Ryan Burgert 2017

Generating Seamless Textures via Neural Style Transfer

If you have ever played a 3d video game before, you'll have noticed that images are mapped onto 3d geometric objects called meshes. In 3d graphics, these images are called textures. If a texture is seamless, it means when we tile that texture, the borders of it aren't very noticeable. Seamless textures are a staple of 3d gaming graphics because storing very large textures is often infeasible, and artists are usually put in charge of creating them.

Above is an example of a texture that is not seamless. goo.gl/dKZDEV

Above is a game called Skyrim. Note that the stones on the path and the dirt on the ground repeat over and over again, but you don't see any seams.

This is a fairly labor intensive process, though. Obviously a photograph of, say, a bunch of pebbles, is extremely unlikely to be seamless before being edited. There are countless Photoshop tutorials out on YouTube when you search for how to create seamless textures. Creating a program to automate this process has immediate practical use in the game development industry, and empowers the layman to take a photograph of a patch of dirt in his backyard and obtain a usable, seamless texture from it with ease. I propose a new, automated solution to this process that can take arbitrary images as inputs, and output seamless textures with minimal human intervention.

(A side-note: in seventh grade, I attempted to create such a filter, you can still download it here: filterforge.com/filters/7198.html. This filter won me a free license of Filter-Forge due to its popularity, but the algorithm I am presenting in this paper blows it out of the water, and even pictures that one would expect to be impossible to make seamless, such as Lena, can be made reasonably seamless with this algorithm.)

This paper will discuss what neural style transfer is, a very brief overview of its history, and the details of my implementation of it. This paper will also briefly discuss my previous attempts at solving this problem, before the current results.

The last two pages of this paper contain a gallery of images I ran through my algorithm. There are three categories: original, dream and hybrid. The only reason there are empty spots because I ran out of credits on Floyd Hub, and was unable to continue GPU computations without paying a fee (all images should have three columns, but right now only two of them are complete).

What *Is* Neural Style Transfer?

In 2015, a paper titled "A Neural Algorithm of Artistic Style" was released by Gatys Et Al (arxiv.org/abs/1508.06576), describing an algorithm that could take any two photographs and apply one's style onto another. As inputs, it takes only two images: one for "content" and another for "style". This general idea, where you have two images (one with content and the other with style) became known as style transfer.

Content Style Result

www.zbdbxg.com/p/9720a26ce8df

Since then, many new research papers have focused on improving this algorithm. Some involve making it run faster or making it work very well on videos (see YouTube video: goo.gl/xUXL6a). My algorithm implements the original method described by Gatys.

Many apps have brought this technology to market, such as Prisma, Pikazo – or if you'd like to try it out for free, deepart.io, developed by the researchers of the original paper.

How does it work?

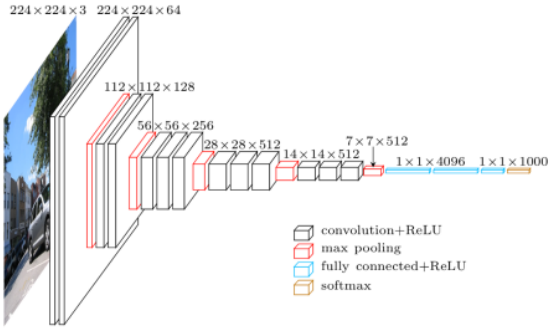
Abstractly, the algorithm works as follows: we create a random image the same size as the content image. This image might have random Gaussian RGB noise in it; we'll call this the input image. We also have two loss parameters regarding this input image that measure A. how close the input image's style is to the Style image, and B. how close the image's content is to the Content image. We apply gradient descent to every individual pixel on the input image to decrease both the content and style losses until we reach an acceptable convergence. The result is the Result image, shown in the above picture. An important thing

to note is that nowhere in the calculation of this image is a neural network trained to do anything: in fact, it uses a pre-trained VGG network for general image recognition to calculate loss and style.

Naturally, the next question becomes how to define these content and style losses. You'll observe that when you perform style transfer where the style image is the same as the content image, the result will be exactly the content image aka the style image. There will be no change. This is because the output image's style difference and content difference loss started out as zero and cannot improve from there.

To measure content and style, one performs the following calculations. Using a trained neural network capable of feature extraction (such as a VGG net, but with the fully connected layers removed), run the input image through the network and record the activations of each neuron in each layer of the network. Convolutional neural networks generally detect higher level features towards the output layer, and simpler features towards the input layer (such as edges and corners).

Style in this paper is defined a correlation matrix between all the different channels of a given layer in the VGG network, called a gram matrix.



www.cs.toronto.edu/~frossard/post/vgg16/vgg16.png

Intuitively, style loss is measured as the total element-wise squared difference between the input's style matrix (These calculations are detailed well in "A Neural Algorithm of Artistic Style", but I found Andrew Ng's explanation of these calculations on Coursera easier to understand: goo.gl/pEVqz9). Generally, as you use deeper layers for defining style, the changes in the image become larger. If you define style as correlations between channels in the first layer or so, for example. An excellent, short presentation given by Leon Gatys himself can be found here providing more intuition behind this algorithm can be found here goo.gl/CpHrS9. As the creator of this algorithm he explains it very well.

Finding the loss for content is actually irrelevant to my algorithm, but it's pretty simple. Finding the content loss is irrelevant to my algorithm because it's coefficient is set to 0, so content is not a factor in the gradient descent process. Instead of trying to match a correlation matrix between the activations the two images, you simply use a least-squares fit between the pixels elementwise. This preserves specific spatial properties of the image (like in the doge example, where the eye is positioned in the result is the same as the content image's doge, while items like the moon in the Starry Night painting are nowhere to be found).

How does my algorithm work?

My algorithm uses the original image as the style, and in 'dream' mode it uses random noise as the input image, while in 'hybrid mode' it uses the original image as the input image. Below is a simplistic representation of what is happening:

Hybrid mode:

(origin)→  →(shifted)  → (stylized) 

Dream Mode:

 →(stylized; completely hallucinated)→ 

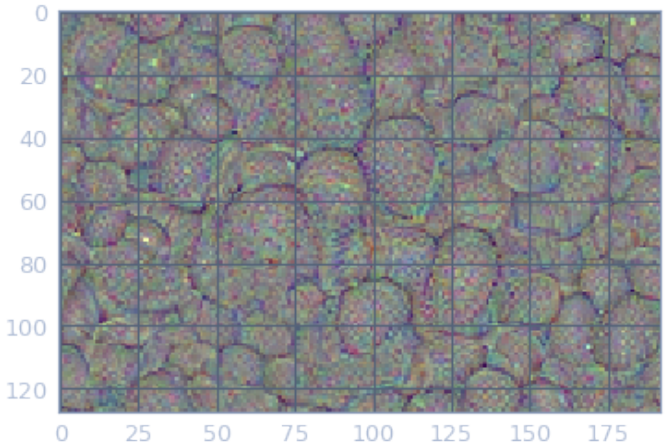
The above diagram is a bit oversimplified, because we shift each iteration of the gradient descent process, so as to avoid creating new seams. (If we fully stylize the image without moving it as we stylize it, it will get rid of the old seams but create new ones in its' place. This is undesirable, so we don't apply many iterations of gradient descent before shifting the image a bit more, exposing the new seams we made from the previous iteration.)

In dream mode, we start out with random noise (raised to a very high power so we have only a few nucleation sites from which the rest of the style grows) and apply style to it. We might apply one or two iterations of gradient descent, then shift the image diagonally by 10 pixels. Advantages of this mode are that we can get an infinitely many significantly different textures from a single source photo. Disadvantages, however, can be seen when the style of the image fails to capture its essential properties, such as the clover dream example in the image gallery (all the leaves are messed up). Note that dream and hybrid are terms I made up for the sake of talking about these methods and exist only in the context of this paper. Dream mode would not be capable of reproducing faces such as in the last three examples on the bottom of the gallery such as Lena's face.

In hybrid mode, we start out with the original image, but essentially apply the same process to the images that we do in dream mode, where we shift the image while applying style to it repeatedly. I've created time-lapses so you can see this process in action (originally I had time-lapses for all of the examples but something went wrong during the file transfer and now here are the two of them left): imgur.com/AsNYmfh, imgur.com/tTRtguu, imgur.com/4EvDn7Z.

Implementation Details

My project was implemented in Python3, using PyTorch. I ran all of my tests on FloydHub; a cloud service offering GPU support for free for a few hours. The code for my algorithm is based on code from a PyTorch tutorial for implementing neural style transfer, which was then modified to implement my algorithm. You can get my code on GitHub: github.com/RyannDaGreat/Seamless.git. I also posted my original, less successful attempt at solving this problem in the repository that produced images such as this:



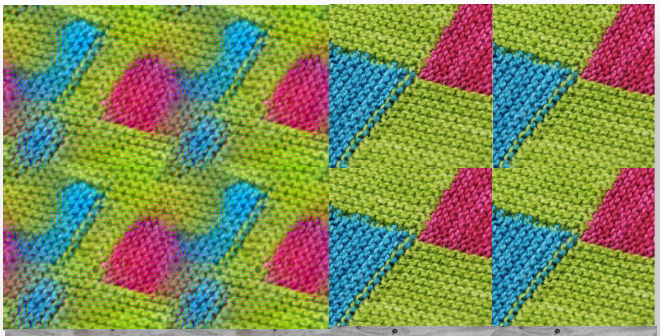
(It was also seamless, but wasn't nearly as useable. That picture is supposed to be pebbles.)

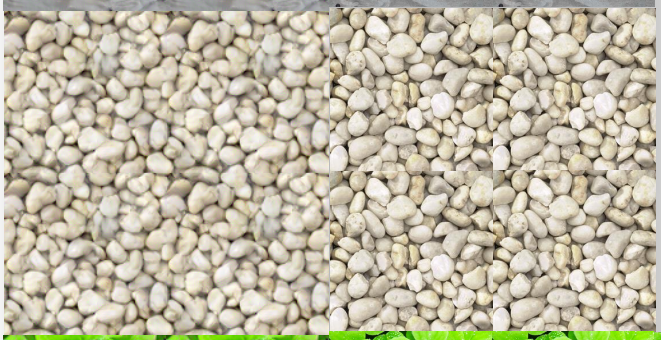
Conclusions

Neural style transfer can be used for much more than creating awesome Instagram photos. It can be used for stitching images together in ways that are very convincing, as well as synthesizing entirely new images from scratch. Through this project I learned how to effectively use PyTorch, a deep learning framework for python, and also learned how to use online cloud services such as FloydHub for computational speedups.

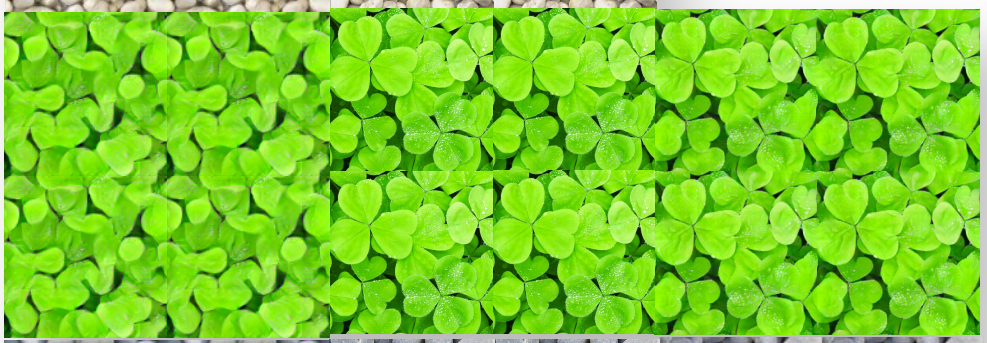
Dreams Originals

Image Gallery





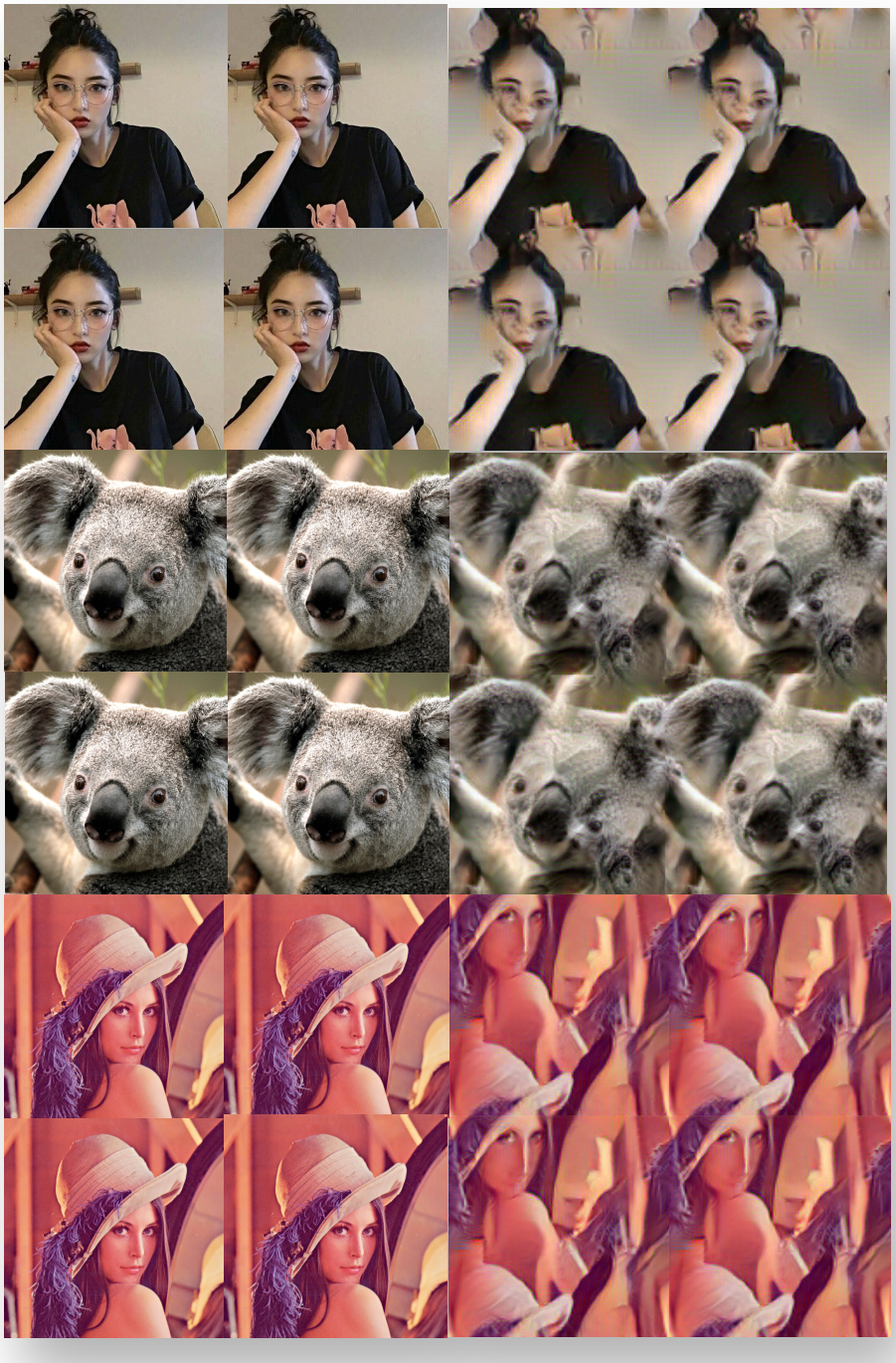
Hybrids



goo.gl/MdwjR9

imgur.com/a/OBA6g

Originals Hybrids



goo.gl/ZWUMPb

goo.gl/YWDap7

goo.gl/WStSK8