute style matrix G: Multiplication of the unrolled filter matrix and its transpose
5. Compute style_layer_cost: $J^{[l]}_{style}(S, G) = \frac{1}{4n_c^2(n_H * n_w)^2} \sum_{i=1}^{n_c} \sum_{j=1}^{n_c} (G^{(S)}_{ij} - G^{(G)}_{ij})^2$
6. Compute style_cost: $J_{style}(S, G) = \sum_l \lambda^{[l]} J^{[l]}_{style}(S, G)$
7. Compute total_cost: $J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G)$
8. Using an optimizer to minimize the total_cost by multiple iterations

Where $n_H * n_W * n_c$ refers to the dimension of the image, $a^{(C)}$, $a^{(S)}$, $a^{(G)}$ are the volume corresponding to a hidden layer activations for content, style and generated image respectively. Base on this algorithm, we explored the following three parallel computing models to accelerate the process. The following four images are tested for the models where first image is the style image "Starry Night" and the others are content images.




**Part 1: Multithreading Process:**

There are several ways to run the original program in multithreading process when we do photo style transfer. In our project, the main idea is to segment the content image into several pieces, and segment the style image as the same shape of the content image. Then we apply "multiprocessing" module on python to create the same number of the image pieces' pool. Lastly, we run the "photo style transfer" program ("art_nn"[2]) on each pieces of image at the same time with multithreading process.

**Part 2: PyTorch with CUDA**

One way to accelerate the neural style transfer is to use GPU computing. Because our code is based on the python, so we utilized the package PyTorch recommended in lectures to accelerate it on CUDA. An outstanding advantage of PyTorch is its mechanism on dynamically auto-calculation for desired variables' gradients, which decreases the complexity of programming. In this model, there are four parts to compose the demo: model loader, image processing, loss function and optimization run. Firstly we referenced Simonyan and Zisserman's paper's [3] pretrained VGG19 network and image processing methods from constructed in PyTorch by the author Leon Gayts [4] so that an tensor output from the specified layer that can grasp relative representative features of general

images could be expected by feeding the network with image input. Because the upgrades occur at generated image rather than the weights parameters within the convolutional neural networks, so it is required to make sure the code do not take gradients of model's parameters by setting "*param.requires_grad == False*". Because the image input are loaded as PIL image valued in [0,1], *preprocess function* composed of a sequential steps of "*torchvision.transforms"* API are conducted which transform the image input into a normalized RGB tensor valued in [0,255] which is requirement of VGG19 model input. Similarly, after the completion of iterations, the final generated_image are transformed back to PIL image by *postprocess function*.

As for the loss function part, those function definitions generally follow the mathematical algorithm proposed in Gatys et al [1], but specifically gram_matrix need to be unrolled from 4 dimensions into 2 dimensions as *(B*C)*(H*W)* where B as batch size 1 in default, C as channel numbers, and H as height of image and W as width of image. And after computing gram matrix of the image, its output need to be normalized by dividing number of elements in each channel. Without normalization, the layer at the earlier stage has higher weighted loss during the gradient descent because of larger dimensions, which might lose more importance from deep layer's features. The distance function that measures how close the generated image to the content image and style image are MSE loss: *torch.mean(torch.pow((gram_matrix(input)-gram_matrix(target)),2))*. By calculating the minimal square error between the gram matrix of target feature maps value passing by *vgg(style, layer)* and gram matrix of generated image feature maps value passing by *vgg(generated, layer)*, the total style error is summed by all layers together times with respective weights. Since we only need to backpropagate on generate image value, so *target.detach()* is need to create the same variable of target whereas it does not needs gradients computation (*requires_grad == False*). In other words, it becomes leaf node in the computation graph. And the total_loss function will be the sum of style error and content error multiplied by respective hyperparameters.

For last step, we formulate an iterative model that get appropriate generated image by minimizing objective of total loss function. After initializing the generate image with *requires_grad == True*, according to the paper, we used the suggested optimization algorithm *L-BFGS* for gradient descents on generate image. Specifically, *optimizer.zero_grad()* makes sure all gradients in the network are cleared, then *loss.backward()* implements automatically for gradient descents on total loss. Finally, the model declares all variables or module by adding ending *.cuda()* so that the model could save scalable running time working on GPU if there is a GPU available in testing machine platform. This is another benefit for using PyTorch where API is easy set up for accessing CUDA.

**Part 3: VGG16(Keras Pretrained) + Tensorflow**

Tested Platform:

a. 2.7 GHz Intel Core i5 - 6th Gen 2 cores and 4 threads

b. 3.1 GHz Intel Core i5 - 7th Gen 4 cores and 8 threads

c. Intel 8700k 16G CPU + Nvidia GeForce 1070 8G GPU

We first basically implemented the idea from Gatys, et al 2015 [1] with default trained weights from ImageNet and a randomly chosen parameter set to reach a baseline work. The content image that we chose is a animated dog while the style is the famous Starry Night.
The VGG16 model includes a training structure of 5 main convolutional parts shown as the following figure (from Quora) together with the last 3 dense layers:

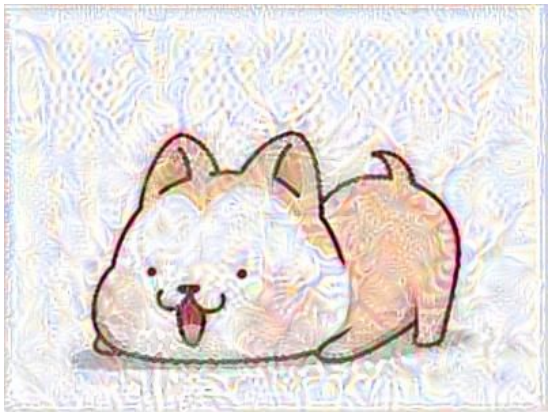
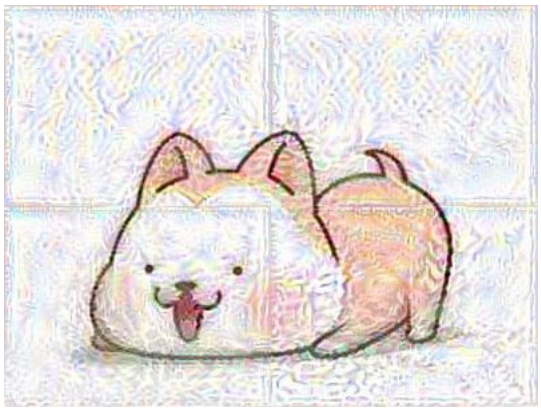

Through an input of (3, 224, 224) into the model, we are able to implement the VGG 16 by assigning self-tuned parameters which would generate an output without the dense layers. Then we could use optimize the loss functions by calculating the loss from the weighted contents, styles and total weights to complete the optimization process. The author has provided the basic framework source codes and in our project we have also included the basic framework but changed its form to be more suitable for ours model [3,5].
Keras has also provided the pretrained VGG19 model which three more layers would be added to the 256 and the two 512 parts. Though in our project we will not to reproduce the VGG19 model but will combine the this property with a reduced model for VGG16.
Also, we have tried different combinations of weights for content, styles and total weight. Within the training layers, we can also have a weight vector for different outputs from the layers to improve the final results.
We have also tried to change the VGG16 model into a new model which keeps the same structure for 5 main parts but only have 13 layers (we removed the 512 layers and add three more 256 layers). This is achieved by reducing the picture size to 256 by 256 from 512 by 512. The reason for doing so is due to the fact that the resolution of many pictures are usually restricted from online sources. Thus reducing the size to 256 by 256 not only reduce the total training time but also increase the style transfer quality. This method could be working, however, through the training we are not able to solve the resources exhausted errors. Thus we will only show the results from the original VGG 16 model.
Both methods are then tested on different platforms that we have. Though Comet has provided better hardwares, in this project our main focus is to have parallel computing available for basic use.

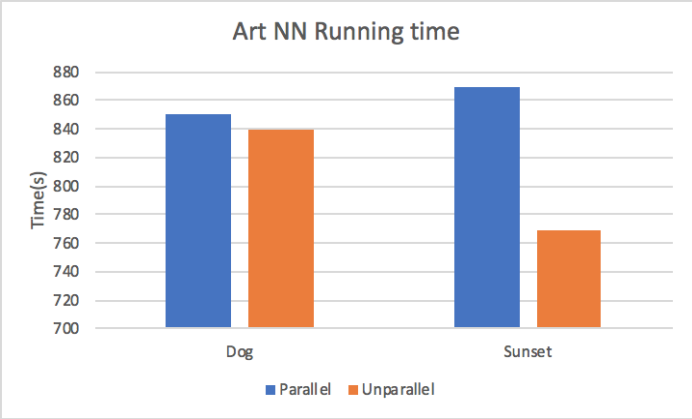
## 3. Results

### Part 1: Multiprocessing method

By applying the multiprocessing method on both of the “dog” and “sunset” picture, the left two pictures are generated under normal method, the right two pictures are generated under parallel multiprocessing method. We divide the figure into 4 equal pieces. From the timing figure, we can see when we apply the method on view figure like “sunset”, the time is greatly saved. However, when we apply the method on “dog” file, the time for parallel is even higher than normal program. The reason of this situation may be the blanked of the dog is too large if we divide it into four small pieces. Hence, when we apply this method, we need to check the content of the content figure detailly.

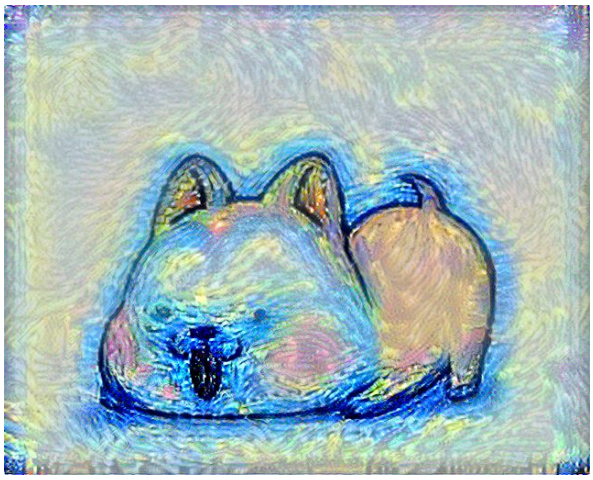


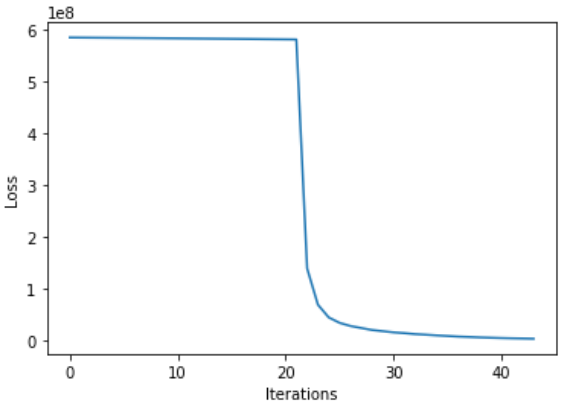



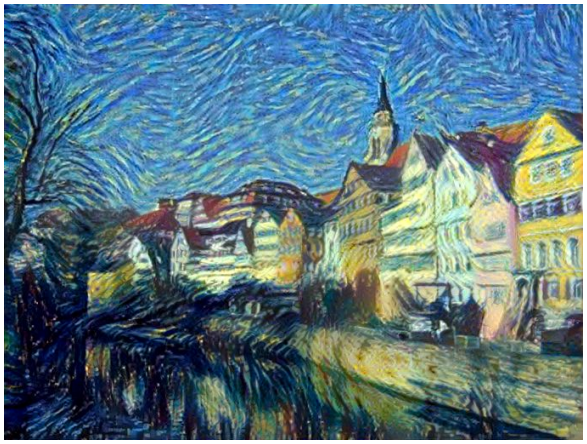


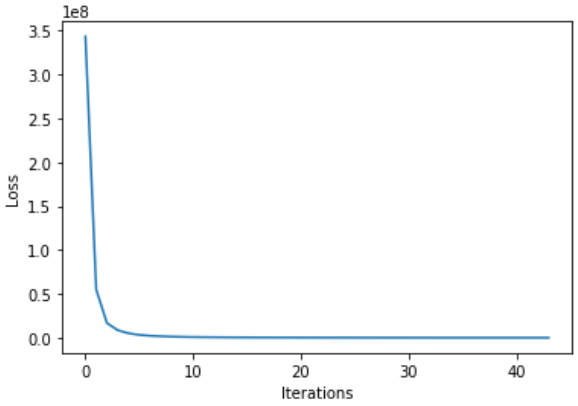

**Part 2: PyTorch**

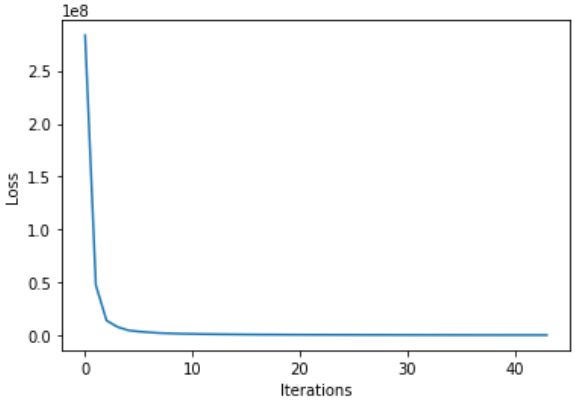
By using the initial equal hyperparameters weighting (1,1) of content and style and Gatys et al suggested layer weights for styles, the output image generated by the PyTorch model demonstrates relative good outputs except for the dog image. This may because the dog image has relatively low complexity on features, in other words, the large white margin flattens the curvature of style features especially the curves from Starry Night, which is also shown in the loss graph. From the output we can see this model have robustness problem for unbalanced complexity content-style image pairs. Namely, the complex content of city and sunset images shows detailed transformation from complex style of Starry Night, but the simple structure of dog image performs bad transformation adaptation towards the complicated style.





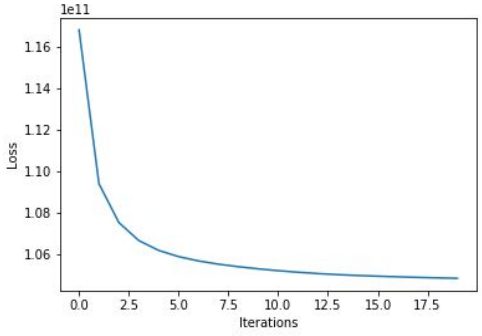




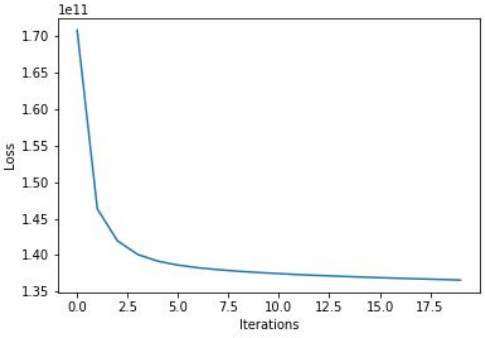




**Part 3: Keras+Tensorflow**

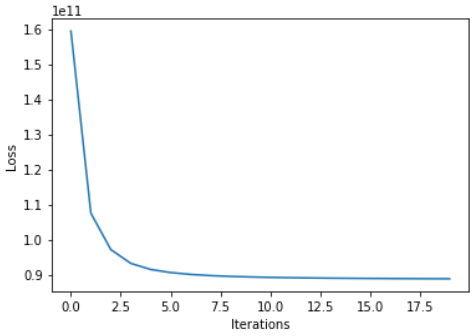
Our first attempt on the dog image which has a large white margin show a very large loss values after 20 iterations of optimization. The weight of content, style and total weight was (0.1, 2.0, 1.0) and from the result we can see the content still has a larger influence.



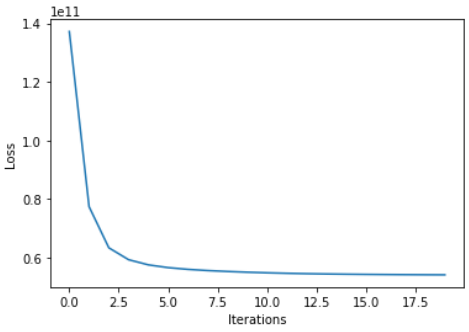

Then we increased the style weight to 5.0 which style image has a larger ratio through training. This returns a better result compared to the starry night photo, however, this brings larger loss.


Then we tested a combination of weight changes for the five convolutional layers on the other two photos. Not only the results are significantly better, but the loss function plot shows better decrease to the far lower results. The weights used here is (0.1, (5.0,10.0,10.0,20.0,20.0),1.0). More results could come out if other weights combinations are tested. The evaluations are based on the loss function, however, we noticed that the difference among the three pictures also show the potential drawbacks of this model applied to style transfer.

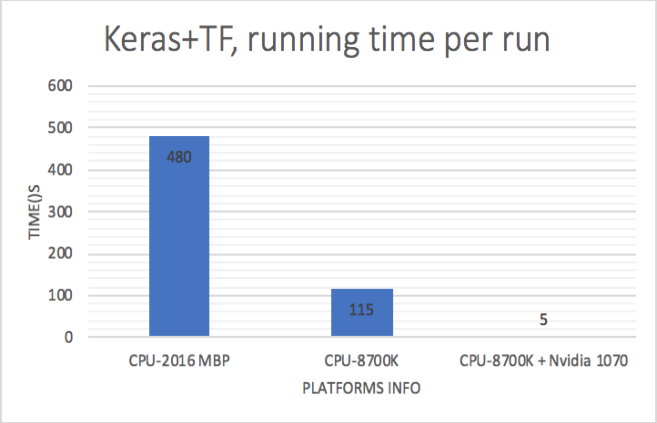


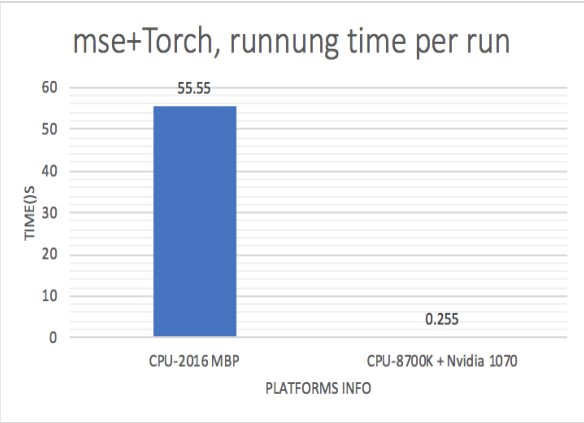



We will see better result for the starry night style transferred to buildings and sunset than that to the happy dog picture. If we take a notice on the picture features, more details have been presented on the buildings that others and can better combine with the multi-line featured starry night picture.

**Part 4: Running Time Analysis**

We have also tested the running time on the three different models and also in the three different platforms we have mentioned in the methodology part. It shows that the average running time has significantly dropped and they are more than 20 times faster with GPU support compared to the Intel 8700K results. We notice that the use of 8700K is able to increase about 4 times of the 2016 6th gen i5 CPU. Compared to the unit running time of Keras model and PyTorch model, the PyTorch has higher efficiency and also shows over 200 times acceleration using GPU computing on CUDA.




## Conclusion

Although we improve the speed when we apply our multithreading process method on image and the effect on each pieces of image is quite well, we are unable to overcome the rough edge between each pieces. There are two ways to overcome the disadvantages. Firstly, in our strategy to segment the content image, we use simple “equal-size” way. Actually, there are many fancy ways to segment the content image into pieces of similar content components. One of the most common image segmentation method is deep learning from A. Garcia-Garcia , et al 2015 [6]. The other widely-used method, especially on medical image segmentation, is level-set method from Osher[7]. Second, we can apply smooth image method to overcome the rough edges. Both of these two ways can definitely improve the effect. Third,depends on the weights parameters on style and content. Possible grid searching algorithm could apply for tuning the hyperparameters. The three models all appeal that the matched complexity of content-style demonstrates good performance of output image, but the unbalanced complexity will largely reduce the quality of output. Finally, because group members are mainly self-educated on deep learning area, the lack of understanding and experience on network model could decreases the accuracy and efficiency of the programming. However, the parallel computing methodology including data decomposition, GPU computing and multiprocessing shows remarkable improvement on running time. Therefore, we will explore more parallel computing techniques for future study.

## Reference:

[1]Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. "A neural algorithm of artistic style." *arXiv preprint arXiv:1508.06576* (2015).
[2]JudasDie, (2015) Deep Learning & Art: Neural Style Transfer [Source Code] https://github.com/JudasDie/deeplearning.ai/blob/master/Convolutional%20Neural%20Networks/week4/ArtTrans/Art%2BGeneration%2Bwith%2BNeural%2BStyle%2BTransfer%2B-%2Bv2.ipynb
[3]Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556.*
[4] Gatys, L (2017) Pytorch Neural Style Transfer on Github [Source code]. https://github.com/leongatys/PytorchNeuralStyleTransfer/blob/master/NeuralStyleTransfer.ipynb
[5] Fchollet, (2018) Keras application Team VGG models [Source Code] https://github.com/keras-team/keras/tree/master/keras/applications
[6]A. Garcia-Garcia, S. Orts-Escolano, S.O. Oprea, V. Villena-Martinez, and J. Garcia-Rodriguez (2017). A Review on Deep Learning Techniques Applied to Semantic Segmentation *arXiv:1703.06857v1.*
[7] S. Osher, Ronald P. Fedkiwb (2010). Level Set Methods: An Overview and Some recent Results *Journal Of Computational Physics Volume 169, Issue 2, 20 May 2001*