Chinmay Tyagi: 57849174
Michael Auld: 88642494
Tycho Bellers: 21859892

Neural Style Transfer

Problem Statement

Humans are capable of creating intense visual experiences through art - what is the algorithmic process for computers to do the same? Our project tackles this interesting application of computer vision in a process known as image stylization. Is it possible to create unique images based on the style of already-existing art? For example, could I "Picasso-fy" a selfie of myself to create an artificially painted image, taking the style of one of his paintings? Recent advancements in biologically inspired vision models called Deep Neural Networks have made this possible - and could change the way humans perceive artistic imagery.

Previous Work

Image stylization relies on the simple idea that the content and style of an image are mathematically separable, through the use of convolutional neural networks (1). Neural style transfer was first explored in a 2016 paper by Gatys *et al*. When convolutional neural networks are pretrained on image recognition, it is possible to extract the representation of the ‘style’ and ‘content’ of an image, from the feature maps in intermediate layers in the network. An ideal neural network for our purposes was the VGG CNN, a 19-layer architecture pre-trained on image classification of over 14 million images. Image content is captured by measuring the relative pixel arrangement of an image. Image style is essentially a measurement of its texture - which is calculated by measuring the correlations of the feature maps with various image filters (2). This correlation captures texture information while disregarding the actual pixel values, making it a good representation of style. The image below shows style and content reconstruction at various layers.

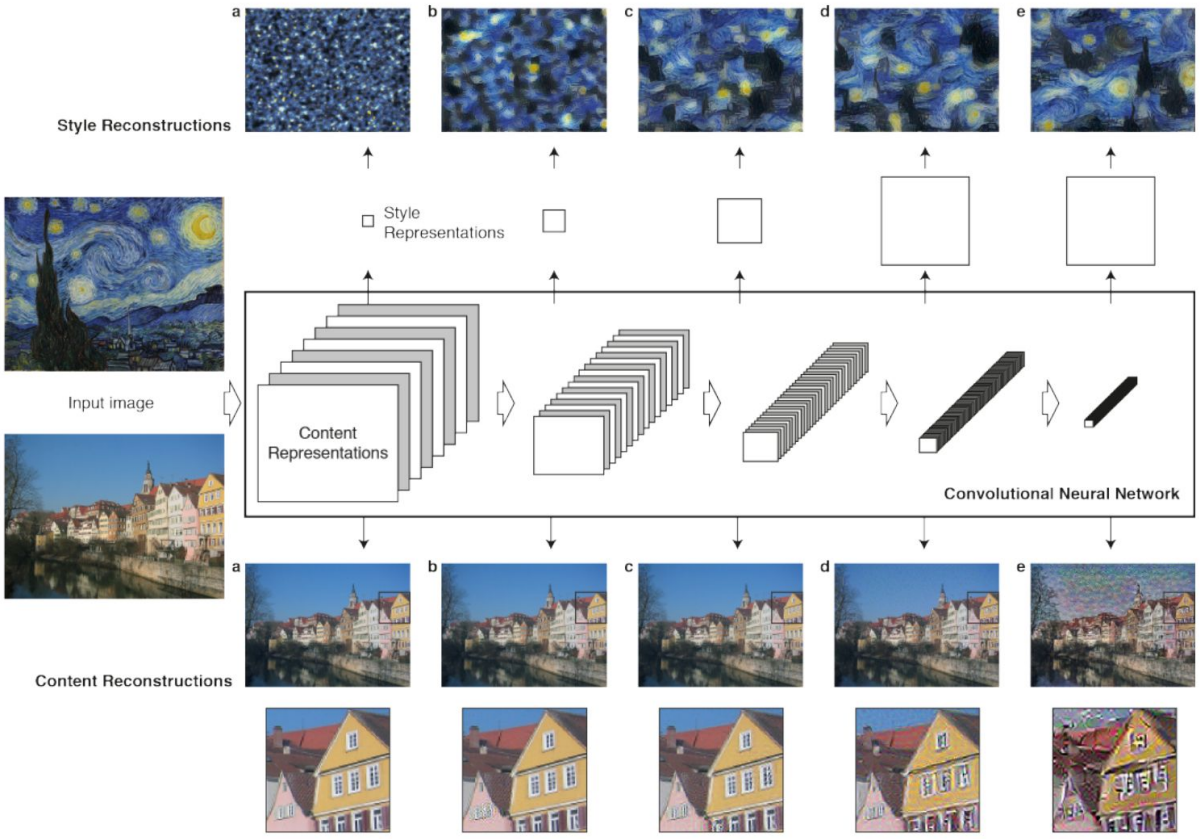


Figure 1 (1): VGG19 is passed an input image, represented as a set of filtered images at each processing layer. Letters A through E represent increasing convolutional layers. **Content**: At lower levels, content reconstruction was almost exact; At higher levels, high-level content was preserved as exact pixel information was lost. **Style**: As the number of layers used increases, style representation becomes more accurate with respect to input.

Once it is clear that content and style can be independently manipulated, our problem can be reduced to one of optimization. Given a 'target' content and style, we can use stochastic gradient descent to iteratively optimize a random image, say white noise, into the appropriate target representations (3). Thus, the entire process is actually quite intuitive:

1. Initialize ‘target’ content and style images, as objects of type VGG-19
2. Decide which layers are to be used for content and style, and extract target content/style matrices from respective input images.
3. Initialize final ‘output’ image (as random white noise)
4. Get content and style matrices of output image (‘current’ layers)
5. Compute style,content,total loss

$$Loss_{content}(x,\ y,\ l) = \frac{1}{2}\sum_{i,j}^{n}\left(X^{l}{}_{i,j} - Y^{l}{}_{i,j}\right)^2$$

Where x,y are input and output images, and X^l,Y^l their respective feature map representation at layer l for row i, column j.

Compute style loss:

$$Loss_{style}(x,\ y,\ l) = \frac{1}{2}\sum_{i,j}^{n}\left(G\left(x^{l}{}_{i,j}\right) - G\left(y^{l}{}_{i,j}\right)\right)^2$$

Where x,y are input and output images, and $G(x)^l$ and $G(y)^l$ their respective feature map representation at layer l for row i, column j. These feature maps are calculated by taking the correlation at each layer via a gramian matrix. (Gramian matrix calculates the dot product between a flattened i’th layer and flattened j’th layer)

Compute total loss with the following function:

$$Loss_{total} = \alpha Loss_{content} + \beta Loss_{style}$$

Where α, β are weighted constants for each representation.

6. Minimize total loss through stochastic gradient descent over a large number of iterations.
7. Pass total loss to output image, resulting in a stylized image

Once we were able to stylize images, we felt a natural next step would be attempting to stylize videos. With this addition, we hoped we would be able to generate, in essence, moving paintings.

In our research, we came upon various examples of stylizing video, with fascinating results. One group, Baumli *et al*., had optimized the process so effectively that they were able to achieve real-time style transfers in an iOS app. We also came across an article by Ruder *et al*., who were able to combine rapid video transfer with image 'stabilization.' In effect, they were able to improve frame-to-frame continuity so that the subjects and background in stylized video would have a consistent stylization throughout.

Though these examples were fascinating and produced incredible looking results, our team ultimately decided to go with a more straightforward frame-by-frame style transfer. We utilized the computer vision library OpenCV to convert videos into frames, and vice versa. From there, it was simply a matter of converting the video into an image sequence, stylizing those images, and converting those stylized images back into a video.

As mentioned previously, the Gatys paper was critical in our understanding of the implementation of neural style transfer. Tensorflow also published a guide for style transfer using VGG, which was of great help in our own implementation. Finally, we used a google blog on feature visualization to better understand gradient descent in application to images.

Decomposition of Project

Chinmay was in charge of implementation of neural style on a single image. He was the 'expert' on the VGG network, providing much of the preliminary code on retrieving layers, and calculating loss and gradient descent. Much of his work can as described in the checklist in the previous section.

Michael was in charge of applying style transfer to videos. He created a script to convert an mp4 video to frame-by-frame jpeg images.

Tycho was in charge of all things testing. He was also responsible for applying Chinmay's code to Michael's, and making sure this combined result ran smoothly. More details on Tycho's work can be found in the testing and results section of this document.

Our Coding Experience

Overall, working on this project was quite stimulating. Working with the VGG architecture was initially pretty tricky, but we soon got the hang of it, and it was exciting seeing the results of our work. We chose to implement this project in tensorflow, in part due to the vast amount of similar projects that also used this library. We had some compatibility issues with this library while performing gradient descent - it turned out that certain important functions were depreciated in Tensorflow 2.0. Although we were able to still find a solution, it was frustrating at times working with this library. Working with Pytorch instead of Tensorflow could have saved us some time and headache on this project.

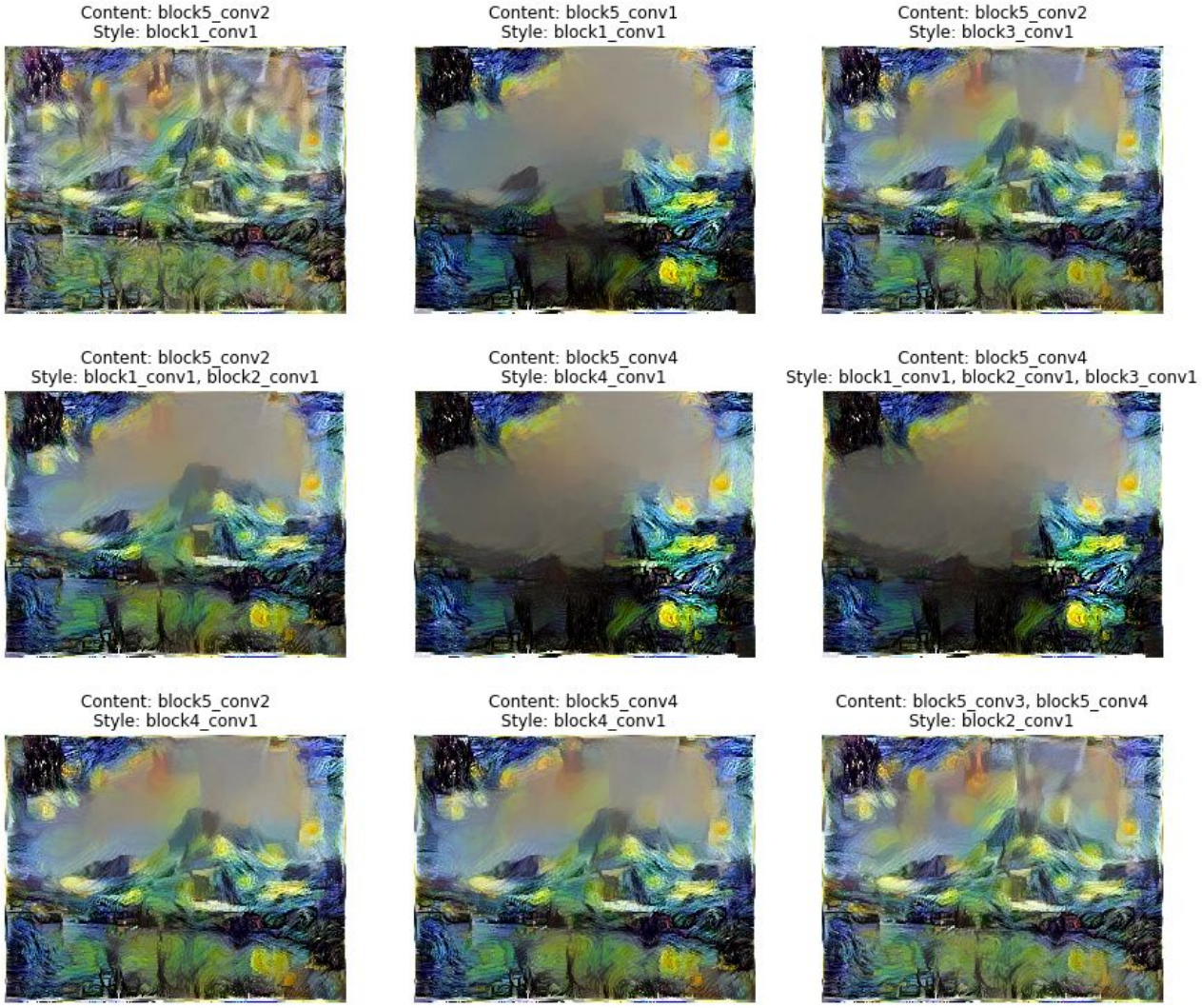
In the process of creating videos, we ran into an issue with Tensorflow's method of memory allocation (7). Due to the fact that we essentially had to create a new model for each frame of the video, tensorflow would attempt to allocate more for each model, despite the old ones being unused. This caused Tensorflow to run out of memory after a certain number of frames. To solve this issue, Tycho created a simple java program that would spawn a new Python process for every batch of frames. This worked very well and we were able to continue without issues.

Testing & Results

Given the artistic nature of this project, much of our evaluations were subjective, ie: Which images look the best? As a result, we used a combination of qualitative and quantitative testing to achieve the best results.

The first thing we had to do was determine which layers of the network to use for the content and style. To do this we ran multiple tests, trying out a different combination of layers each time.

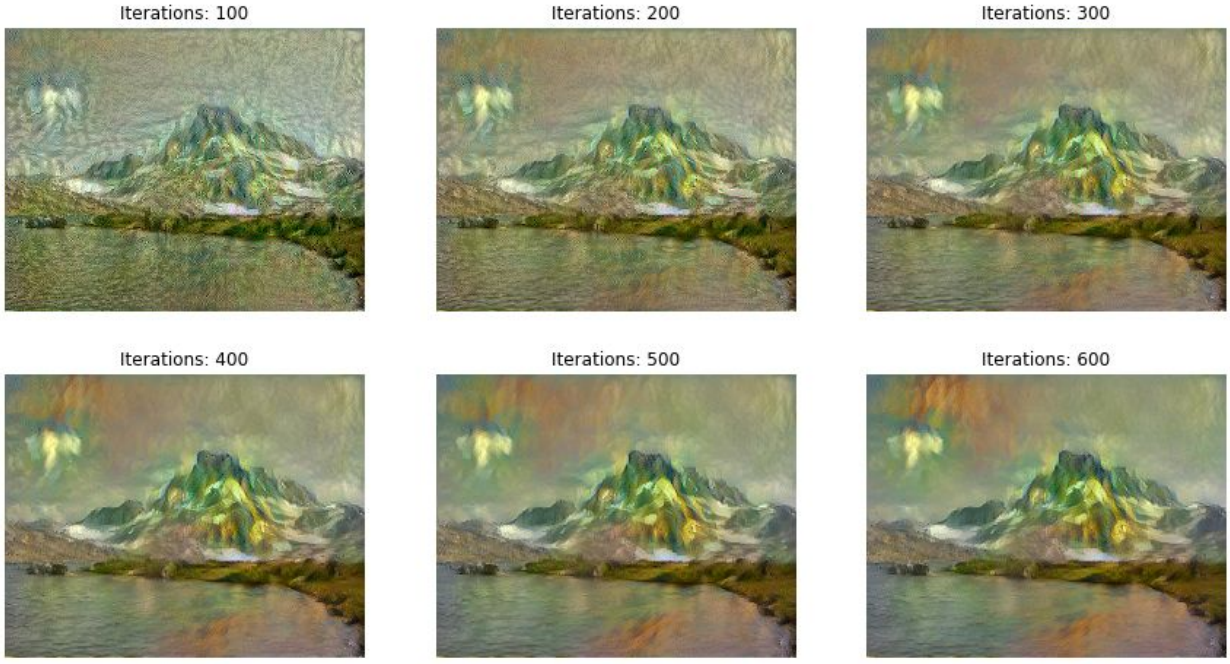
Below are some of the results. We found that using a single layer from the deep end of the network (block 5) for the content typically provided the best results. For the style, we found that using multiple layers from different levels of the network provided the best results. For these reasons, we decided to use the following layers for the rest of our tests: Content: block5_conv2, Style: block1_conv1, block2_conv1, block3_conv1, block4_conv1, block5_conv1. Below are some sample images displaying how the different layers affect the output image.



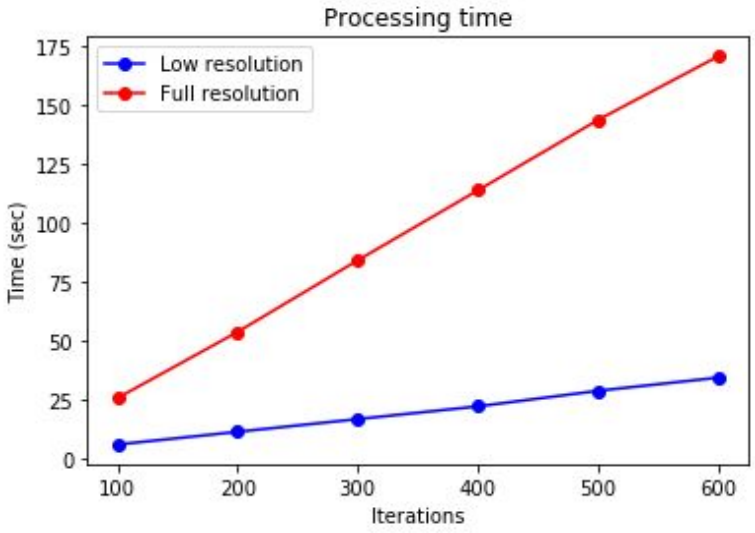
The next thing we explored was varying the values of alpha, beta, and variation weight. These values are used during gradient descent to compute the loss. Alpha and beta determine how much of an effect the style layers and content layers have to the resulting image, respectively. The

variation weight should limit the amount of total variation allowed between the input and output images (ie. how different the output image is allowed to be). During our testing, we found that modifying these values had very little effect on the resulting image. This was quite surprising as we had tested varying values by many orders of magnitude. There is a sample video provided in the repository, “weights_test_video.mp4”, that shows the effects of alpha, beta, and variation weight on the output image.

One of our initial goals for this project was to apply this style transfer to real-time video. Achieving this would require us to find an appropriate tradeoff for quality and processing speed. Below is a diagram showing the effects of more processing iterations on the image. The images begin to look acceptable at around 400 iterations. Unfortunately, running this many iterations took about 22 seconds on an Nvidia GTX 1080. Note that these images were already rescaled to approximately 500x400 pixels. For the full resolution of the image (approximately 1200x900), processing 400 iterations took nearly 2 minutes.



As expected, the processing time scales linearly with the number of iterations as shown in the graph below.



Clearly, this was not going to work for real-time video so we decided to shift our goal to creating pre-rendered videos instead.

Conclusions

It's quite remarkable that a convolutional neural network is capable of creating high-quality artwork and videos based on the simple idea that image content and style can be separated. Via manipulation of the VGG-19 architecture and using stochastic gradient descent to optimize the input images, we were able to create artificial artwork that could easily be mistaken for the real thing. While our image results were impressive, there is still much left to be desired in application to video. Style transfer requires a large amount of processing power, which makes real-time video more difficult. Neural style transfer has the potential to create amazing pieces of art, as well as improve human creativity.

Bibliography

1. Leon A. Gatys, et al. “A Neural Algorithm of Artistic Style.” 2016, https://arxiv.org/pdf/1508.06576.pdf
2. Olah, Chris, et al. “Feature Visualization.” 28 Aug. 2019, distill.pub/2017/feature-visualization/.
3. Huang, Eddie. “An Intuitive Understanding to Neural Style Transfer.” Towards Data Science, 12 May 2019, towardsdatascience.com/an-intuitive-understanding-to-neural-style-transfer-e85fd80394b
4. “Neural Style Transfer.” *TensorFlow*, www.tensorflow.org/tutorials/generative/style_transfer.
5. Manuel Ruder, et al. “Artistic style transfer for videos.” 2016, https://arxiv.org/pdf/1604.08610.pdf
6. John Sigmon, et al. “jsigee87/Real-Time-Style-Transfer.” *GitHub*, 2 June 2020, github.com/jsigee87/real-time-style-transfer.
7. Tensorflow. “Issue #17048: Tensorflow Gpu Memory.” *GitHub*, 2018, github.com/tensorflow/tensorflow/issues/17048#issuecomment-367948448.