EE8601 Project 2: Image Style Transfer

Implementation of *Image Style Transfer Using Convolutional Neural Networks* by Gatys et al

Alegre, Edwin Philippe, EIT
Department of Electrical, Computer, and Biomedical Engineering
Ryerson University
Toronto, Ontario, Canada
edwin.p.alegre@ryerson.ca — 500552920

***Abstract*—The aim of this report is to document the procedures, as well as the theory gained from the implementation of Gatys' paper - *Image Style Transfer Using Convolutional Neural Networks*. This paper outlines the motivation, a brief top level theoretical outline on the workings of the above mentioned paper, the implementation (including dependencies, code, etc), as well as outcomes related to the directed studies course, EE8601 - Fundamentals of Computer Vision and Deep Learning.**

***Index Terms*—image processing, convolutional neural networks, style transfer learning**

I. Introduction

The visual arts has been a medium that conveys ones sense of wonder and abstract viewpoint of the universe to its patrons. From Vincent Van Gogh, to Picasso, to the Group of Seven, an artist wields their brush in their own manner, showcasing their own personal genius in the use of colours, strokes, and realism to convey how they see a certain subject. It is no wonder that certain artists have become synonymous with their own personal style. Van Gogh is known for his vibrant colours ad swirled strokes that are evident in his great works, with "Starry Night" as the perfect example. Picasso brings out the abstract use of shapes in his work. Monet captures the vibrant realism of his landscape masterpieces. Artists have tried to copy that style and take years to reach a level that was achieved by these masters. Recent technological advancements have allowed for the use of *convolutional neural networks* (CNN) to achieve great feats in the field of computer vision and engineering. An example of image style transfer can be seen in Figure 1.

This paper will focus on the implementation of *Image Style Transfer Using Convolutional Neural Networks* by Gatys et al. This paper utilizes convolutional neural networks to extract the *style* of one image (the way an artist paints) to an image with a different *content*. In the world of computer vision this is known as *style transfer*. This is done with the use of CNNs that extract the style of one image and combines it with another.

Figure 1. An example of image style transfer. The top left photo showcases Stanford University's Tower and is known as the *content image*. The top right photo is that of Van Gogh's Starry Night and is known as the *style image*. By applying the image transfer technique, a synthesized image of the two input images is formed and shown at the bottom

II. Dependencies

In order to successfully implement this project, the following technical dependencies are needed.

A. Python (ver. 3.7.7, 64-bit)

Python is the primary prototyping language used for machine learning, computer vision, and image processing applications. As such, it is the programming language used to implement this project.

It is important to note that Python will be supplemented by Tensorflow and Keras (both tools that correspond with easily implementing machine learning models within the Python environment).

It is highly recommended that Python is supplemented with *pip*, a Python package manager that makes it easier to install libraries to use within Python. An ideal IDE that the reader is comfortable with using is also suggested.

B. Tensorflow 2.0

Tensorflow is an open source machine learning library, created by the Google Brain Team, that is capable of interfacing with Python. Tensorflow is widely used in academia and the industry, which makes it a viable platform to use for this project. Specifically, this model relies on a CNN model known as VGG19 which needs to have been trained using a specific data set known as *ImageNet*. Luckily, Tensorflow has already implemented this exact model and set for use.

Tensorflow is capable of training and running deep learning models. The most recommended API to use when dealing with neural networks on Tensorflow is called Keras. It is worth noting that Tensorflow may be used locally (on the host machine) or with a cloud computing service such as Amazon Web Services (AWS) in order to run the models utilized. While it is entirely possible to run Tensorflow locally, it may be time consuming as the vanilla Tensorflow package utilizes the host CPU in order to process the model. This method takes approximately 1 to 2 hours to run the code. The time used to run the model can be optimized by the use of cloud computing or though outsourcing the processing to a local machine's GPU. Typically, GPU outsourcing is achieved through the use of CUDA.

C. CUDA (optional)

CUDA is a parallel computing platform created by NVIDIA that enables the use of a local machine's GPU for general purpose computing. Specifically, the use of CUDA in this project allows our model to return the results at a much quicker rate due to the use of the local machine's dedicated GPU instead of its CPU. CUDA relies on a variety of dependencies in order to interface with Tensorflow. The following dependencies must be met before CUDA can be utilized:

- NVIDIA GPU card with CUDA Compute Capability of 3.5 or higher
- NVIDIA GPU Drivers - CUDA 10.1 requires 418.x or higher
- CUDA Toolkit - Tensorflow 2.1.0 requires CUDA 10.1
- CUPTI - Installed along with the CUDA Toolkit
- cuDNN SDK

D. Google Colab (optional)

Google Colab is a notebook service that allows for the creation and execution of Python code within the Google interface. By uploading Python code to a linked Google Drive, code can be run online. Furthermore, Google Colab allows for free GPU utilization, which is extremely useful if CUDA is not a viable solution due to the hardware constraints of the local machine.

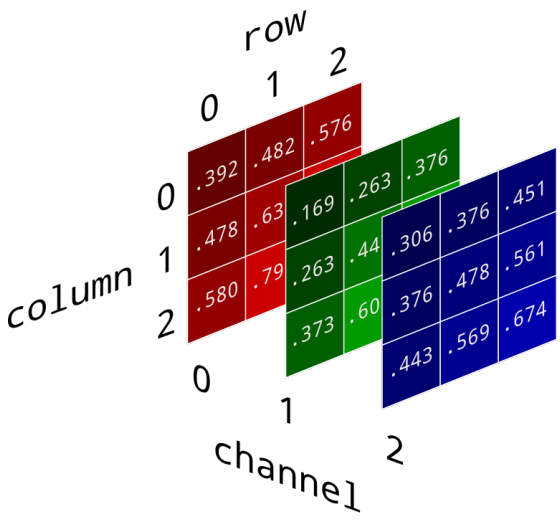


Figure 2. Visual representation of an RGB image

III. Background and Related Work

In order to properly understand how Gatys' algorithm works, a background in various concepts in Image Processing and Deep Learning is needed. The main concepts are discussed below.

A. Image Representation

In the realm of engineering, images are defined as a matrix of pixel values of size MxN. Typically, the pixel intensity is represented by 1 *byte* with a range of [0, 255]. Greyscale images are restricted to this range and are represented by a single matrix. Color images are typically represented as a 3 channel matrix that corresponds to red, green, and blue, which is why colour images are known as RGB images of size MxNx3. An example of an RGB image is show in Figure 2.

B. The Convolution Operation

The *Convolution* Operation is the backbone on which Convolutional Neural Networks are built on. Convolution is an elementary operation in the field of electrical engineering, specifically in signal processing and image processing. Convolution works by sliding a *filter* or a *kernel* of a pre-specified dimension (typically 3x3) over a matrix. Each element in the matrix is multiplied by the overlaying kernel value and all of the products are summed to produce a single output. Typically, convolution involves flipping the matrix, especially in the field of signal processing. However, in the case of computer vision and CNNs, the matrix remains unflipped. This specific case of convolution is known as *cross correlation*

Specifically, convolution in CNNs is used to extract features from the input image. By using small filters of input data, convolution maintains the relationship between pixels by

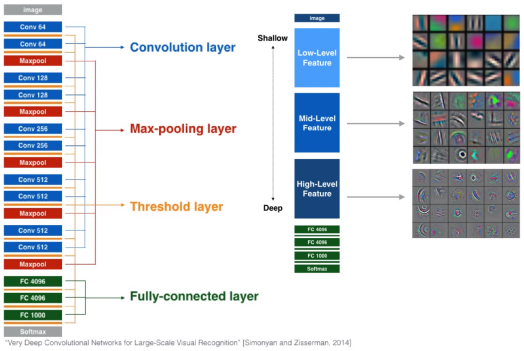


Figure 3. A generic CNN based off the VGG19 network that shows all the typical layers used. The smaller figure on the right side displays what types of features are captured at each convolutional layer in the CNN

identifying image features. The resultant matrix formed by the convolution within a convolutional layer is known as a *feature map*. This feature map is a 3D matrix of size N_{width} x N_{height} x $N_{convolution}$ where N_{width} is the width of the input image, N_{height} is the height of the input image, and $N_{convolution}$ is the number of filters in the layer.

C. Zero-Padding

Zero Padding is another concept widely used in signal and image processing. Zero padding surrounds the input data by a border of zeros such that the borders of the input image can also be captured int eh feature map during the convolution. This also allows for preservation of the dimensions when the input data is convolved with a filter.

D. Convolutional Neural Network, CNN

The *Convolutional Neural Network* is broken down into a smaller set of functions of made up of layers, where each one of those layers are a neural network in of itself. Specifically, the CNN is comprised of 3 main layers, those being: *the convoluton layer*, *the pooling layer*, and *the threshold/non linear layer*. Depending on the CNN architecture, there may be more layers involved in the CNN, but these 3 layers remain constant. The CNN works extremely well for image based problems. Specifically, the architecture of the CNN allows it to learn a variety of features, from low-level features (lines, edges, etc) to mid-level features (combined lower level features that showcase patterns) high-level features (truly sophisticated and complex features/textures at the top level). The initial layers usually learn the low-level features, with each progressive layer learning higher-level features. An example of a CNN is shown in Figure 3, specifically showcasing the CNN used in this project, the VGG19 network. THe VGG19 network contains 3 fully connected layers and 1 softmax layer, which are not utilized in this project. Hence it will not be discussed, however each utilized layer is explained in more detail below.

1) The Convolution Layer: The convolution layer is the most important layer in the CNN and is the name sake of

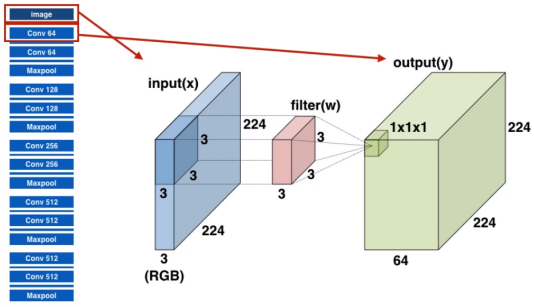


Figure 4. A graphical example of how a convolutional layer in a CNN works. Note that the actual convolutional layer is comprised of a tensor with size 224x224x64

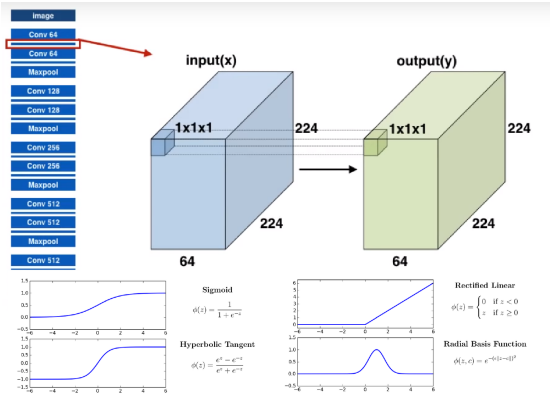


Figure 5. A graphical representation of the threshold layer in a CNN. The threshold is based off the activation function used in the network. The 4 most common activation functions are shown at the bottom of the figure

the CNN. In general with the case of images as the input, the convolution layer is a 3D convolution. The input image is convolved with a kernel where the output is the basis for the feature map. In the case of the VGG19, the first layer generates as 224x224x64 feature map, which is due to 64 different kernels being used to produce the different depths of this layer. The first stage convolution layer of the VGG19 network is shown in Figure 4.

2) The Threshold/Non Linear Layer: Following the Convolution Layer is the *threshold layer*. This layer introduces a non linear component to the network, where the most used activation function is normally the *rectified linear unit (ReLU)* but other functions such as the hyperbolic tangent can also be used. This layer is the same size as the previous feature map and each element corresponds to the same element space of the feature map before it. This can be seen in Figure 5

3) The Pooling Layer: The *pooling layer* is essentially the layer that downsamples its input layer in space. For example, the first pooling layer in the VGG19 network takes an input from the threshold layer of size 224x224x64 and applies *max pooling* onto it. Usually, pooling is done by sliding over the input layer with a 2x2 kernel of *stride* 2, meaning it moves two pixels over instead of the usual one.

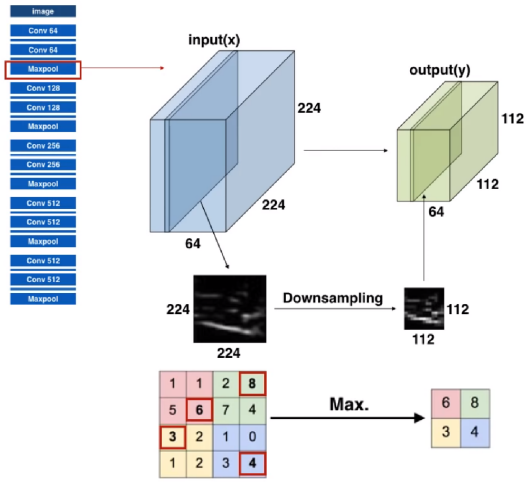


Figure 6. A graphical representation of the pooling layer in a CNN. Note that this layer essentially downsamples the input feature map. The figure at the bottom shows how max pooling is achieved, where a 2x2 kernel with a stride of 2 slides over the input matrix and translates the maximum value in the viewing window as the new downsampled pixel in the resultant pooled feature map.

This kernel performs a max function over its window, choosing the maximum element to represent that feature area in the downsampled feature map. There exists other pooling methods such as the *average pooling* or *sum pooling*. Gatys states that average pooling yields better results, which is why it is utilized as the pooling method for this implementation. As specified earlier, the first pooling layer in VGG19 takes a 224x224x64 input feature map and returns a 112x112x64 downsampled feature map. In all cases, pooling helps make representations become approximately invariant to small translations to the input. Invariance to translation causes the values of most of the pooled output values to stay the same despite the input being translated by a small amount. Invariance to translation is particularly useful if the presence of the feature is more important than the location of it. A sample of the pooling layer can be seen in Figure 6

E. VGG19 Network

The Visual Geometry Group Network (VGG) is the winning submission of the 2014 ImageNet challenge. This network has two versions, the VGG16 and VGG19 networks. The number symbolizes the amount of layers in the network, with VGG16 having 16 layers and VGG19 having 19 layers. VGG19 is used for this application and consists of 16 convolutional layers and a rectified linear unit (ReLU) as its non-linear threshold function. The convolutional layers are separated by 5 pooling layers and ends with 3 fully connected layers. As stated previously, the fully connected networks will not be used for this project. Specifically, the content representation will be matched using 'conv4_2', and the style representation will be matched using 'conv1_1', 'conv2_1', 'conv3_1', 'conv4_1', and 'conv5_1'. Each layer will have a weight of 1/5. The graphical representation of the VGG19 network can be seen in Figure 7

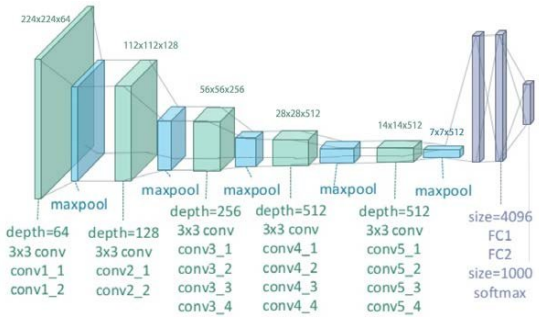


Figure 7. The VGG19 Network. Notice the last 3 layers of the network, known as the fully connected layers. While these layers are part of the VGG19 network, they will not be utilized in this implementation

IV. Deep Image Representation

The main takeaway from this paper is that **the representations of the content and style are separable and thus, both representations cna be manipulated independently to produce an output of the user's liking**. Using this finding, it was determined that the overall loss function is a linear combination of the two individual loss functions. Thus, by regulating the individual loss functions, the emphasis on the resultant image can be based on either the reconstruction of the content or style input. This will be discussed in further detail in the upcoming sections.

A. Content Representation

Given an input image $\vec{x}$, this input image is passed into each layer of the CNN, providing a feature representation at each layer, F^{l}. Gradient descent can be applied to a white noise image in order to visualize the image information that is encoded at every layer of the network. This provides another image that can be used to match the feature responses of the original image. These responses in a layer, l, can be stored in a matrix with the generalized form of $X^{l} \in \mathcal{R}^{N_l \text{x} M_l}$, where each layer contains N_l distinctive filters and thus, has N_l each of size M_l, where M_l is the height times the width of the feature map. From this, the activation of the i^{th} filter at position j in layer l can be symbolized as X^{l}_{ij}

These two images can be called the *content image*, $\vec{p}$ with feature representation (content features) $\vec{P}$ generated in Layer 4_1, and the *generated image*, $\vec{x}$ with feature representation (content features) $\vec{F}$ generated in the same layer.

Thus, the squared-error loss between the two feature representations can be defined as

$$\mathcal{L}_{\text{content}}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} (F^{l}_{ij} - P^{l}_{ij})^2$$

From the derivative of this loss function, defined below, the gradient descent with respect to the generated image, $\vec{x}$, can be computed using standard back-propagation.

$$\frac{\partial \mathcal{L}_{\text{content}}}{\partial {F^{\text{l}}}_{\text{ij}}} = \begin{cases} ({F^{\text{l}} - P^{\text{l}}})_{\text{ij}}{}^2 & if {F^{\text{l}}}_{\text{ij}} > 0 \\ 0 & if {F^{\text{l}}}_{\text{ij}} < 0 \end{cases}$$

As stated in earlier sections, CNNs are invariant to translation but are increasingly sensitive to the actual *content* of the image. To reiterate, the lower layers capture the pixel based relationships of the features, such as line and edges, whereas the higher layers capture the actual content of the image. It is for this reason that the feature responses of the higher layers are referred to as the *content representation*.

B. Style Representation

To generate the *style representation* of an image, a feature space, which can be built on top of the filter responses in any layer, is used to obtain the texture information from this image. Specifically, this feature space contains the feature correlations between different filter responses. These correlations are computed through the *Gram matrix*, $\mathcal{G}^{\text{l}} \in \mathcal{R}^{\text{N}_\text{l} \text{ x } \text{N}_\text{l}}$, where ${\mathcal{G}^{\text{l}}}_{\text{ij}}$ is the inner porduct between the vectorised feature maps i and j in the layer l:

$${\mathcal{G}^{\text{l}}}_{\text{ij}} = \sum_k {F_{\text{ik}}}^{\text{l}} {F_{\text{jk}}}^{\text{l}}$$

Just like the content representation, this gradient descent can be used on a random white noise image to minimise the mean-squared error to achieve a similar visualization. However, this optimization utilizes the Gram matrices from the original image $\vec{a}$ and the generated image $\vec{x}$. The respective style representations in layer l would then be A^{l} and G^{l}. The contribution of layer l to the total loss would then be

$$E_{\text{l}} = \frac{1}{4{N_{\text{l}}}^2 {M_{\text{l}}}^2} \sum_{i,j} ({G_{\text{ij}}}^{\text{l}} - {A_{\text{ij}}}^{\text{l}})^2$$

Thus, the total style loss would then be:

$$\mathcal{L}_{\text{style}}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_{\text{l}} E_{\text{l}}$$

where w_{l} are the weights of the contribution of each layer to the total loss. Since there are 5 layers used for the style transfer algorithm, each layer is weighted at $w_{\text{l}} = \frac{1}{5}$.

Once again, similar to content representation, the derivative of this mean-squared error,defined below, can be taken with respect to the activation in layer l. From this, the gradient of E_l with respect to the generated image $\vec{x}$ can be computer using standard back-propagation

$$\frac{\partial E_{\text{l}}}{\partial {F^{\text{l}}}_{\text{ij}}} = \begin{cases} \frac{1}{{N_{\text{l}}}^2 {M_{\text{l}}}^2} ((F^{\text{l}})^{\text{T}} (G^{\text{l}} - A^{\text{l}}))_{\text{ij}} & if {F^{\text{l}}}_{\text{ij}} > 0 \\ 0 & if {F^{\text{l}}}_{\text{ij}} < 0 \end{cases}$$

C. Style Transfer

The concept of *style transfer* involves the synthesizing of a new image that simultaneously matches the content representation of a photograph $\vec{p}$ and the style representation of $\vec{a}$. The mathematical problem of style transfer essentially boils down to an optimization problem. Specifically, it is the joint minimization of a white noise image's, $\vec{x}$, feature representation distance from the photograph's content representation in one layer and the artwork's style representation on a number of layers. The total cost function to me minimized is:

$$\mathcal{L}_{\text{total}}(\vec{p}, \vec{a}, \vec{x}) = \alpha \mathcal{L}_{\text{content}}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{\text{style}}(\vec{a}, \vec{x})$$

where α and β are weighing factors for the respective content and style representations. The gradient with respect to the generated image $\frac{\partial \mathcal{L}_{\text{total}}}{\partial \vec{x}}$ can be used for a numerical optimisation strategy. Gatys et al utilized the L-BFGS optimizer, however, this is not available in Tensorflow. Thus, the ADAM optimizer is used as a substitute. The style image is always resized to the same size as the content image before computing the feature representation in order to extract image information on comparable scales.

V. Summary of the Algorithm

The summary of the algorithm is shown in Figure 8. The main steps are as follows:

1) Provide the input images for the style $\vec{a}$ and content $\vec{p}$ and are passed through the network.
2) The style representation P^l is computed on all layers. The style Gram matrix at each layer is also calculated from this feature representation.
3) The content representation A^l is computed on one layer.
4) The randomly generated white noise image is passed through the network to generate its style features G^l and its content features F^l.
5) On each layer included in the style representation, the mean-squared error is calculated between the Gram matrix of the generated image G^l and the Gram matrix of the original image A^l to provide the style loss $\mathcal{L}_{\text{style}}$.
6) On the one layer included in the content representation, the mean-squared error is calculated between the generated content features F^l and the photograph's content representation P^l to provide the content loss $\mathcal{L}_{\text{content}}$
7) Since the total loss $\mathcal{L}_{\text{total}}$ is a linear combination of the content and style losses, use these two losses to compute it.

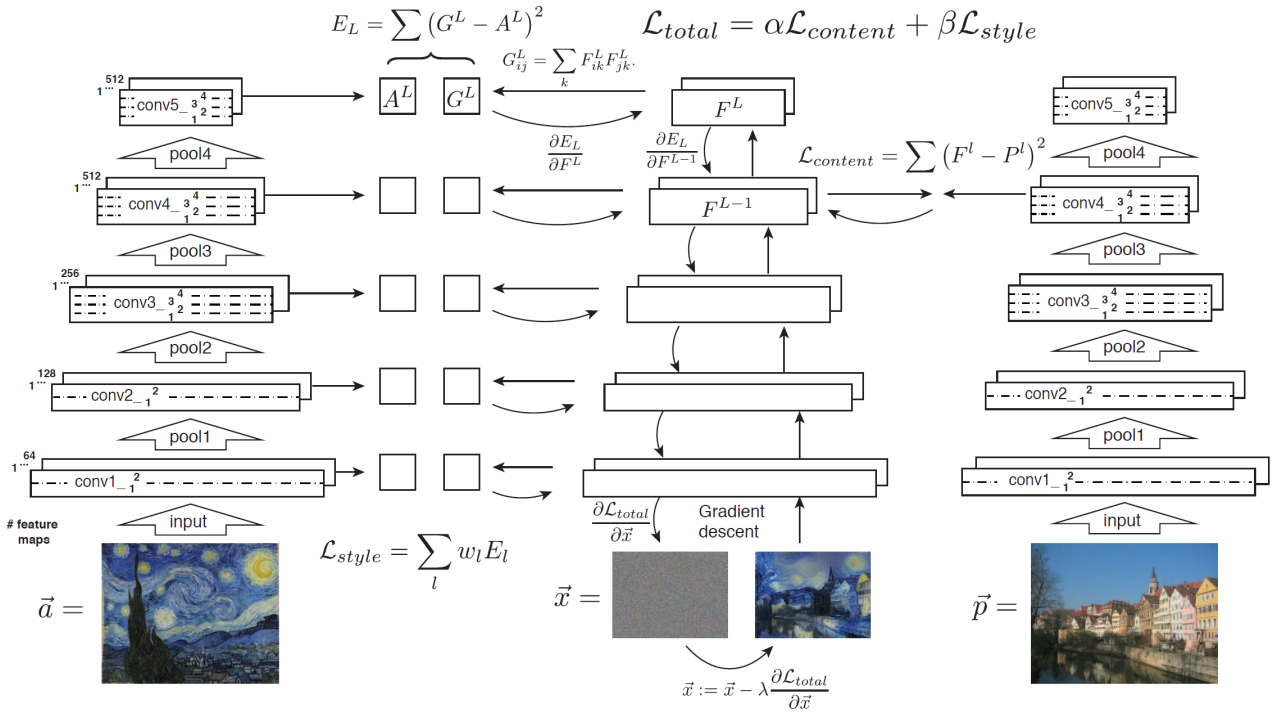


Figure 8. Style Transfer Algorithm. The detailed steps are discussed in Section V - Summary of the Algorithm

8) The derivative of the total loss with respect to the pixel values of the generated image $\frac{\partial \mathcal{L}_{\text{total}}}{\partial \vec{x}}$ is computed through standard back propagation.
9) The gradient computed in the previous step is used as an input into the ADAM optimizer and is iterated until it simultaneously matches the style features of the style image $\vec{a}$ and content photograph $\vec{p}$

VI. Python Code

The following section details the code used for this project. The first program involves functions that are utilized for the image utilities. This includes loading the image, processing the image for the VGG19 network, deprocessing it back as an image, and saving the image.

The second code is the main function. It outlines the functions used for setting up the VGG19 model, as well as the loss functions and feature extractions. The last part of the code is the actual execution that runs the training steps and saves the resulting image.

Appendix A, which is attached with this report, shows a step by step result of the code as it's run. It was executed using Google Colab and demonstrates the output and flow of how image style transfer actually works based off the written code.

A. Image Utilities

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr  6 13:26:18 2020

@author: ealegre
"""

import tensorflow as tf
import numpy as np
from PIL import Image
import matplotlib as plt

# IMAGE UTILITIES - FUNCTION 1 - LOAD IMAGE

def load_img(img_path):
    # Read the image and convert the computation graph to an image format
    img = tf.io.read_file(img_path)
    img = tf.image.decode_image(img, channels=3)
    img = tf.image.convert_image_dtype(img, tf.float32)

    # Setting the scale parameters to change the size of the image
    # Get the width and height of the image. Cast it to a float so it can be divided
    shape = tf.cast(tf.shape(img)[:-1], tf.float32)
    # Set the absolute maximum dimension for the image
    # max_dim = 1024
    max_dim = 512
    # Find which side is the longer side, this will be used to generate our scale
    max_side = max(shape)
    scale = max_dim / max_side
    new_shape = tf.cast(shape * scale, tf.int32)
    img = tf.image.resize(img, new_shape, method=tf.image.ResizeMethod.BILINEAR)

    img = img[tf.newaxis, :]
    return img

# IMAGE UTILITIES - FUNCTION 2 - DEPROCESS IMAGE
```

```
def deprocess_img(processed_img):
    processed_img = processed_img*255
    processed_img = np.array(processed_img, dtype=np.uint8)
    if np.ndim(processed_img)>3:
      assert processed_img.shape[0] == 1
      processed_img = processed_img[0]
    return Image.fromarray(processed_img)

# IMAGE UTILITIES - FUNCTION 3 - SAVE IMAGE

def save_img(best_img, path):
    img = Image.fromarray(best_img)
    img.save(path)

def imshow(image, title=None):
  if len(image.shape) > 3:
    image = tf.squeeze(image, axis=0)

  plt.imshow(image)
  if title:
    plt.title(title)
```

B. *Model Utilities & Main Function*

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr  6 13:16:38 2020

@author: ealegre
"""

# DEPENDENCIES, VARIABLES, AND PATHS

# Libraries
import tensorflow as tf
from tensorflow.python.keras import models
import os
import glob
import image_utils as IU

# Define paths
content_path = 'input/content/elon.jpg'
style_path = 'input/style/ironman.jpg'
init_path = 'input/init/512.jpg'
lena = 'input/content/lena_test.png'
lion = 'input/content/lion.jpg'
dog = 'input/content/dog.jpg'

# Load the style and content images
style_img = IU.load_img(style_path)
content_img = IU.load_img(content_path)
generated_img = tf.Variable(content_img)

# Define which layers are to be used for this model. These layers are defined in Section
# 3 of Gatys' paper
content_layers = ['block5_conv2']
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']

num_content_layers = len(content_layers)
num_style_layers = len(style_layers)

# MODEL UTILITIES - FUNCTION 1 - VGG19 LAYERS

def get_vggLayers(style_or_content_layers):
    """
    Creates the model with intermediate layer access

    This function will load in the VGG19 model used in Gatys' paper, with access to the
    intermediate layers. A new model will be generated by using these layers that will take an
    input image and return an output from the intermediate layers from the VGG19 model

    Returns
    -------
    A Keras model that takes inputs and outputs of the style and content intermediate layers

    """
    # Instantiate the VGG19 model. We are not including the fully connected layers, instantiate the weight based off ImageNet training
    vgg = tf.keras.applications.VGG19(include_top=False, weights='imagenet')

    # VGG19 model is already trained off ImageNet, set trainable to False
    vgg.trainable = False

    # All we are doing is clipping the model accrodingly. Just set the input of the model to be the same as its original
    # The outputs are going to be set to the layers specified in either the style or content layers, this will
    # be set once you've passed one of them into the function as an argument
    vgg_input = [vgg.input]
    vgg_output = [vgg.get_layer(name).output for name in style_or_content_layers]
    model = tf.keras.Model(vgg_input, vgg_output)

    return model

# MODEL UTILITIES - FUNCTION 2 - THE GRAM MATRIX

def gram_matrix(input_tensor):
  """
  Generates the Gram matrix representation of the style features. This function takes an input tensor and
  will apply the proper steps to generate the Gram matrix

  Returns
  -------
  A tensor object Gram matrix representation

  """
  # Equation (3)

  # Generate image channels. If the input tensor is a 3D array of size Nh x Nw x Nc, reshape
  # it to a 2D array of Nc x (Nh*Nw). The shape[-1] takes the last element in the shape
  # characterstic, that being the number of channels. This will be our second dimension
  channels = int(input_tensor.shape[-1])

  # Reshape the tensor into a 2D matrix to prepare for Gram matrix calculation by multiplying
  # all of the dimensions except the last one (which is what the -1 represents) together
  Fl_ik = tf.reshape(input_tensor, [-1, channels])

  # Transpose the new 2D matrix
  Fl_jk = tf.transpose(Fl_ik)

  # Find all the elements in the new array (Nw*Nh)
  n = tf.shape(Fl_ik)[0]

  # Perform the Gram matrix calculation
  gram = tf.matmul(Fl_jk, Fl_ik)/tf.cast(n, tf.float32)

  # Generate the Gram matrix as a tensor for use in our model
  gram_tensor = gram[tf.newaxis, :]
  return gram_tensor

# MODEL UTILITIES - FUNCTION 3 - STYLE & CONTENT MODEL CLASS

class StyleContentModel(tf.keras.models.Model):
  def __init__(self, style_layers, content_layers):
    super(StyleContentModel, self).__init__()
    self.vgg = get_vggLayers(style_layers + content_layers)
    self.style_layers = style_layers
```

```
    self.content_layers = content_layers
    self.num_style_layers = len(style_layers)
    self.vgg.trainable = False

  def call(self, inputs):
    # Expects a float input between [0, 1]
    inputs = inputs * 255.0
    preprocessed_input = tf.keras.applications.vgg19.preprocess_input(inputs)
    outputs = self.vgg(preprocessed_input)
    style_outputs, content_outputs = (outputs[:self.num_style_layers], outputs[self.num_style_layers:])

    style_outputs = [gram_matrix(style_output) for style_output in style_outputs]

    content_dict = {content_name:value for content_name, value in zip(self.content_layers, content_outputs)}

    style_dict = {style_name:value for style_name, value in zip(self.style_layers, style_outputs)}

    return{'content':content_dict, 'style':style_dict}


feature_extractor = StyleContentModel(style_layers, content_layers)

# Load the style and content images
style_img = IU.load_img(style_path)
content_img = IU.load_img(content_path)

# Extract the features from the style and content images
style_features = feature_extractor(style_img)['style']
content_features = feature_extractor(content_img)['content']

# The Tensorflow variable function initializes our image to be used for gradient descent.
# This image has to be the same size and type as the content image, which means that as long as it's loaded in before being
# called upon as the generated image, it should be fine to use. This is essentially the image that will be optimized.
# In case you want to use another image, you can uncomment the code below
'''
init_img = load_img(content_path)
generated_image = tf.Variable(init_img)
'''
generated_img = tf.Variable(content_img)

# Remember that all of our images are read in as a float32 type, so we have to define the range as [0, 1] to keep it within 255
def clip_range(img):
  return tf.clip_by_value(img, clip_value_min=0.0, clip_value_max=1.0)

# Choose an optimizer. The paper chose L-BFGS but Tensorflow doesn't have that, so we'll use ADAM
optimizer = tf.optimizers.Adam(learning_rate=0.02, beta_1=0.99, epsilon=1e-8)

# Remember that we are trying to optimize the Total Loss function. However, there are values that correspond to the style and content weight
# The Alpha value corresponds to content weight and the Beta corresponds to the style weight. Let's define these
alpha = 1e4
beta = 10

# MODEL UTILITIES - FUNCTION 4 - STYLE, CONTENT, & TOTAL LOSS FUNCTIONS

# Now we define our loss functions
def style_content_loss(outputs):
  style_outputs = outputs['style']
  content_outputs = outputs['content']


  # Equation 5
  # Define the style loss function
  style_loss = tf.add_n([tf.reduce_mean(tf.square(style_outputs[name] - style_features[name])) for name in style_outputs.keys()])
  # Multiply the style loss by the weighted variable to get the weighted style loss
  style_loss *= beta / num_style_layers

  # Equation 1
  # Define the content loss function
  content_loss = tf.add_n([tf.reduce_mean(tf.square(content_outputs[name] - content_features[name])) for name in content_outputs.keys()])
  # Multiply the content loss by the weighted variable to get the weighted content loss
  content_loss *= alpha / num_content_layers

  # Equation 7
  # Define the total loss function
  total_loss = style_loss + content_loss
  return total_loss

# State the total variational weight
total_variational_weight= 30

@tf.function()
def train_step(img):
  # Tensorflow's GradientTape function performs automatic differentiation of the input
  with tf.GradientTape() as tape:
    outputs = feature_extractor(img)
    loss = style_content_loss(outputs)
    loss += total_variational_weight * tf.image.total_variation(img)

  # Apply the gradient by passing the loss and the generated image
  grad = tape.gradient(loss, img)

  # The gradient is optimized using the ADAM optimizer we declared earlier. This optimizes the
  # generated image using the gradient values
  optimizer.apply_gradients([(grad, img)])

  # The image is rewritten, being sure that the values all stay within the viable range of [0, 1]
  img.assign(clip_range(img))

import time
from IPython import display
# Start the run time
start = time.time()

epochs = 100
iterations = 100

step = 0
for n in range(epochs):
  for m in range(iterations):
    step += 1
    train_step(generated_img)
    print(".", end=' ')
  display.clear_output(wait=True)
  display.display(IU.deprocess_img(generated_img))
  print("Train step: {}".format(step))

end = time.time()
print("Total Time: {:.1f}".format(end-start))

file_name = 'output/stylized-image.png'
IU.deprocess_img(generated_img).save(file_name)
```

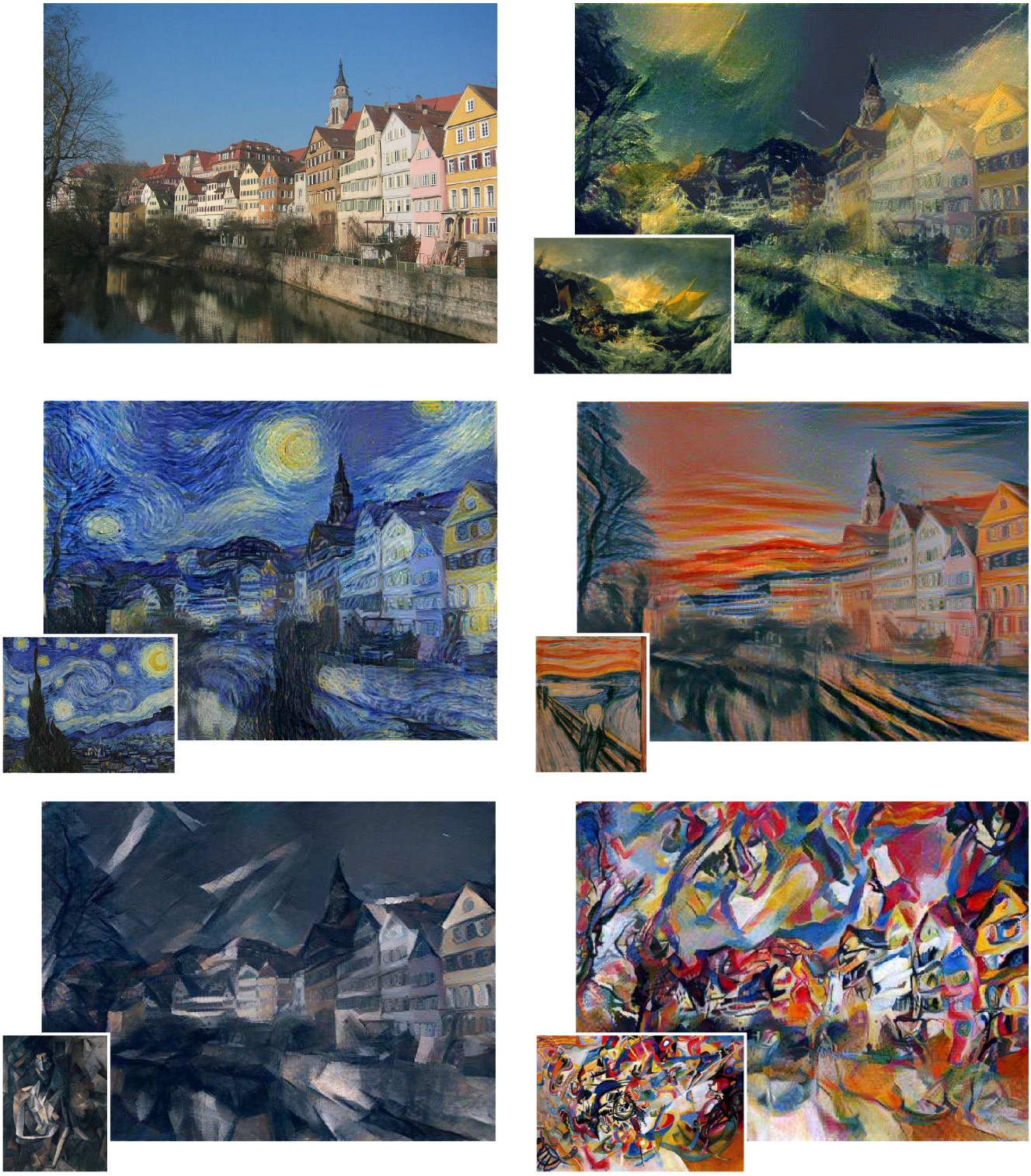


Figure 9. Theoretical Results from Gatys et al. The images were created by finding an image that simultaneously matches the content representation of the photograph and the style representation of the artwork. The original photograph depicting the Neckarfront in Tubingen, Germany, is shown (from the top left downwards then across to the right side) in **A** (Photo: Andreas Praefcke). The painting that provided the style for the respective generated image is shown in the bottom left corner of each panel. **B** The Shipwreck of the Minotaur by J.M.W. Turner, 1805. **C** The Starry Night by Vincent van Gogh, 1889. **D** Der Schrei by Edvard Munch, 1893. **E** Femme nue assise by Pablo Picasso, 1910. **F** Composition VII by Wassily Kandinsky, 1913.

Figure 10. Actual results from implementation. The original photograph depicting the skyline in Toronto, Canada, is shown (from the top left downwards then across to the right side) in **A** (Photo: Andreas Praefcke). The painting that provided the style for the respective generated image is shown in the bottom left corner of each panel. **B** The Shipwreck of the Minotaur by J.M.W. Turner, 1805. **C** The Starry Night by Vincent van Gogh, 1889. **D** Der Schrei by Edvard Munch, 1893. **E** Femme nue assise by Pablo Picasso, 1910. **F** Composition VII by Wassily Kandinsky, 1913.

VII. Results, Discussions, and Future Work

A. Theoretical Results

The results achieved by Gatys et al is shown in Figure 9. There are a couple of interesting differences to note. First and foremost, while it was very easy to derive the generic steps, there are some nuances that Gatys does not explicitly state in his paper. One of these is how many iterations are ran per result. The number of iterations dictates how much of the "style" is transferred over to the newly synthesized image. However, this can be remedied easily as it is very obvious to note after some iterations when the loss function converges. However, Gatys never stated what size his initial or final images were, despite his suggestion of constraining the image size before passing it through the VGG19 network.

The second is the optimizer. Gatys suggest using the L-BFGS optimizer to optimize the gradient of the total loss function with respect to the generated image. However, as discussed previously, the L-BFGS optimizer is not implemented in Tensorflow. Furthermore, the parameters of the optimizer were not stated.

The third nuance is that Gatys extracted the content image on layer 4_2. While it is never explicitly stated why this was done, it is assumed that Gatys assumed that extracting the content features from this layer was high enough to allow the more profound "features" of the content image to be displayed in the style transfer.

Finally, while the results from Gatys' style transfer serve as the benchmark, there is no numerical results to quantify how well his method works. As such, the results are evaluated just as art would be in real life: objectively.

B. Experimental Results

The implemented results from my code can be viewed in Figure 10. This section will discuss the deviations and remedies to the nuances discussed earlier. First and foremost, our images were synthesized with the following benchmark parameters

- Maximum Image Dimension: 1024 pixels
- Iterations/Epoch: 100
- Epochs: 100
- Optimizer: ADAM
- Learning Rate: 0.02
- Beta: 0.99
- Epsilon: 1e-1
- Alpha (Content Weight): 1e4
- Beta (Style Weight): 10
- Content Layers: 5_2

These parameters are important to note as some deviate from Gatys' original suggestions and others are derived from experimentation due to the lack of information provided in the paper. As stated previously, Gatys never specified the number of iterations or the image size. The maximum image size of 1024 was chosen as it is a staple size when it comes to images, as well as the fact that it is substantially large enough that the contents of the image are distinguishable when viewed. After some experimentation with the benchmark parameters, it was decided that 100 epochs with 100 iterations provided a result that was satisfactory.

The most notable deviation from Gatys' method was the difference in where the content image was extracted. Instead of extracting the content image on Gatys' suggested layer of 4_2, our content image was extracted on layer 5_2. This was due to the difference in the optimizer used in our algorithm. While it was possible to extract our content image on the same layer Gatys did, the results were not always pleasant or consistent. By that, it's meant that the model needs to be constantly adjusted based on the style image and content image used. It was fairly difficult to find the right balance of style/content weights and optimizer parameters that would yield suitable results from this layer. We also used the ADAM optimizer that was tuned with a learning rate of 0.02, a beta value of 0.99, and an epsilon of 1e-1. Using Gatys' Alpha and Beta values also yielded poor results, as we basically had to siwtch his values. In Gatys' method, the style weight was weighed significantly more that the content weight, ours was the complete opposite.These values were achieved after some investigation online. These were the hyper parameters that contributed to the deviation of our results from Gatys'.

The final deviation that our method has is that it accounts for total variational loss. While this was never mentioned in Gatys' paper, the total variational loss allows for the elimination of high frequency artifacts from our synthesized image, making them appear more natural.

Overall, our method seems to work quite well, although not as well as Gatys'. The majority of the images are comparable to the original results, however, the deviations become apparent when comparing images synthesized from Van Gogh's Starry Night. Gatys' results yielded a comparable image, with Van Gogh's swirled portrayal of the night sky as being more prominent, as well as the actual moon being included in the final image. Our result does a good job of copying this swirl pattern but not to the same degree. Furthermore, the patterns of the stars are not as intelligently placed as Gatys' results are. Surprisingly enough, Starry Night was used as the default image and is what was used to test the parameters found above. Other images were very easy to tune regardless of the parameters but Van Gogh's Starry Night had a very small window of passable results, which prompted the final benchmark values seen above.

C. Future Work

First and foremost, any future work done on this project should focus on trying to implement Gatys' method as closely as possible. While most modern and up to date tools make it easy to get close to Gatys' method, there are elements that are

lacking and crucial to obtaining the same results. Specifically, the optimizer used for gradient descent plays a larger role thatn originally understood. Despite my best efforts, implementing the L-BFGS optimizer was out of the scope for this project as it would require a much deeper understanding of numerical analysis that what this project entailed. Furthermore, many online resources have stated that using the ADAM optimizer was ”comparable” to using the L-BFGS optimizer, it was never truly elaborated on what that meant. While the results obtained can be called comparable to Gatys', there are still glaring issues that need to be addressed.

Furthermore, the hyper parameters used could also be fine tuned. With the use of a more powerful machine and parallel computing, different ranges and combinations of these parameters can be run in parallel to see what would give the closest result to the original.

There was also an interest in writing a script that produces a high resolution stylized image. The framework of this is to essentially start from a low resolution input, with a maximum dimension of 256 pixels, run that through the algorithm for a certain number of iterations, save the result and then use this result as the generated image for a secondary run with a maximum dimension of 512 pixels. After each run, the process would be repeated, with the result of the previous run being used as the generated initial image and continuously up-scaling the output until the desired high resolution result was achieved. The issue with this is that it would be fairly time consuming as the number of iterations would need to change every time, the hyper parameters tuned after every run, and the run times would increase exponentially as the resolution of the image increased. Current implementations exist and utilize 4 GPUs which result in an image with a maximum dimension of 3620 pixels.

The most impressive improvement to this algorithm would come in the form of either *photorealistic style transfer* or *multiple style transfer*. While the prior was not researche during this project, the latter already has an implementation. With a bit of tweaking, this code could accommodate multiple style images that would be used for the style transfer. The only constraint is optimization of

References

[1] L. Gatys, A.Ecker, M.Bethge, “Image Style Transfer Using Convolutional Neural Networks,” 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, 2016, pp. 2414-2423.

Image Style Transfer Using Convolutional Neural Networks

This notebook is intended to help breakdown the step by step process of the paper: *Image Style Transfer Using Convolutional Neural Networks* by Gatys et al.

It will go through the code step by step to showcase how each block of code works. Let us first begin by loading our dependencies and mounting our Google Drive for use

In [29]:
```
# Mount Google Drive
from google.colab import drive
from google.colab.patches import cv2_imshow
import os
drive.mount('/content/drive')
# Change the working directory so that we have access to our files
os.chdir('/content/drive/My Drive/Engineering Projects/EE8601-Project2/project_builds')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.moun
t("/content/drive", force_remount=True).
```

Dependencies

In [0]:
```
# DEPENDENCIES, VARIABLES, AND PATHS

# Libraries
import tensorflow as tf
import numpy as np
from tensorflow.python.keras import models
import os
import glob
from PIL import Image
from tensorflow.python.keras.preprocessing import image as tf_image
import matplotlib.pyplot as plt
import cv2

# Define paths
content_path = '/content/drive/My Drive/Engineering Projects/EE8601-Project2/project_bui
lds/input/content/elon.jpg'
style_path = '/content/drive/My Drive/Engineering Projects/EE8601-Project2/project_build
s/input/style/ironman.jpg'
init_path = '/content/drive/My Drive/Engineering Projects/EE8601-Project2/project_build
s/input/init/512.jpg'
lena = '/content/drive/My Drive/Engineering Projects/EE8601-Project2/project_builds/lena
_test.png'
lion = '/content/drive/My Drive/Engineering Projects/EE8601-Project2/project_builds/lio
n.jpg'
dog = '/content/drive/My Drive/Engineering Projects/EE8601-Project2/project_builds/inpu
t/content/dog.jpg'
```

Part 1: Image Utilities

First and foremost, we have to define a couple of functions that will allow us to actually read the images and manipulate them for use in the VGG19 network. This next section will define functions to be used for image manipulation, including loading an image, preprocessing it for use in the VGG19 network, deprocessing it once it has been run through the network, and then saving the image itself.

Function 1 - Load Image

In [0]:
```
# IMAGE UTILITIES - FUNCTION 1 - LOAD IMAGE

def load_img(img_path):
    # Read the image and convert the computation graph to an image format
    img = tf.io.read_file(img_path)
    img = tf.image.decode_image(img, channels=3)
    img = tf.image.convert_image_dtype(img, tf.float32)

    # Setting the scale parameters to change the size of the image
    # Get the width and height of the image. Cast it to a float so it can be divided
    shape = tf.cast(tf.shape(img)[:-1], tf.float32)
    # Set the absolute maximum dimension for the image
    max_dim = 1024

    # Find which side is the longer side, this will be used to generate our scale
    max_side = max(shape)
    scale = max_dim / max_side
    new_shape = tf.cast(shape * scale, tf.int32)
    img = tf.image.resize(img, new_shape, method=tf.image.ResizeMethod.BILINEAR)

    img = img[tf.newaxis, :]
    return img
```

Example 1 - Load an Image

Let's run this code to see how it will work

In [32]:
```
# We will use the the Lena image as an example, the path to which was already defined ab
ove
# First read the image and see how a tensor works

img_ex= tf.io.read_file(content_path)
img_ex.shape
```

Out[32]:
```
TensorShape([])
```

Okay, great! We successfully imported our image using Tensorflow's built in function. The caveat is that it reads the files as a *Tensor* instead the regular 3D matrix object format.. The problem with using PIL is how it *reads the image as an input*. That is, it doesn't actually read the image the way we want it to. If you're familiar with OpenCV (which I will now refer to as cv2), then you should know the progenitor function

```
img = cv2.imread(path)
```

This is a very useful function since it reads the image as an actual array but since we're going to be manipulating the image using a CNN, we need to read it as a tensor.

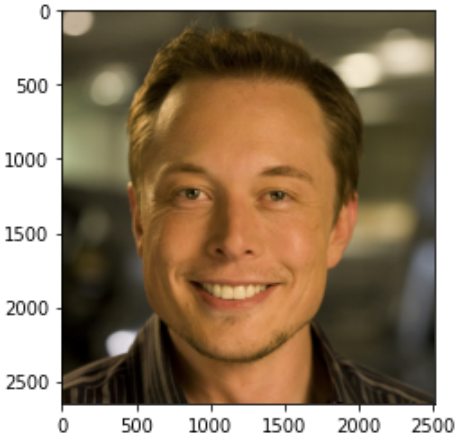
If we tried to use cv2, t would return an error simply because of how OpenCv reads the image. Now the problem is, we can't extract any information about this image!

It all boils down to how Tensorflow reads the image, it doesn't know it's an image since it creates a computation graph that needs to be evaluated. We can do this by using their command to decode this as an image. We're also going to take this time to convert the image from uint8 to float32 since we're going to need that flexibility down the road. Specifically, it's due to the means calculated from the ImageNet data set, which will be discussed further in this notebook.

In [33]:
```
# Decode the input tensor into a displayable image format, while converting the data typ
e to float
img_ex = tf.image.decode_image(img_ex, channels=3)
img_ex = tf.image.convert_image_dtype(img_ex, tf.float32)

# Display the image
plt.imshow(img_ex)
img_ex.shape
```

Out[33]:
```
TensorShape([2649, 2513, 3])
```



Now we see our image! We also wish to be able to quickly change our image dimensions for processing since our input style and content images must match dimensions. Let's take care of that.

First, we need to extract the width and height of the image so we can adjust them.

In [34]:
```
# Get the width and height of the image. Cast it to a float so it can be divided
shape = tf.cast(tf.shape(img_ex)[:-1], tf.float32)
shape
```

Out[34]:
```
<tf.Tensor: shape=(2,), dtype=float32, numpy=array([2649., 2513.], dtype=float32)>
```

The Tensorflow shape function returns the shape of a tensor. The [:-1] at the end just makes it so that the shape its returning is one dimension LESS than whatever it currently is. Basically, it's dropping the colour channel dimension since we only care about the width and height.

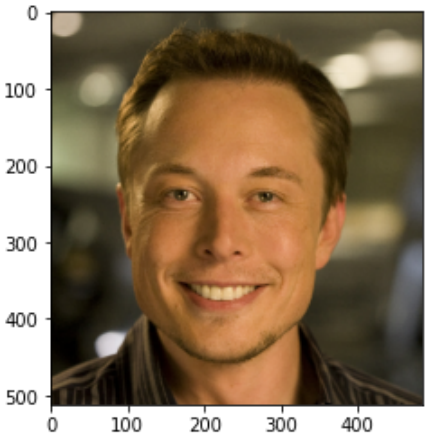
The Tensorflow cast function changes the tensor data type from one into another. The shape function would return an output that is of type integer. We want to be able to ADJUST the aspect ratio, which means we will be dividing it shortly. Thus, we want a float data type to ensure that the division goes smoothly.

In [35]:
```
# Set the absolute maximum dimension for the image
max_dim = 512
# Find which side is the longer side, this will be used to generate our scale
max_side = max(shape)
scale = max_dim / max_side
print('Our maximum side is', np.array(max_side, dtype=np.uint32), 'pixels long so it wil
l be scaled down to', max_dim, 'pixels.')
new_shape = tf.cast(shape * scale, tf.int32)
img_ex_a = tf.image.resize(img_ex, new_shape)
print()

# Display the image
plt.imshow(img_ex_a)
print('The new dimension of our image is', np.array(new_shape, dtype=np.uint32))
```

```
Our maximum side is 2649 pixels long so it will be scaled down to 512 pixels.

The new dimension of our image is [512 485]
```



Notice what happened here. We essentially downsampled our image by applying a scaled value to each of its dimensions in the resize operation.

Now that it has been resized, we're going to add an extra dimension in the front so it becomes a 4D object that the VGG19 network can take in.

In [36]:
```
img_ex_b = img_ex_a[tf.newaxis, :]
print('Our tensor has the dimensions:', img_ex_b.shape)


 # Sanity check to make sure our defined function works as well
img_ex_c = load_img(lena)
img_ex_c.shape
```

```
Our tensor has the dimensions: (1, 512, 485, 3)
```

Out[36]:
```
TensorShape([1, 682, 1024, 3])
```

Great! Our first function works! Now let's define a couple more that we'll need to use. We need to deprocess the image once we've received it back from our CNN. Then, we need a function that will save our actual result. Finally, I'll define a modified imshow function to make it easy to see our images at any point during our examples

Let's define these below

Function 2 - Deprocess Image

In [0]:
```
# IMAGE UTILITIES - FUNCTION 2 - DEPROCESS IMAGE

def deprocess_img(processed_img):
    processed_img = processed_img*255
    processed_img = np.array(processed_img, dtype=np.uint8)
    if np.ndim(processed_img)>3:
      assert processed_img.shape[0] == 1
      processed_img = processed_img[0]
    return Image.fromarray(processed_img)
```

Function 3 - Save Image

In [0]:
```
# IMAGE UTILITIES - FUNCTION 3 - SAVE IMAGE

def save_img(best_img, path):
    img = Image.fromarray(best_img)
    img.save(path)
```

Function X - Modified imshow

In [0]:
```
def imshow(image, title=None):
  if len(image.shape) > 3:
    image = tf.squeeze(image, axis=0)

  plt.imshow(image)
  if title:
    plt.title(title)
```

Example 2 - Deprocessing an Image

Let's run through these functions to understand a little bit about how they work.

The deprocess_img() function essentially turns our tensor back into a 3D image once again. It also incorporates the mean values used to train teh VGG19 network with the ImageNet dataset since that's extremely important.

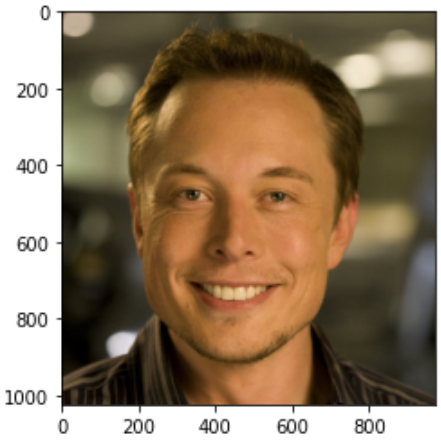
The save_img() function is pretty basic and self explanitory. Once our image utility functions are all verified, we can dive into the defining the actual model!

In [40]:
```
# Verify deprocess_img()

img_ex2 = load_img(content_path)
img_ex2_deproc = deprocess_img(img_ex2)

plt.imshow(img_ex2_deproc)
print('The deprocessed image with size', img_ex2_deproc.size)
```

```
The deprocessed image with size (971, 1024)
```

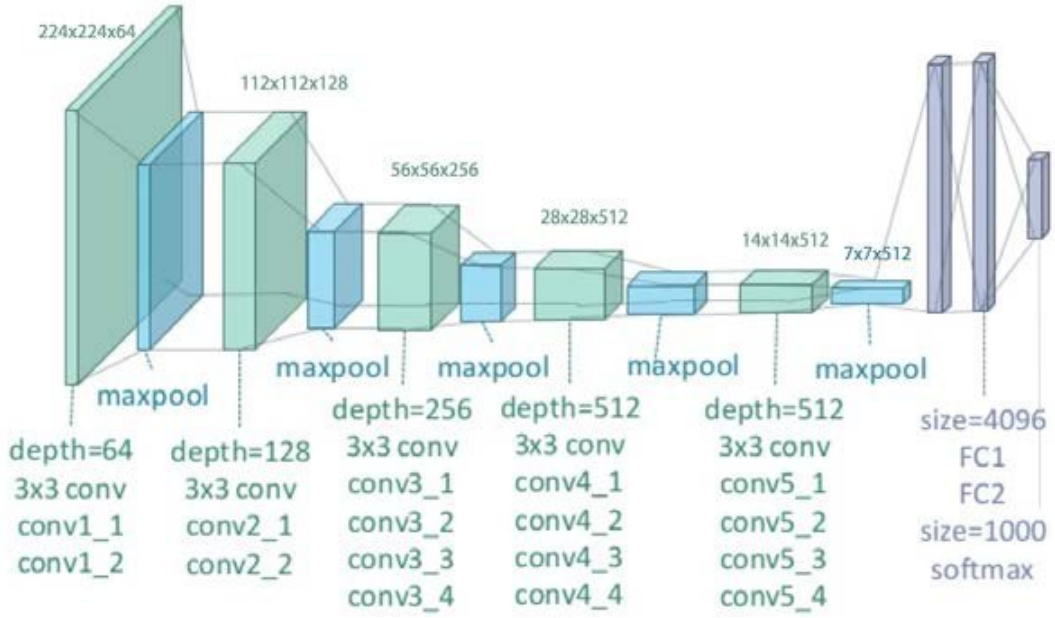


Part 2: The VGG19 Model

Now, we're going to define our model, specifically the VGG19 network. We'll be using Tensorflow to handle the actual model building, as well as Keras to make it easier to run the model itself. Every function built in this section corresponds to the theory outlined by Gatys' paper.

Let's define some prerequisite knowledge first. Obviously we're going to need two images, the *content* image (which is what the actual subject of our final synthesized image will be) and the *style* image (which is the... style of our synthesized image... no easy way to describe that!).

Next we need the model, which is obviously the VGG19 network. The VGG19 network is shown below



The '19' in VGG19 is an indication of how many layers there are in the network. There are 16 convolutional layers and 3 fully connected layers.

As stated before, we'll be using Keras to manipulate this network. Keras makes it easy since we don't need to build the network with its weights and biases from scratch. Let's load up the model and verify that all of its layers are included

In [41]:
```
# The argument, include_top, can be set to either 'True' or 'False'. In this case we
# are setting it to True since we want to see ALL the layers. The include_top is specifi
cally
# the last 3 fully connected layers. Weights is the weights that are used for each laye
r, specifically
# the results that the network was trained with when the Imagenet data set was run throu
gh it
vgg = tf.keras.applications.VGG19(include_top=True, weights='imagenet')

for layer in vgg.layers:
  print(layer.name)
```

```
input_6
block1_conv1
block1_conv2
block1_pool
block2_conv1
block2_conv2
block2_pool
block3_conv1
block3_conv2
block3_conv3
block3_conv4
block3_pool
block4_conv1
block4_conv2
block4_conv3
block4_conv4
block4_pool
block5_conv1
block5_conv2
block5_conv3
block5_conv4
block5_pool
flatten
fc1
fc2
predictions
```

Great, it loaded up correctly! Let's actually try it out and try to get a handle on what this CNN is capable of. We're going to pass an image of a lion into the actual thing. Before we do that, we have to call a Keras function that will preprocess the image for us to use in the VGG19 network. All this function does is extract the features from the image on a specified layer(s) to be used in the VGG19 model. Then run it through the VGG19 model to have it predict what's in the actual image itself. If the code works, it should give us a lion as its highest predicted output.

Example 3 - Testing a Network

In [42]:
```
# Load the image. Remember that it's being loaded as a float so multiply it by 255
img_ex3 = load_img(lion)
ex3 = tf.keras.applications.vgg19.preprocess_input(img_ex3*255)

# Change the input size since the default input size of the VGG19 model is 224x224
img_ex3_b = tf.image.resize(ex3, (224, 224))

vgg = tf.keras.applications.VGG19(include_top=True, weights='imagenet')
prediction_probabilities = vgg(img_ex3_b)
prediction_probabilities.shape

predicted_top_5 = tf.keras.applications.vgg19.decode_predictions(prediction_probabilitie
s.numpy())[0]
[print(class_name, '=', prob) for (number, class_name, prob) in predicted_top_5]

topprediction = predicted_top_5[0]
if topprediction[1] == 'lion':
  print('The top prediction is LION, our model works!')
```

```
lion = 0.99999964
cheetah = 1.8043703e-07
chow = 7.2701944e-08
macaque = 2.1276694e-08
tiger = 1.9176262e-08
The top prediction is LION, our model works!
```

Now we can get started on actually defining our first function of this section. Specifically, building our layer based on how the paper defines in. In the paper, only specific layers are used by Gatys. The content image is only passed to 'block4_conv2' and the style image is passed to 'block1_conv1', 'block2_conv1', 'block3_conv1', 'block4_conv1', and 'block5_conv1'. Let's start this section off by keeping these layers in their own array so we can call them up at any point

To define a model using Keras we use the function

```
model = Model(inputs, outputs)
```

Let's write a function to SPECIFICALLY call upon the layers we want to use instead of all the layers being utilized by default

Function 1 - The VGG19 Network

In [0]:
```
import tensorflow as tf
import numpy as np
from tensorflow.python.keras import models

# Define which layers are to be used for this model. These layers are defined in Section
# 3 of Gatys' paper
content_layers = ['block5_conv2']
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']

num_content_layers = len(content_layers)
num_style_layers = len(style_layers)

# MODEL UTILITIES - FUNCTION 1 - VGG19 LAYERS

def get_vggLayers(style_or_content_layers):
    """
    Creates the model with intermediate layer access

    This function will load in the VGG19 model used in Gatys' paper, with access to the
    intermediate layers. A new model will be generated by using these layers that will t
ake an
    input image and return an output from the intermediate layers from the VGG19 model

    Returns
    -------
    A Keras model that takes inputs and outputs of the style and content intermediate la
yers

    """
    # Instantiate the VGG19 model. We are not including the fully connected layers, inst
antiate the weight based off ImageNet training
    vgg = tf.keras.applications.VGG19(include_top=False, weights='imagenet')

    # VGG19 model is already trained off ImageNet, set trainable to False
    vgg.trainable = False

    # All we are doing is clipping the model accrodingly. Just set the input of the mode
l to be the same as its original
    # The outputs are going to be set to the layers specified in either the style or con
tent layers, this will
    # be set once you've passed one of them into the function as an argument
    vgg_input = [vgg.input]
    vgg_output = [vgg.get_layer(name).output for name in style_or_content_layers]
    model = tf.keras.Model(vgg_input, vgg_output)

    return model
```

Example 4 - Creating a Model

Let's create a model using the style layers

In [44]:
```
# Load an image
style_img_ex4 = load_img(style_path)

# Extract the layers needed
style_extractor_ex4 = get_vggLayers(style_layers)

# Find the outputs
style_outputs_ex4 = style_extractor_ex4(style_img_ex4*255)

# Look at the stats of each layer's output
for name, output in zip(style_layers, style_outputs_ex4):
  print(name)
  print("  Shape: ", output.numpy().shape)
  print("  Min: ", output.numpy().min())
  print("  Max: ", output.numpy().max())
  print("  Mean: ", output.numpy().mean())
  print()
```

```
block1_conv1
  Shape:  (1, 1024, 1024, 64)
  Min:  0.0
  Max:  838.2029
  Mean:  38.70089

block2_conv1
  Shape:  (1, 512, 512, 128)
  Min:  0.0
  Max:  4707.831
  Mean:  215.44833

block3_conv1
  Shape:  (1, 256, 256, 256)
  Min:  0.0
  Max:  10762.438
  Mean:  265.72168

block4_conv1
  Shape:  (1, 128, 128, 512)
  Min:  0.0
  Max:  31911.98
  Mean:  800.5783

block5_conv1
  Shape:  (1, 64, 64, 512)
  Min:  0.0
  Max:  3708.13
  Mean:  54.836765
```

Perfect, now we know our function works as we want it to since it only returned details about the layers we called. We can try another example with the content layers to verify it once again

In [45]:
```
# Load an image
content_img_ex4 = load_img(content_path)

# Extract the layers needed
content_extractor_ex4 = get_vggLayers(content_layers)

# Find the outputs
content_outputs_ex4 = content_extractor_ex4(content_img_ex4*255)

# Look at the stats of each layer's output
for name, output in zip(content_layers, content_outputs_ex4):
  print(name)
  print("  shape: ", output.numpy().shape)
  print("  min: ", output.numpy().min())
  print("  max: ", output.numpy().max())
  print("  mean: ", output.numpy().mean())
  print()
```

```
block5_conv2
  shape:  (64, 60, 512)
  min:  0.0
  max:  1087.1655
  mean:  7.5524983
```

Now we REALLY know it works! Let's move onto the next part of the code. Specifically, the Gram matrix!

The Gram matrix is a bit tricky. It will essentially act as our style representation. At its core, the Gram matrix is the result of multiplying a matrix with the transpose of itself. We normalize it by dividing it with the total number of elements in that layer. Let's make that function.

Function 2 - The Gram Matrix

In [0]:
```
# MODEL UTILITIES - FUNCTION 2 - THE GRAM MATRIX

def gram_matrix(input_tensor):
  """
  Generates the Gram matrix representation of the style features. This function takes an
input tensor and
  will apply the proper steps to generate the Gram matrix

  Returns
  -------
  A tensor object Gram matrix representation

  """
  # Equation (3)

  # Generate image channels. If the input tensor is a 3D array of size Nh x Nw x Nc, res
hape
  # it to a 2D array of Nc x (Nh*Nw). The shape[-1] takes the last element in the shape
  # characterstic, that being the number of channels. This will be our second dimension
  channels = int(input_tensor.shape[-1])

  # Reshape the tensor into a 2D matrix to prepare for Gram matrix calculation by multip
lying
  # all of the dimensions except the last one (which is what the -1 represents) together
  Fl_ik = tf.reshape(input_tensor, [-1, channels])

  # Transpose the new 2D matrix
  Fl_jk = tf.transpose(Fl_ik)

  # Find all the elements in the new array (Nw*Nh)
  n = tf.shape(Fl_ik)[0]

  # Perform the Gram matrix calculation
  gram = tf.matmul(Fl_jk, Fl_ik)/tf.cast(n, tf.float32)

  # Generate the Gram matrix as a tensor for use in our model
  gram_tensor = gram[tf.newaxis, :]
  return gram_tensor
```

In [0]:
```
# This is the Tensorflow backup, to be used in case my implementation doesnt work for wh
atever reason

def gram_matrix_(input_tensor):
  summation = tf.linalg.einsum('bijc, bijd->bcd', input_tensor, input_tensor)
  input_shape = tf.shape(input_tensor)
  num_locations = tf.cast(input_shape[1] * input_shape[2], tf.float32)
  result = summation/num_locations
  return result
```

Example 5 - Understanding the Gram Matrix

Let's see how it actually works. First thing's first, we need a tensor to use for an example. Let's just input an image since our load_img() function instantiates the image as a tensor type object. Once we do that, we can extract the number of channels in the input

In [48]:
```
# Load an input image as an example tensor
style_img_ex5 = load_img(style_path)
print('The original shape of this tensor is', style_img_ex5.shape)

# Grab the number of channels in the image. The shape[-1] takes the last element in the
 shape
# characterstic, that being the number of channels. This will be our second dimension
channels = int(style_img_ex5.shape[-1])

print('This input tensor has', channels, 'channels, which will be our new second dimensi
on')
```

```
The original shape of this tensor is (1, 1024, 1024, 3)
This input tensor has 3 channels, which will be our new second dimension
```

Now let's reshape the input tensor accordingly. Since we're basically multiplying the matrix by its own transpose, we need to apply a matrix multiplication. We have to change this tensor into a 2D matrix. We'll use the Tensorflow reshape function to do that. By passing in our tensor as the first argument and then stating the bounds of the resulting output shape, we can make the 3D image (technically it's a 4D tensor, but since the first dimension has a value of 1, we'll disregard it knowing that this is an image) into a 2D matrix. the [-1, channels] makes it so that it takes all of the dimensions before the last dimension, that being [1, W, H] and multiplies them together.

In [49]:
```
# Reshape the tensor into a 2D matrix to prepare for Gram matrix calculation
Fl_ik = tf.reshape(style_img_ex5, [-1, channels])

print('This tensor will be reduced by multiplying the first 3 dimensions together. Remem
ber that Dimension 1 is of size 1')
print('Dimension 2 is of size', style_img_ex5.shape[1])
print('Dimension 3 is of size', style_img_ex5.shape[2])
print('Multiplying these together, the new size of the first element of the reshaped 2D
 matris is', style_img_ex5.shape[1]*style_img_ex5.shape[2])

# Verification of the above logic
if Fl_ik.shape[0] == style_img_ex5.shape[1]*style_img_ex5.shape[2]:
  print('Since Fl_ik has the new shape', Fl_ik.shape, 'our logic is correct')
```

```
This tensor will be reduced by multiplying the first 3 dimensions together. Remember tha
t Dimension 1 is of size 1
Dimension 2 is of size 1024
Dimension 3 is of size 1024
Multiplying these together, the new size of the first element of the reshaped 2D matris
is 1048576
Since Fl_ik has the new shape (1048576, 3) our logic is correct
```

Now let's actually start the Gram matrix calculation. We've already defined the matrix we need to multiply. Now, we just have to define its transpose and actually multiply the two matrices.

While we're at it, we'll also grab the number of elements (pixels) in the image to normalize our Gram matrix. Once we do that we'll perform the matrix multiplication and normalization.

In [50]:
```
# Generate the transpose of our new 2D matrix
Fl_jk = tf.transpose(Fl_ik)

# Get number of elements in the full matrix
n = tf.shape(Fl_ik)[0]

# Perform the matrix multiplication
gram = tf.matmul(Fl_jk, Fl_ik)/tf.cast(n, tf.float32)

# Instantiate the Gram matrix as a tensor since it still represents our style features,
 at the end of the day
gram_tensor = gram[tf.newaxis, :]
print(gram_tensor)

# Sanity check
print(gram_matrix(style_img_ex5))
print(gram_matrix_(style_img_ex5))
```

```
tf.Tensor(
[[[0.5022685  0.32141384 0.2898315 ]
  [0.32141384 0.22958724 0.19764337]
  [0.2898315  0.19764337 0.19610038]]], shape=(1, 3, 3), dtype=float32)
tf.Tensor(
[[[0.5022685  0.32141384 0.2898315 ]
  [0.32141384 0.22958724 0.19764337]
  [0.2898315  0.19764337 0.19610038]]], shape=(1, 3, 3), dtype=float32)
tf.Tensor(
[[[0.5022685  0.32141384 0.2898315 ]
  [0.32141384 0.22958724 0.19764337]
  [0.2898315  0.19764337 0.19610038]]], shape=(1, 3, 3), dtype=float32)
```

Function 3 - Style and Content Feature Extraction

We're going to build the model so that it can easily extract the style and content feature tensors. Specifically, we can do this by using a Python class, which is similar to classes in OOP like Java. Every Python class object has a "**init**()" function that is called every time the class is being used to create a new object. All this function does is assign values to the object properties that the object needs when it's being created.

In this case, we basically need to define all the properties related to making our model which we already know since we've done a couple of examples already. But it basically needs to know which model it's using and what layers it needs to activate on. All of this is related to the tf.keras.models.Model and can be defined using the self parameter

The "call()" method is called when the instance is called. When a variable of this class is called, this method is what is used to call it. Let's discuss what each line does. When we call this function, we provide an image input of type float32. The image is then multiplied by 255 such that it can be preprocessed for use in the VGG network. It's then passed through the network to provide the outputs. These outputs are split between the style and content outputs and placed in dictionaries so that it can be called upon when needed. The function returns both dictionaries which can be called upon if specificed. The style dictionary returns the gram matrix (style) of the style layers and the content of the content layers.

In [0]:
```
# MODEL UTILITIES - FUNCTION 3 - STYLE & CONTENT MODEL CLASS

class StyleContentModel(tf.keras.models.Model):
  def __init__(self, style_layers, content_layers):
    super(StyleContentModel, self).__init__()
    self.vgg = get_vggLayers(style_layers + content_layers)
    self.style_layers = style_layers
    self.content_layers = content_layers
    self.num_style_layers = len(style_layers)
    self.vgg.trainable = False

  def call(self, inputs):
    # Expects a float input between [0, 1]
    inputs = inputs * 255.0
    preprocessed_input = tf.keras.applications.vgg19.preprocess_input(inputs)
    outputs = self.vgg(preprocessed_input)
    style_outputs, content_outputs = (outputs[:self.num_style_layers], outputs[self.num_
style_layers:])

    style_outputs = [gram_matrix(style_output) for style_output in style_outputs]

    content_dict = {content_name:value for content_name, value in zip(self.content_layer
s, content_outputs)}

    style_dict = {style_name:value for style_name, value in zip(self.style_layers, style
_outputs)}

    return{'content':content_dict, 'style':style_dict}


feature_extractor = StyleContentModel(style_layers, content_layers)
```

Example 6 - Style and Feature Extraction

Let's try to extract the features and see what the model outputs. Our above function integrates the previously built functions to make our flow seamless

In [52]:

```
# Load an image
content_img_ex6 = load_img(content_path)

# A Tensorflow constant is different from a variable. A constant type makes it so that w
hen you declare it,
# the actual value of it can't be changes
results_ex6 = feature_extractor(tf.constant(content_img_ex6))

# This line of the code technically doesn't do anything for this example but it's an exa
mple for how the results would be called
# further down the line. Basically, you can pull either the style or content results by
 calling the variable holding the
# feature extractor outputs and specifying whether you wish to get the style or content
 results in a square bracket box
style_results_ex6 = results_ex6['style']

print('Styles:')
for name, output in sorted(results_ex6['style'].items()):
  print("  ", name)
  print("    shape: ", output.numpy().shape)
  print("    min: ", output.numpy().min())
  print("    max: ", output.numpy().max())
  print("    mean: ", output.numpy().mean())
  print()

print("Contents:")
for name, output in sorted(results_ex6['content'].items()):
  print("  ", name)
  print("    shape: ", output.numpy().shape)
  print("    min: ", output.numpy().min())
  print("    max: ", output.numpy().max())
  print("    mean: ", output.numpy().mean())
```

```
Styles:
   block1_conv1
    shape:  (1, 64, 64)
    min:  0.0024659375
    max:  17750.146
    mean:  257.1598

   block2_conv1
    shape:  (1, 128, 128)
    min:  0.0
    max:  31234.38
    mean:  5096.993

   block3_conv1
    shape:  (1, 256, 256)
    min:  0.0
    max:  98666.7
    mean:  3463.4058

   block4_conv1
    shape:  (1, 512, 512)
    min:  3.2328677
    max:  966244.75
    mean:  76172.0

   block5_conv1
    shape:  (1, 512, 512)
    min:  0.0
    max:  14113.433
    mean:  415.8068

Contents:
   block5_conv2
    shape:  (1, 64, 60, 512)
    min:  0.0
    max:  832.2692
    mean:  6.642741
```

Function 4 - Gradient Descent and Loss Functions

Now that we're able to actually extract our features, we can implement the actual style transfer algorithm! We need to be able to compute the mean squared error for the image's output relative to each target. Following this, the weighted sum of each loss needs to be taken as well.

We first start off by actually loading our style and content image and then extracting their features. By calling on the class we defined earlier, we can specify the extracted feature based on the style and content as shown below.

Once this is done, we need an image to optimize. Remember that Equation 1 and Equation 4 are compared between the features (content or style) and the generated image. Thus, we need to generate this image. It can either be generated from the style image, content image, or even a generated white noise image. We can generate this using the tf.Variable function. This basically inititalizes a variable based on the shape and details of the input.

Since this generated image is of a float32 type, we need to clip this between 0 and 1. Thus we need a function clip_range to do this for us.

In the paper, Gatys uses an optimisation strategy to find the optimal loss. The paper uses L-BFGS but this isn't available in Tensorflow. We'll use ADAM instead. This will optimize Equation 7 (Total Loss Equation), which uses the gradient WRT the generated image's pixel values as an input. Thus, to optimize this image, we need a weighted combination of the two losses to get the total loss, hence Equation 7. We state these weights as alpha and beta.

Once all of these are defined, we can define a function to calculate our loss functions.

In [0]:

```
# Load the style and content images
style_img = load_img(style_path)
content_img = load_img(content_path)

# Extract the features from the style and content images
style_features = feature_extractor(style_img)['style']
content_features = feature_extractor(content_img)['content']

# The Tensorflow variable function initializes our image to be used for gradient descen
t.
# This image has to be the same size and type as the content image, which means that as
 long as it's loaded in before being
# called upon as the generated image, it should be fine to use. This is essentially the
 image that will be optimized.
# In case you want to use another image, you can uncomment the code below
'''
init_img = load_img(content_path)
generated_image = tf.Variable(init_img)
'''
generated_img = tf.Variable(content_img)

# Remember that all of our images are read in as a float32 type, so we have to define th
e range as [0, 1] to keep it within 255
def clip_range(img):
  return tf.clip_by_value(img, clip_value_min=0.0, clip_value_max=1.0)

# Choose an optimizer. The paper chose L-BFGS but Tensorflow doesn't have that, so we'll
use ADAM
optimizer = tf.optimizers.Adam(learning_rate=0.02, beta_1=0.99, epsilon=1e-8)

# Remember that we are trying to optimize the Total Loss function. However, there are va
lues that correspond to the style and content weight
# The Alpha value corresponds to content weight and the Beta corresponds to the style we
ight. Let's define these
alpha = 1e4
beta = 10

# MODEL UTILITIES - FUNCTION 4 - STYLE, CONTENT, & TOTAL LOSS FUNCTIONS

# Now we define our loss functions
def style_content_loss(outputs):
  style_outputs = outputs['style']
  content_outputs = outputs['content']

  # Equation 5
  # Define the style loss function
  style_loss = tf.add_n([tf.reduce_mean(tf.square(style_outputs[name] - style_features[n
ame])) for name in style_outputs.keys()])
  # Multiply the style loss by the weighted variable to get the weighted style loss
  style_loss *= beta / num_style_layers

  # Equation 1
  # Define the content loss function
  content_loss = tf.add_n([tf.reduce_mean(tf.square(content_outputs[name] - content_feat
ures[name])) for name in content_outputs.keys()])
  # Multiply the content loss by the weighted variable to get the weighted content loss
  content_loss *= alpha / num_content_layers

  # Equation 7
  # Define the total loss function
  total_loss = style_loss + content_loss
  return total_loss
```

All this little script does is check if the GPU is being used. It doesn't really do anything for our actual code.

In [54]:
```
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

```
Found GPU at: /device:GPU:0
```

Function 5 - Total Variational Loss

This part is aside from the paper. The basic implementation of this code produces a lot of high frequency artifacts. We want to get rid of these so we'll have to decrease these using an explicit regularization term on the high frequency components of the image. This is known as *total variational loss*. Technically, we can use Tensorflow's built in function to get this variational loss. However, we'll define the functions so that we UNDERSTAND how this works. At the end of the day, we essentially use tf.GradientTape to update the image using gradient descent

The high frequency component is essentially an edge detector. The regularization loss is associated with this is the sum of the squares of the values. The two functions are shown below

```
def high_pass(img):
  x_var = img[:, :, 1:, :] - img[:, :, :-1, :]
  y_var = img[:, 1:, :, :] - img[:, :-1, :, :]

  return x_var, y_var

def total_variational_loss(img):
  x_delta, y_delta = high_pass(img)
  return tf.reduce_sum(tf.abs(x_delta)) + tf.reduce_sum(tf.abs(y_delta))
```

In [0]:
```
# State the total variational weight
total_variational_weight= 30

@tf.function()
def train_step(img):
  # Tensorflow's GradientTape function performs automatic differentiation of the input
  with tf.GradientTape() as tape:
    outputs = feature_extractor(img)
    loss = style_content_loss(outputs)
    loss += total_variational_weight * tf.image.total_variation(img)

  # Apply the gradient by passing the loss and the generated image
  grad = tape.gradient(loss, img)

  # The gradient is optimized using the ADAM optimizer we declared earlier. This optimiz
es the
  # generated image using the gradient values
  optimizer.apply_gradients([(grad, img)])

  # The image is rewritten, being sure that the values all stay within the viable range
 of [0, 1]
  img.assign(clip_range(img))
```

In [57]:

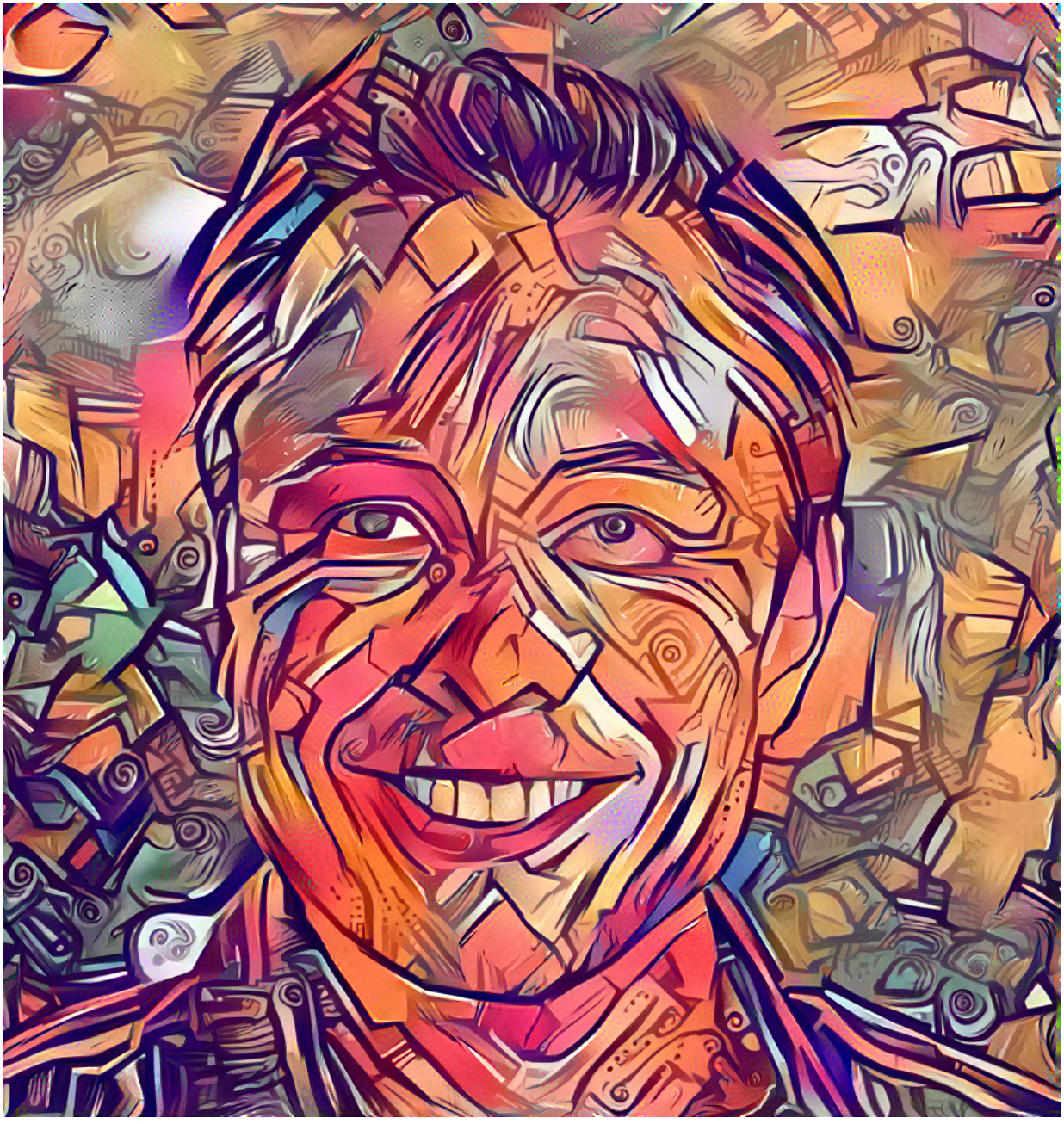
```
import time
from IPython import display
# Start the run time
start = time.time()

epochs = 100
iterations = 100

step = 0
for n in range(epochs):
  for m in range(iterations):
    step += 1
    train_step(generated_img)
    print(".", end=' ')
  display.clear_output(wait=True)
  display.display(deprocess_img(generated_img))
  print("Train step: {}".format(step))

end = time.time()
print("Total Time: {:.1f}".format(end-start))

# Save Image Files
file_name = '/content/drive/My Drive/Engineering Projects/EE8601-Project2/project_builds/output/stylized-image.png'
deprocess_img(generated_img).save(file_name)
```



```
Train step: 1000
Total Time: 181.7
```