Shawn McClain

Style Transfer and Image Analysis: Milestone Report 1

Problem Statement:

There are many uses for neural networks in the modern world, but one that's not thought of as often is the impact machine learning can make on art and graphics. Thus far, software products like Photoshop, Lightroom, and Illustrator have transformed the tools of many graphics designers, directors, photographers and made their work that much better. But with the AI revolution, art is set to make another leap forward with technologies like augmented and virtual reality. This project attempts to combine data and neural networks with art to uncover new horizons for the art world.

Data/ Solving the Problem

One technique I'd like to dig into with this project is Style Transfer. Style Transfer takes the content of one image and the style of another image to make a third image with blended values. The data for this project can come from the internet. I will need to collect some style images and some content images for testing. In terms of model training, I can look for a pre-trained model and incorporate that into my use case. I can also train a new model, but this might be counter-productive if we can build off someone else's model.

Adopting a pre-trained model saves the need to store and train billions of images and the processing power to process all of the images. By saving this extra space, my dataframe can be my style and content images, and I can host these locally.

Clients

Style Transfer has many potential use cases. In the consumer world, it can be used to apply styles onto clothing and artwork and sold to the general public. On another level, Style Transfer can be used in virtual and augmented reality technology, to recreate environments in the real world in virtual worlds. Recently augmented reality was popular with Nantic's PokemonGo game which used real maps and places to augment a game over top of those places. With those real places as the 'content', the style could be graphically designed, generated digitally or grabbed from other places. It really has a lot of potential in a variety of entertainment and business ventures.

Deliverables

For the casual reader or recruiter who does not want to sift through data, I will provide a 'lite' version that has many visualizations and descriptions for my process on each section of the project. I'll also provide a full notebook that consists of all of my code along with annotated notes for my thought process at each part for those interested. As output, I'll have style transferred images that look aesthetically pleasing.

Initial Image Analysis

For my project on transferring style onto content images, I first wanted to get a good grasp of what others have done and been successful with, when taking on a similar project. From listening to others, it seems that the best results for style photos are paintings and artwork as opposed to photographs. This is because the style attempts to pick up patterns in brush strokes and the details in between the lines. Content images that are photographs tend to be better because they have clearly defined boundaries. Something like an abstract painting usually doesn't have a clear interpretation of a figure, which makes it

abstract. Also, many projects used a pre-trained VGG model for the best results. VGG is named after the Visual Geometry Group at Oxford University who achieved very good classification results with their convolutional neural network on the ImageNet Database, which is a massive repository of internet photos.

After researching what other style transfer projects used as their foundations, I sought out a reasonable group of style photos to experiment on, from pulling art on the internet that I really liked. After initializing my Jupyter Notebook with the correct libraries, I first copied and pasted my style urls into a list. After creating this list, I defined two functions that could show and read images from paths on the internet. Using the requests.get method, this function pulls in images, decodes them, opens the images, and then converts them to numpy arrays for manipulation. I then made another function that could save these files to a local folder so that I wouldn't have to pull from the internet each time I wanted to use a picture.

From other image projects, understanding the format of image data and the corresponding pixels seemed like a good place to start. I started with my first style photo of an ocean painting of Thailand and grabbed the image dimensions which gave the height, width and number of channels. From here, I flattened my image array and made an intensity plot of the initial image. An intensity plot has pixel values from 0 to 255 on the X-axis and the density of those pixels on the Y-axis. The intensity plot is used to gauge how bright or dark an image is with a density peak towards 0 signifying a dim image and density peak towards 255 signifying a bright image. For the initial image, the overall density peak occurred near 200, which meant the image was quite bright.

In the Thailand image, I then separated the color channels on the image by reds, greens, and blues and showed the image in its separate color channels. Certain areas of the image were filled in deeply with a color but not the others. The boat in the image had deep reds, and when plotting the separate channels, blues and greens were absent on the boat in their plots, but on the red plot, the boat was filled. Anything white on the original image appeared as fully filled in by all 3 colors on their splitted plots.

Moving forward, I wanted to make a dataframe of my style images to prepare the images for analysis. I imported ImageCollection from skimage which collected the images from my local folder and made them accessible again to the notebook. After showing the images with subplots, I flattened each image into a single array per image and saved them into a list. I then converted this list to a dataframe and renamed the only column “Pixel Array”. A flattened image yields three times the number of pixels of the image, because each image has 3 channels (RGB). Each of these channel pixels corresponds to a value between 0 and 255. In order to separate out the red, green and blue channels of each picture, I pulled the third values from the flattened arrays and saved each third value into the correct channel. I also saved the height and width of each image at the index 0 and 1 of the shape.

From this dataframe, I was able to do some visualizations to compare my style images against each other. I first started out with a bar plot of the height and widths of each image. After this, I used the 3 separate color channels to plot a combined intensity plot for each image. This was a similar process to the plot on the Thailand Ocean picture except each channel is separated in the plot, so we can see the peaks of each channel on each image. Using the Stats package, I found the mode of each array on each image channel. This gave me the peak density location of each channel on the 0 to 255 scale. This data gave me the most common shade of red, green and blue on each image. Overall, the average red shade of the images was very bright at 224.75, while the average blue and green shades were more moderate at 158.68 and 158.94.

Machine Learning/ Style Transfer

Moving on from the image analysis, I now have a good idea as for what may images look like in terms of their pixels. In order to implement style transfer to our images, we now need to pick a style image to extrapolate style from and a content image to extrapolate content from. Those 2 images will then be merged together to create a 3rd image byproduct. The style image will be an attempt to grab brush stroke patterns and colors associated with the artist's style, while the content image will be an attempt to grab the important boundaries of figures in the content image and preserve those. For the content image we can just minimize the mean squared error between the content and output image. In order to capture style, we need to implement a more complex process of extraction with a gram matrix. The gram matrix is calculated using matrix multiplication to represent numerically the style at different levels of the image. We will minimize the difference in these calculations from the style image to output image.

Along with content and style, we are also concerned about noise that could occur in our generated image. In order to reduce noise we can shift pixels that are next door to each other on the generated image and minimize their color difference to each other. So for our combined loss function that we want to minimize, we want to minimize content loss, style loss and image pixelation together.

For our function, we can use a pretrained VGG model for the best results. VGG is a deep convolutional neural net built on the Keras framework and was developed by Oxford's Visual Geometry Group. The VGG was trained on thousands of ImageNet images and is able to classify images with 92% accuracy. Because this model is pre trained for classification, we do not need to retrain the model. The VGG model uses Relu as its activation function which stands for a

rectified linear unit. Relu is simple to compute by turning negative numbers to zero and keeping positive values as they are. This is calculated after processing between layers in our neural network. After we achieve a result passing the content and style images through our network, the network performs what's known as "back-propagation" in order to minimize the loss function on the next iteration of the images. Back propagation moves against the flow of the network to adjust weights based on the loss achieved from our loss function. In the end, this has the effect of progressively minimizing the loss function through each iteration.

In order to feed information through the VGG model, we need to pull the layers that we need from the VGG model, because we don't need to use all of them. Five layers are pulled for calculating the style loss and one is pulled to calculate the content loss. The layers are fed through the loss functions by their features and then a gradient object is created for gradient descent. The loss and gradients are packaged into a Keras Function object for evaluation.

Normally with Keras algorithms, there's a built in evaluator which passes in than loss and gradients, and evaluates both. The algorithm I used for optimization is not on the Keras framework and is called the L-BFGS. The L-BFGS algorithm stands for the Limited (memory) Broyden–Fletcher–Goldfarb–Shanno algorithm after the scientists who discovered the algorithm. L-BFGS starts with an initial estimate of the optimal value, and refines that estimate with the derivatives of the function. The derivatives are used to identify the direction of steepest descent, and to compile local minimums of the descent. Because L-BFGS is not on the Keras framework and is on Scipy, I built an evaluator which saves and compiles the loss and gradients to be learned upon.

We now have everything we need to feed the Keras tensors through the VGG model and optimize the loss and gradients through the L-BFGS algorithm. After each iteration, the model minimizes the style, content, and variation loss and produces a picture from the pixel values. The produced picture is then fed in again as the generated image and incrementally becomes a blended picture of the style and content picture. The model is run through a GPU to accelerate

computing time, because each iteration would take approximately 10 minutes to complete with just a CPU.

After testing out the neural net, there are a few things that can be tried and altered moving forward that may affect the results achieved and the goals desired from the neural net. The weights of the style, content, and variation loss functions, relative to the total loss, can be altered to penalize loss in one component more than the others. A strong penalty on content could result in an image with firm boundaries, and may be less cartoonish. Instead of the L-BFGS, another algorithm could be used for gradient descent, like the popular Adam optimizer. And finally, the number of interactions can be adjusted. We would assume the more iterations will produce a better byproduct of the style and content images, but there may be a point with diminishing returns.

Model Tuning

For my model tuning, I tried adjusting the style, content, and variation weights on the same set of style and content pictures while holding other variables constant. Usually with machine learning, we may want to do a grid search, but because we are dealing with art, I want to do the grid search manually to decide aesthetically which weights gave the best results. To begin, I started with standard weights of 1 for all of the different types of losses. For my style and content pictures, I kept those constant for every run. For my content photo, I used a picture of an Icelandic landscape and for my style photo, I used an abstract mountain layers painting. The painting features a dripping appearance, which I thought may add some cool effects to the final result.


I first tuned the content weight by holding the number of iterations at 15, the style weight at 1 and the variation weight at 1. The different content weights were 0.01, 0.025, 1 and 10. After testing all of the different weights, there was little difference in the results after 15 iterations each. One of the subtle differences was the lower content weights tended to save the shadows of the content picture a bit more, which seemed counter-intuitive. I went with a weight of 0.025 because I like that it preserved more of the shadows in the content photo.

Moving on to the style weight, I repeated 15 iterations again and went with the content weight of 0.025 and the variation weight of 1. The different style weights I tested were .01, .1, 1 and 10. After testing all of the combinations, the lower style weights once again seemed to preserve the shadows a little more. Otherwise, I was surprised to find that there was little difference in the results. I went with a weight of 1, because not much of a difference was made.

For the variation weight, I repeated the 15 iterations again and went with the style weight of 1 and the content weight of 0.025. The different variation weight combinations that I tested were .5, 1, 2, 3. The variation weight of .5 produced an error and did not produce an output image. When I increased the variation weight from 1, 2, and to 3, the output became more and more granulated and produced more lines in the output image. Increasing the variation weight seemed to add more style details, but ultimately made the output less appealing.

Using the trained weights, I then tried different numbers of iterations on my images. The iterations I tried on my images were 5, 15, 50, 100 and 200. Less iterations produced an output that was more similar to the original content image, because I used the content image as the initial generated image. More iterations in turn sacrificed some of the content image for more style details. Overall, the higher iterations erased much of the shadows of the content image and made an undesirable result. My best results for this particular combination of images was 5 and 15 iterations. Depending on the goal of the output, more iterations doesn't automatically guarantee better results.

Using my trained results, I then applied the same weights and iterations to other style images. For one test, I used the same content image with a Japanese painting, with intricate detail. For another test, I used the content image with a 2D abstract image, with less detail, After running through each test, the 2D abstract image produced a better result that looked a bit more natural. The test with the Japanese painting added more detail in the shadows, which didn't look natural and caused some aberrations in the sky of the output image.

The 2D abstract image made a smoother output that could be more passable as a piece of digital art.

Conclusion

In conclusion, the neural network was successful for transforming a content image and a style image into a piece of art. The pre-trained VGG model, which is a convolutional neural network that has been trained on thousands of images, gave a good starting point for identifying shapes and patterns. For content, I identified a loss function with the mean squared error of the difference in pixel values between the content and generated image. For style, I used a gram matrix to extract style patterns in different spaces of the style image, and minimized the loss of those style patterns in the generated image with the mean squared error of the difference between the gram matrices. Variation loss was also used to produce a smooth output image that blended pixel borders by shifting neighbor pixels and minimizing their difference. Through backpropagation with the L-BFGS optimizer, the losses are minimized with each iteration, and the processes are repeated on each generated image.

Through manually testing and optimizing different weights, there weren't big differences when adjusting the weights on the output images. Adjusting the variation weight and number of iterations did produce a difference in my outputs, but more interactions did not necessarily produce a better output, even though loss was minimized further.

There is still much that could be tested and improved upon in the future regarding style transfer. Style transfer has promising potential to make art easier and become a tool for digital art. There may be other loss functions that can be minimized to help yield better results. Also, if the model could identify parts of the image to not include in the style transfer process, it could allow for more customization and better results for artists. It'll be interesting to see if style transfer is incorporated in new software applications like Adobe Photoshop. It will also be interesting to see if style transfer can be applied to augmented reality environments to improve different augmented reality experiences.