Style Transfer of Images and Videos

Yicheng Shen, Zhenyuan Wang
CIS students
University of Pennsylvania

***Abstract*—For this project, we implement style transfer and analyze various design choices in the algorithm, which is important for learning from the artistic styles of existing artworks and generating new visually pleasing results. We implement two versions of style transfer. The first version requires a long runtime for images with high resolution. We explore many possible solutions and implement the second version that provides a significant speedup. We are able to efficiently stylize high resolution on GPU and scale up to stylizing videos, which requires transforming a large number of frames of the videos.**

In addition to implementing the style transfer algorithm, we evaluate different design decisions in our experiments. We compare the performance of our two implementations. Moreover, we study the effects of using different model architectures, loss functions, optimizers, etc. In this report, we document the implementation ideas, analyze our evaluation results, and present our visual results.

I. Introduction

Style transfer is the process of applying the style of one image or artwork to another image. It is a technique used in image processing and computer graphics that allows the user to combine the content of one image with the style of another image to create a new, unique image. This is typically achieved using deep learning algorithms, which are able to analyze the style of an image and then apply that style to another image. The result is a new image that incorporates the content of the original image, but with the stylistic elements of the other image.

Style transfer has the potential for meaningful applications: it can be a useful tool for artists, photographers, and others who want to experiment with different styles and create unique visual effects. For example, photographers can use style transfer to add artistic flair to their photos or to create stylized versions of their images for use in advertising or other contexts. Designers can use style transfer to create new and unique visual designs or to apply the style of a specific artist or designer to their own work. The applications can even extend beyond generating artistic pieces: the algorithm demonstrates its ability to generate images with higher resolutions [4]. This can be applied in practical fields, such as medical imaging.

II. Goals and Objectives

This project aims to implement style transfer. The process of transferring artistic styles from masterpieces onto arbitrary photos seems to be a "magical" process. We aim to understand what is under the hood. Moreover, we hope to use our implementation to create visually pleasing results.

Specifically, our objectives include i) training a model that can transform images with a small size, ii) exploring methods to scale up to high-resolution images, iii) converting our code into a pipeline so that we can conveniently learn new styles and batch transform images, and iv) extending style transfer to process videos. In addition to implementing the algorithm, we also aim to study the performance of various design choices, such as different backbones, loss functions, and optimizers.

III. Related Works

There have been many prior works on style transfer, with a range of approaches and techniques being proposed. Gatys et al. proposed the neural style transfer approach, which uses convolutional neural networks (CNNs) to extract high-level semantic representations of the content and style of two images, and a loss function is defined as the difference between these representations [2]. Then this loss function is used to train the network to transfer the style of one image to the other while preserving the content. This approach has been shown to produce high-quality results and has become one of the most widely used techniques for style transfer.

Johnson et al. proposed the use of a pre-trained VGG network as the feature extractor for the content and style representations, which improved the quality of the results and showed a large speedup [4]. Huang et al. proposed another approach, fast neural style transfer, which uses a feedforward neural network to transfer the style of one image to another [3]. This approach is faster than neural style transfer and can be used to process large numbers of images in real-time.

For computer vision tasks, the OpenCV (Open Computer Vision) library provides great support [1]. It could help us perform a wide range of operations on images and videos, including reading, writing, and displaying images and videos; applying image filters and transformations; and performing feature detection, object detection, and other computer vision algorithms.

For the deep learning components, our project relies on PyTorch [5], which is an open-source machine learning library for Python. PyTorch allows users to easily define and train neural networks using a simple, high-level API. It also provides weights of many pre-trained models in its model zoo. Moreover, training written in PyTorch can be conveniently run on GPU.

IV. Implementations

We implement 2 versions of style transfer. The first one generally follows ideas in the paper of Gatys et al. [2] and we refer to it as **Version 1** in the following sections. The second one generally follows ideas in the paper of Johnson et al. [4] and we refer to it as **Version 2** in the following sections.

A. Version 1

According to ideas in [2], the neural style transfer algorithm works by first extracting high-level, semantic representations of the content and style of the content image and the style image using a pretrained CNN. These representations capture the overall structure and content of the content image and the textures, colors, and other visual details of the style image, abstracting away from the specific pixels and low-level details of the images. The loss function is then defined as the difference between the output image and these representations, and this loss is used to train the network to combine the content and style of the two images. The network is trained to minimize the difference between the output image and the target image, and this enables it to learn to combine the content and style of the two input images.

We take the following steps to implement Version 1:

1) We prepare 2 source images: i) a content image, and ii) a style image.
   - Two images are resized to have the same shape. We start with a small size for debugging purposes.
   - On CPU, we set the image size to be 128×128 pixels. On GPU, we scale up and increase the size to 512×512 and then 1024×1024 pixels.
2) We prepare the input image, which is the gradient descent initialization. We have 3 options: i) white noise, ii) the content image, and iii) the style image. The paper [2] indicates that options (ii) and (iii) tend to bias towards the spatial structure of the initialization. In our attempts, we find that (ii) works well. Starting with the content image ensures that the original general structure is easily identifiable, thus producing more ideal results.
3) We select a pre-trained model as our backbone. We first try the VGG network pretrained on the ImageNet dataset and then the other pre-trained models, such as AlexNet.
   - We only use the feature maps in the backbone network, so we focus on the convolution and pooling layers which contain information about image features.
   - Note that in the original VGG network, there are fully connected layers following the convolution layers for tackling the classification task. Here we only need the feature maps. So we removed those fully connected layers.
   - We normalize the network: scale the weights such that the mean activation of each convolutional filter over images and positions is equal to one.
   - We use average pooling instead of max pooling.
4) We implement content loss, which measures the difference between the content of the generated image and the content of the original image. Content loss is calculated using the output of a specific convolutional layer. Content loss is defined as

$$\mathcal{L}_{\text{content}}(p, x, l) = \frac{1}{2} \sum_{i,j} (F_{i,j}^l - P_{i,j}^l) \tag{1}$$

   where l is the layer in the network; p is the content image and $P_{i,j}^l$ is its feature representation at the j^{th} position of the i^{th} filter; x is the input/generated image and $F_{i,j}^l$ is its feature representation.
5) We implement style loss, which measures the difference between the style of the generated image and the style of the reference image. This loss is calculated using the Gram matrix of the activations of specific layers in the neural network. The Gram matrix captures the correlations between the different channels in a layer, which encode the style information in the image. The goal of the style loss function is to match the style of the generated image to the style of the reference image, while still preserving the content of the original image. Style loss is defined as

$$\mathcal{L}_{\text{style}}(a, x) = \sum_{l=0}^{L} w_l E_l \tag{2}$$

   where a is the style image and x is the input/generated image; w_l is the weight of layer l; E_l is

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - A_{ij}^l)^2 \tag{3}$$

   where N is the number of filters; M is the size (height $\times$ width) of each filter; A_{ij}^l and G_{ij}^l are style representations, which are defined by the Gram matrix

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l. \tag{4}$$

   Note that G_{ij}^l is the inner product between vectorized feature maps i and j.
6) We add the content loss and style loss calculations to the backbone network. The sum of the content loss and the style loss is the total loss. In optimization, we use the total loss for backpropagation.
   - We match style representations up to deeper layers (stages 1-5) because deeper layers capture higher-level style information. For example, the first convolutional layer in a neural network might capture the colors and textures in an image, while deeper layers capture more abstract style information, such as the shapes and arrangements of objects in the image. By matching the style of the generated image to the style of the reference image at deeper layers, the generated image can be stylized in a more sophisticated and nuanced way.
   - We match content representations at stage 4. (Each stage means a shrink in filter size and an increase in filters.) This is because the activations at stage 4 encode the content of the image at a high level of abstraction. This means that the content loss function can capture the overall structure and composition of the image, while still allowing the generated image to be stylized according to the style of the reference image.
   - In terms of implementation, we add calculations of content loss and style loss into the backbone network at the positions described above.

7) After defining content and style loss, and setting up the backbone network, we need to determine another series of design choices for training the model, such as the optimizer, the loss functions, and different hyperparameter choices.
   Optimizer: We use LBFGS (Limited-memory Broyden–Fletcher–Goldfarb–Shanno) optimizer, which approximates the Hessian matrix of the loss function using a limited amount of memory. This allows it to make more efficient use of the available resources and converge to a good solution more quickly than other optimizers. In our experiments, we also compare LBFGS with other optimizers, such as SGD and Adam.
   Loss Function: We use MSE (mean squared error) loss, which is a commonly used loss function. We also compare it with other loss functions in our experiments.
   Hyperparameter choices: We have a number of hyperparameters to tune, including the number of epochs, the weight for style, the weight for content, and the learning rate. We initialize our optimization with the content image, so we set the style weight to be much higher than the content weight. It is important to note that these choices differ for training different styles.
8) We train on many different styles and analyze the results.

After we feel confident about our implementation on processing small images, we attempt to scale up. We are able to scale up to images with 1024×1024 pixels with Version 1. However, as we scale up more, we notice a significant slowdown while possibly facing an out-of-memory issue using colab. Considering smartphones, not to mention the fancy digital cameras, can produce photos with a much higher resolution, we are not satisfied with the performance of Version 1 and we want to implement a faster version that can efficiently process high-resolution images. Therefore, we explore other methods and decide to follow ideas in [4] to implement Version 2.

B. Version 2

Version 2 shares similarities with Version 1. We also use the pretrained VGG network as a loss network. Style loss and content loss are computed through the loss network. Although there are many similarities, what we optimize is different. In the previous version, we optimize the input which we initialize as the content image (or the white noise image). In this version, we optimize the weights of the Image Transformation Network. The Image Transformation Network is a residual network consisting of a combination of convolution layers, residual layers, and deconvolutional layers to extract the style and content features from the input images, and then uses these features to generate a new output image that combines the content of one image with the style of the other. What we optimize in training is the weights of this network. Therefore, for each style, we can train and get a set of weights for the transformation network.

With a set of weights learned from an artistic style, we can transform an arbitrary number of images into that particular artistic style. It is important to note that having the trained transformation networks enables us to significantly speed up our runtime for processing images. Training each network is time-consuming, but once they are trained, we can transform high-resolution images within seconds on GPU.

We take the following steps to implement Version 2:

1) Similar to Version 1, we prepare 2 source images: a content image and a style image.
2) We download an image dataset for training our image transformation network. We use the training portion of the 2017 COCO (Common Objects in Context) dataset, which contains 118K images with a total size of 18 GB.
3) Same as Version 1, we also use the VGG16 network as our loss network. We use it to calculate the content loss and style loss.
4) We use a similar architecture of the image transformation network in [4]. It consists of 3 convolutional layers 5 residual layers, and 2 deconvolutional layers followed by another convolutional layer. This architecture allows us to first downsample the input image and then upsample it, which speeds up computation. The residual layers in this architecture help the network to learn the structure of the input image, which is the desired behavior for the style transfer task. In our training process, we optimize the weights of this network.
5) The content loss is similar to the one defined in the previous version, which is the normalized mean squared distance between content representations. To write it out, it is
$$\mathcal{L}_{\text{content}}(\hat{y}, y, l) = \frac{1}{C_l H_l W_l} \|\phi_l(\hat{y}) - \phi_l(y)\|_2^2 \quad (5)$$
   where l is the layer, and C, H, W are the feature map's shape; $\phi(\hat{y})$ is content representation of the generated image, and $\phi(y)$ is content representation of the target image.
6) The style loss is also similar to the previous version, which is the mean squared distance between the Gram matrix of style representations. (See Version 1 for Gram matrix.) To write it out, the style loss of a single layer is
$$\mathcal{L}_{\text{style}}(\hat{y}, y, l) = \|G_l^{\phi}(\hat{y}) - G_l^{\phi}(y)\|_2^2 \quad (6)$$
   where l is the layer; $G^{\phi}(\hat{y})$ is the Gram matrix of the generated image's style representation and $G^{\phi}(y)$ is the Gram matrix of the target image's style representation. The style loss of multiple layers is the sum of the losses of all single layers.
7) We sum up the content loss and the style loss to get the total loss, which is used for backpropagation.
8) After setting up the loss functions, the loss network, and the image transformation network, we implement the training portion. We take training data which are images from the COCO dataset and perform a series of transformations, such as resizing, center corping, and converting to tensor. We load the data into PyTorch's data loader, which returns a batch of images at a time. In training, we iterate through all batches in the data loader. The weights of the image transformation network are updated based on gradients calculated from each batch.

By the end of the training, we get a set of weights for the specific style that we train on.

9) We use the trained weights to transfer images. To achieve this, we use the image transformation network to load in a set of weights that corresponds to a specific style and pass a content image through the network to get a stylized image.
10) We scale up to high-resolution images. This version of implementation provides us with a significant speedup. We are able to transform a 4K image within seconds on GPU. We include more details in the experiment section.
11) With the capability to efficiently transform images, we extend style transfer from images to videos. We first implement a function to transform a batch of different content images into the same style. This speeds up the transformation even more by leveraging GPU's parallel processing power of numeric operations. It also sets a good foundation for style transfer of videos.
When we tackle the problem of style tranfer for videos, we break it into a few steps: i) We use the OpenCV library to extract all frames from the video so that we can treat each frame like an input image which we can pass into the image transformation network. ii) We utilize the batch transformation function we define to efficiently convert all the frames into stylized frames. iii) We extract specifications of the original video, such as the FPS and dimensions. iv) Using the specifications, we stitch the stylized frames together to generate a stylized video.

C. Automated Efficient Workflow

In addition to completing 2 versions of implementations, we spend much time and effort coding an automated efficient workflow for training new styles and generating stylized images and videos. We manage to combine all of our Python code implementing functionalities specified in Version 2 (which is the version with better performance) into a single Colab notebook. Any user only needs to correctly modify the configuration section which is the first code cell in the notebook and press the "run all" button to perform training or style transferring using existing pretrained weights.

First, we automate the environment setup in Colab, we use bash commands and scripts to automatically download the training dataset and pretrained network weights. We make sure they are placed in the correct directories so that the following code for training can run without errors.

Furthermore, we automate file loading and saving. A user just needs to set the correct root directory as a parameter in the configuration. Then, the program will create the required subdirectories which are used for saving files during training. For loading pretrained style weights, a user can set the name of the style weights file in the configuration, and the program will automatically locate the file and load the image transformation network with the weights.

Most importantly, we provide useful functionalities to simplify the training and model selection process. We checkpoint the model every 500 batches during training and save the model checkpoints to Google Drive. This functionality allows us to recover from any interruptions or failures that may occur during training. For example, if our training process is interrupted due to a disconnection with Colab caused by a power cut, we can use a model checkpoint to resume training from the point where it was interrupted, rather than starting from scratch.

Checkpointing models also help us with model selection. Our model checkpoints saved in Google drive can be used to evaluate the performance of a model at different stages of training. We can conveniently pick a number of models at different stages and use them to transform the same content image. Then we can visually compare their performance.

Another mechanism that we design for visual debugging is that every 500 batches, we randomly select a training image from the current batch and display itself and its stylized version side by side. This provides a way for us to visually monitor the training progress. If any hyperparameter setting goes wrong, we will be able to catch it early on without the need to wait until the end of the training.

Overall, the automated workflow largely improves our efficiency on training and generating visually pleasing results. We are able to train many sets of weights for different styles in a short period of time using the limited GPU resource on Colab. In the visual results section, we will present some good looking images that we stylize.

V. Experiments

In this section, we compare the qualitative and quantitative results from Version 1 and 2. Moreover, we design different sets of experiments to evaluate the algorithm with various backbones, loss functions, and optimizers in Version 1. We compare and analyze their differences in training time, loss trends in epochs, and qualitative visual results.

A. Comparison between Implementation Version 1 and 2

Qualitative Result

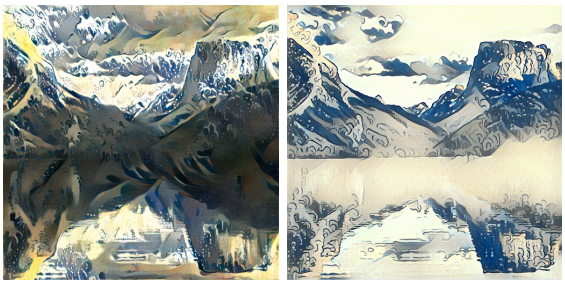

Fig. 1. Both images above are using the same content (A mountain picture) and style image (The Great Wave of Kangagawa, Hokusai), but are generated by different implementation versions. The image on the left is generated by Version 1 and the one on the right is generated by Version 2.

We first evaluate the direct comparisons between visual results generated by the two versions since the quantitative comparisons such as algorithms' efficiency will be meaningful only if they both have fair visual outputs. The visual results are shown in Figure 1.

While it is not an easy task to tell how beautiful an image is, we compared our output images visually and set up our comparison standard using the following two criteria: i) how many features in the content image have been preserved and ii) to what extent the features in the style image have been applied on the resulting image. We considered an output as a failure if we can not easily tell the content image and style image it generated from.

After a lot of tuning of hyperparameters, we are able to achieve similarly good visual results from both versions by testing on various content and style images. However, there are some slight visual differences between two versions from the example shown in Figure 1. One of the differences we observed is that Version 1 seems to preserve more style features than Version 2 does. In the output image version 1 generated, we can see that the clouds are stylized into waves instead and lose the cloud features to a considerable degree. By contrast, we can still observe clear outlines of all the clouds in version 2 which are also applied with the colors learned from the style image. These subtle differences potentially result from the following causes:

1) The hyperparameters such as the content and style weights we set for the training for the models.
2) The layers we selected for the calculation of the content and style loss.
3) The differences in model architectures.
4) The pretrained weights of the backbone network.

Although the two images have some small differences in how outputs are generated, both outputs are visually pleasing and share many similarities in both colors and patterns. Hence, we can further discuss their quantitative distinctions in performance.

Quantitative Result

Image Size	Version 1		Version 2	Speedup
	AlexNet	VGG		
256×256	9.15	23.0	0.020	457x - 1150x
512×512	16.0	78.8	0.070	228x - 1126x
1024×1024	51.4	285	0.172	299x - 1657x

TABLE I
THE PERFORMANCE LISTED ABOVE MEASURES THE SPEED OF TRANSFORMATION IN SECONDS USING COLAB STANDARD RAM GPU.

From the quantitative aspect, we compare the efficiency of both models by investigating the time needed for transforming an image. We list our experiment results in Table I.

We run our Version 1 and Version 2 with images in different sizes of 256, 512, and 1024. Since we apply two backbones in Version 1, we collect results using both AlexNet and VGG models. To evaluate a generalized performance, we use Version 1 and 2 to transform 100 images randomly selected from the MS-COCO dataset, using The Great Wave off Kanagawa as the style image, and then compute the average time each version takes for transforming one image.

From the results, we notice a significant speedup in image transfer with Version 2, especially when we transform images with higher resolutions. Even if we move the backbones to AlexNet in Version 1, a model with much fewer layers, Version 2 still demonstrates its advantage of having hundreds of times of speedup for every image size. Besides, due to the limitations in Colab GPU resource, in Version 1, 1024×1024 is a considerably large size which will consume much time to transform. In contrast, in version 2, the image size can easily be scaled up to 4K and be transformed within 2-3 seconds. The results show the advantage of training an image transformation network in Version 2. In Version 1, for each new image, we need to run optimization on the new image from scratch, which is time-consuming. In contrast, Version 2 only requires 1 trained network for each style to be able to quickly transform any new image.

B. Comparison between Backbones

The following comparisons in sections B, C, and D are evaluated on Version 1 so that we do not need to retrain the model on 83k images every time we change the experiment configurations. All the comparisons are tested on the same content image (mountain image) and the same style image (The Starry Night, Vincent van Gogh).

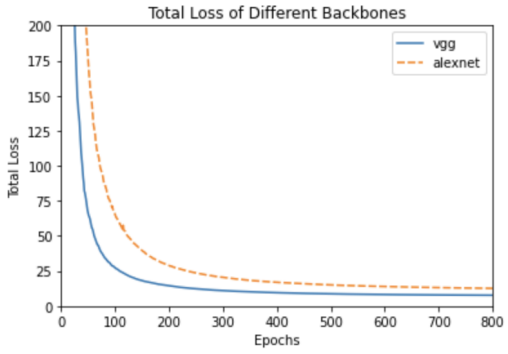

Fig. 2. Comparison of loss history using different backbones.

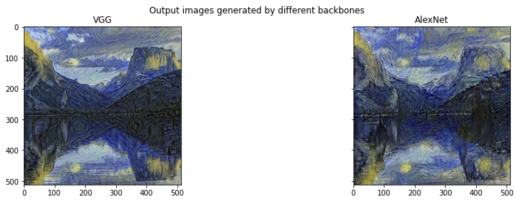

Fig. 3. Comparison of visual results using different backbones.

We evaluate pretrained backbones, VGG-19 and AlexNet, in Version 1 to see their performance differences in the scope of loss trend, runtime, and visual results. Generally, both backbones have similar results in loss trend and visual results as shown in Fig 2 and Fig 3. However, for the same content and style images in the size of 512×512, AlexNet is around 5 times faster than the VGG-19. Hence, for this particular setup in Version 1, AlexNet might be a better choice as a backbone. However, in the case of scaling up, its performance cannot be guaranteed.

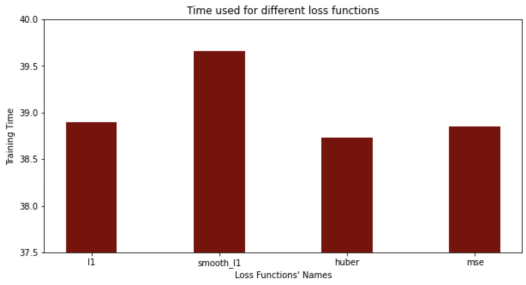

Fig. 4. Comparison of runtimes using different loss functions.

C. Comparison between Loss Functions

We evaluate 4 different loss functions, L1, smooth L1, Huber, and MSE loss under the same backbone (VGG-19) and the same optimizer (LBFGS). All of them share a similar loss trend while transforming. They also achieve similar visual outputs. However, we notice slight differences in training times among loss functions shown in Figure 4. Huber and MSE losses show shorter training times.

D. Comparison between Optimizers

Three optimizers, LBFGS, ADAM, and SGD, are tested under the same backbone (VGG-16) and the same loss function (MSE). As shown in Figure 5, SGD shows a fluctuating pattern with an extremely high total loss with distorts the curves of the other two optimizers. Therefore, for visualization purposes, we plot using the log scale for the y-axis. The result indicates that the optimizer LBFGS has the lowest total loss. In addition, the visual results in Figure 6 demonstrate stronger contrasts between different optimizer choices. Compared with LBFGS, ADAM has more emphasis on content features and limited focus on features in style images; SGD cannot provide a valid visual result with clear content features.

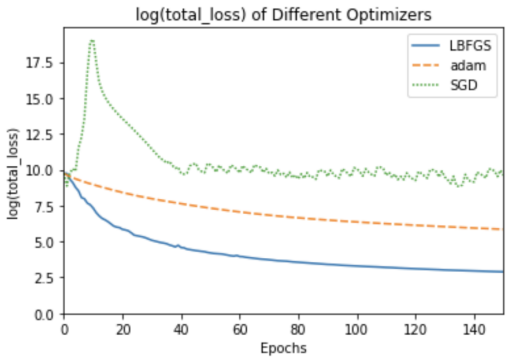

Fig. 5. Comparison of loss history using different optimizers.

VI. Visual Results

A. Version 1 Image Results

Visual results of Version 1 are shown in Figure 7. We use the mountain image as the content image. We stylize it with 3 different style images: specifically, the style images are The Starry Night by Vincent Van Gogh, The Scream by Edvard Munch, and The Great Wave off Kanagawa by Hokusai. All the images are cropped in the size of 512×512.

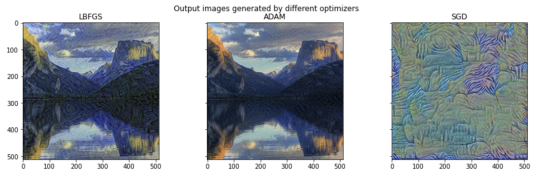

Fig. 6. Comparison of visual results in different optimizers.

B. Version 2 Image Results

The visual results of Version 2 are shown in Figure 8. We emphasize Version 2's capability in transforming high-resolution images: we select a 4k image directly taken by our iPhone as the content image. We transform it with the style image, The Great Wave off Kanagawa. Version 2 is able to transform the fine details on the high-resolution image: for example, the original texture on the walls of the buildings is replaced with wave-like strokes after transformation.

C. Video Results

Different from the transformation of a single image, the transformation of videos requires consideration of the continuity and consistency between frames during the style transfer on the video's frames. Version 2 focuses on high-level features and is able to keep the consistency and flow of videos.

We demonstrate our result by transforming a commercial video of a car brand that used some artistic filming techniques to show off the dreamlike and modern concepts of the brand. We apply our style transfer Version 2 to some parts of the video to enhance the artistic quality of the video. We think our implementation serves as a powerful tool that largely increases the expressiveness of the original video. The transformed video is available on YouTube[1].

VII. Conclusion

In this project, we implement two versions of style transfer. The second version provides a significant speedup. We are able to extend the style transfer from images to videos given the fast speed of the second version. We also evaluate various design choices in our implementation and our results provide insights for enhancing the performance of style transfer. Overall, our project demonstrates style transfer's potential to generate visually pleasing stylized photos and videos. With our results in Version 2, style transfer is also shown to be capable of scaling up and can be used in applications in practice.

As a future step that is inspired by our stylized video result, we could try merging the ideas in style transfer with music generation to build an application that creates stylized music videos, in which the music and the video share a common theme defined by user inputs of style images and music styles.

[1]Link to the transformed video: https://youtu.be/UZIe1LwtETI

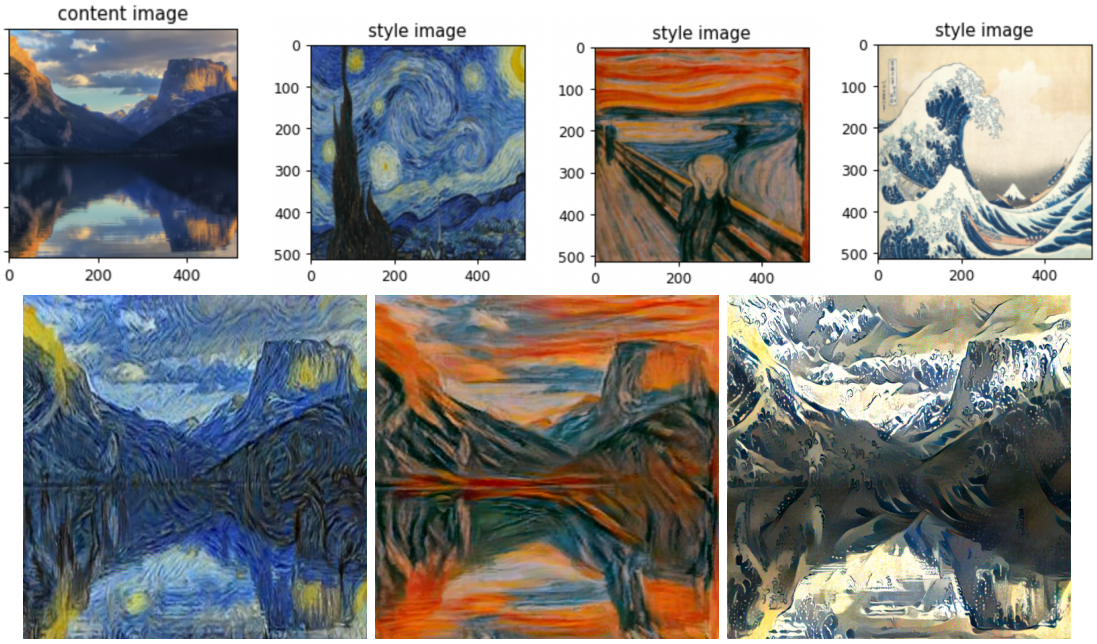

Fig. 7. **[VERSION 1]** A photo of mountains and a lake stylized. Image 1 on row 1 is the content image which is a scenery photo. Images 2-4 on row 1 are style images that are masterpieces painted by Vincent van Gogh, Edvard Munch, and Katsushika Hokusai respectively. Images on row 2 are results generated by our Neural Style Transfer implementation.

Fig. 8. **[VERSION 2]** A photo of a mountain with the display of its style and content images for a style transformation by applying the algorithm in Verison 2. The content image size is 4032×2480, the style image has the size 1200×807 and the result stylized image has the same size as the content image. The result is qualitatively similar to Gatys et al. [2] but with a much larger result in image size and generation speed.

REFERENCES

[1] OpenCV Library. https://opencv.org/.

[2] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2414–2423, 2016.

[3] Xun Huang and Serge Belongie. Arbitrary style transfer in real-time with adaptive instance normalization, 2017.

[4] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution, 2016.

[5] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit

Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.