Overview

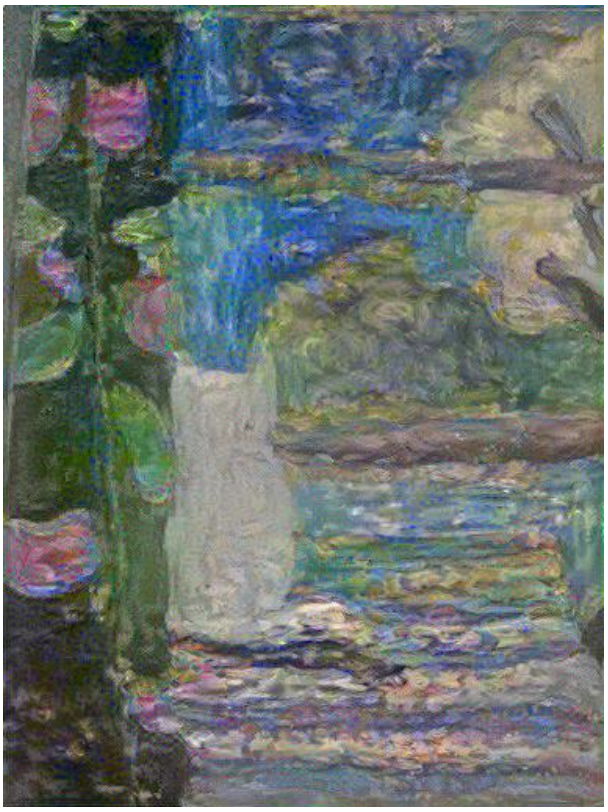
We will learn how to use deep learning to compose images in the style of another image (ever wish you could paint like Picasso or Van Gogh?). This is known as **neural style transfer**! This is a technique outlined in [Leon A. Gatys' paper, A Neural Algorithm of Artistic Style (https://arxiv.org/abs/1508.06576)](https://arxiv.org/abs/1508.06576).

But, what is neural style transfer?

Neural style transfer is an optimization technique used to take three images, a **content** image, a **style reference** image (such as an artwork by a famous painter), and the **input** image you want to style -- and blend them together such that the input image is transformed to look like the content image, but “painted” in the style of the style image.

For example, let’s take an image of this picture I drew as a second grader and Claude Monet's famous water lily:

Now how would it look like if the 2nd grader me can draw someing with impressionism style:



The principle of neural style transfer is to define two distance functions, one that describes how different the content of two images are , $L_{content}$, and one that describes the difference between two images in terms of their style, L_{style} . Then, given three images, a desired style image, a desired content image, and the input image (initialized with the content image), we try to transform the input image to minimize the content distance with the content image and its style distance with the style image. In summary, we'll take the base input image, a content image that we want to match, and the style image that we want to match. We'll transform the base input image by minimizing the content and style distances (losses) with backpropagation, creating an image that matches the content of the content image and the style of the style image.

Background: For details on how style transfer works in neural network, check the following:

- Read Gatys' paper (https://arxiv.org/abs/1508.06576) - we'll explain along the way, but the paper will provide a more thorough understanding of the task
- Understand reducing loss with gradient descent (https://developers.google.com/machine-learning/crash-

Download Images

In [1]:
```
import os
img_dir = '/tmp/nst'
if not os.path.exists(img_dir):
    os.makedirs(img_dir)
!cp *.jpg /tmp/nst/
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/common
s/1/1a/Koi_fish_golden.jpg
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/common
s/5/5d/Monet_Water_Lilies_1916.jpg
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/common
s/d/d7/Green_Sea_Turtle_grazing_seagrass.jpg
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/common
s/0/0a/The_Great_Wave_off_Kanagawa.jpg
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/common
s/b/b4/Vassily_Kandinsky%2C_1913_-_Composition_7.jpg
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/common
s/0/00/Tuebingen_Neckarfront.jpg
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/common
s/6/68/Pillars_of_creation_2014_HST_WFC3-UVIS_full-res_denoised.jpg
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/common
s/a/a3/Van_Gogh_Sunflowers_Neue_Pinakothek_8672.jpg
!wget --quiet -P /tmp/nst/ https://upload.wikimedia.org/wikipedia/common
s/thumb/e/ea/Van_Gogh_-_Starry_Night_-_Google_Art_Project.jpg/1024px-Van
_Gogh_-_Starry_Night_-_Google_Art_Project.jpg
```

Import and configure modules

In [2]:
```
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['figure.figsize'] = (10,10)
mpl.rcParams['axes.grid'] = False

import numpy as np
from PIL import Image
import time
import functools
```

In [3]:
```
import tensorflow as tf
import tensorflow.contrib.eager as tfe

from tensorflow.python.keras.preprocessing import image as kp_image
from tensorflow.python.keras import models
from tensorflow.python.keras import losses
from tensorflow.python.keras import layers
from tensorflow.python.keras import backend as K
```

We'll begin by enabling [eager execution (https://www.tensorflow.org/guide/eager)](https://www.tensorflow.org/guide/eager). Eager execution allows us to work through this technique in the clearest and most readable way.

In [4]:
```
tf.enable_eager_execution()
print("Eager execution: {}".format(tf.executing_eagerly()))
```

```
Eager execution: True
```

In [5]:
```
# Set up some global values here
content_path = 'IMG_20181003_124459.jpg'
style_path = '/tmp/nst/Monet_Water_Lilies_1916.jpg'
```

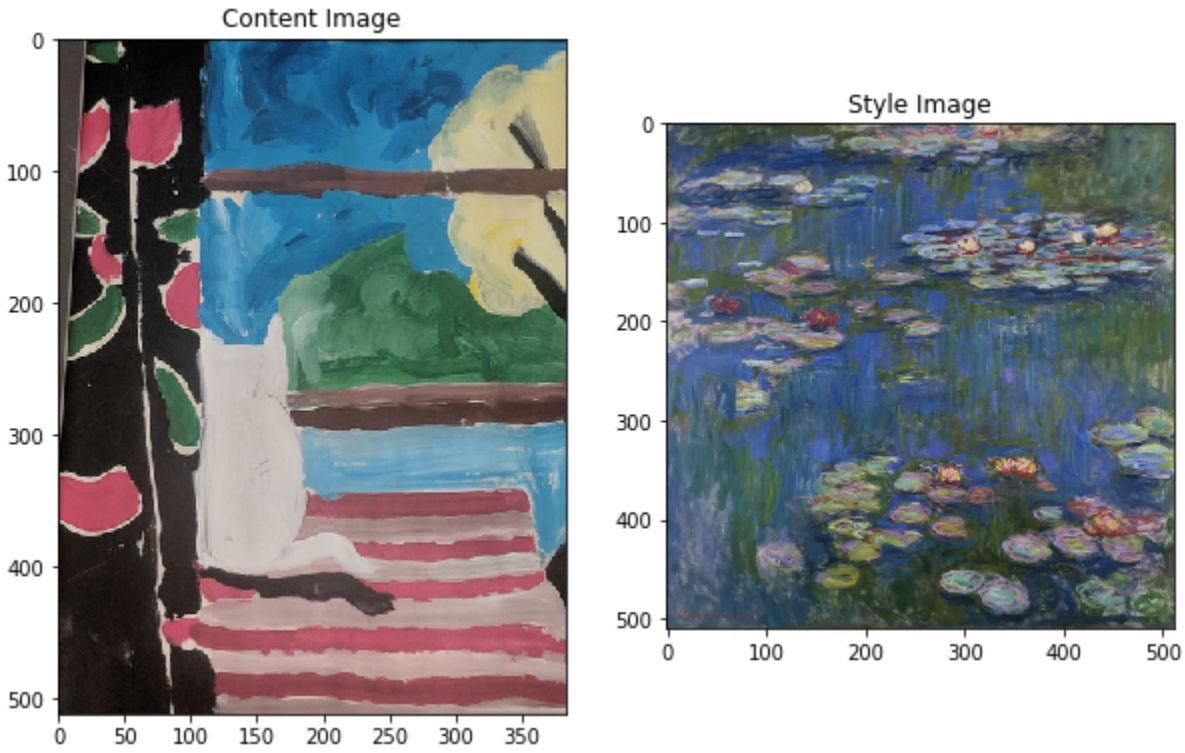
Visualize the input

In [6]:
```
def load_img(path_to_img):
  max_dim = 512
  img = Image.open(path_to_img)
  long = max(img.size)
  scale = max_dim/long
  img = img.resize((round(img.size[0]*scale), round(img.size[1]*scale)),
 Image.ANTIALIAS)

  img = kp_image.img_to_array(img)

  # We need to broadcast the image array such that it has a batch dimens
ion
  img = np.expand_dims(img, axis=0)
  return img
```

In [7]:
```
def imshow(img, title=None):
  # Remove the batch dimension
  out = np.squeeze(img, axis=0)
  # Normalize for display
  out = out.astype('uint8')
  plt.imshow(out)
  if title is not None:
    plt.title(title)
  plt.imshow(out)
```

These are input content and style images. We hope to "create" an image with the content of our content image, but with the style of the style image.

In [8]:
```
plt.figure(figsize=(10,10))

content = load_img(content_path).astype('uint8')
style = load_img(style_path).astype('uint8')

plt.subplot(1, 2, 1)
imshow(content, 'Content Image')

plt.subplot(1, 2, 2)
imshow(style, 'Style Image')
plt.show()
```



Prepare the data

Let's create methods that will allow us to load and preprocess our images easily. We perform the same preprocessing process as are expected according to the VGG training process. VGG networks are trained on image with each channel normalized by `mean = [103.939, 116.779, 123.68]` and with channels BGR.

In [9]:
```
def load_and_process_img(path_to_img):
  img = load_img(path_to_img)
  img = tf.keras.applications.vgg19.preprocess_input(img)
  return img
```

In order toview the outputs of our optimization, we are required to perform the inverse preprocessing step. Furthermore, since our optimized image may take its values anywhere between $-\infty$ and ∞, we must clip to maintain our values from within the 0-255 range.

In [10]:
```
def deprocess_img(processed_img):
  x = processed_img.copy()
  if len(x.shape) == 4:
    x = np.squeeze(x, 0)
  assert len(x.shape) == 3, ("Input to deprocess image must be an image
 of "
                             "dimension [1, height, width, channel] or
 [height, width, channel]")
  if len(x.shape) != 3:
    raise ValueError("Invalid input to deprocessing image")

  # perform the inverse of the preprocessiing step
  x[:, :, 0] += 103.939
  x[:, :, 1] += 116.779
  x[:, :, 2] += 123.68
  x = x[:, :, ::-1]

  x = np.clip(x, 0, 255).astype('uint8')
  return x
```

Define content and style representationst

In [11]:
```
# Content layer where will pull our feature maps
content_layers = ['block5_conv2']

# Style layer we are interested in
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1'
               ]

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

Build the Model

In this case, we load VGG19 (https://keras.io/applications/#vgg19), and feed in our input tensor to the model. This will allow us to extract the feature maps (and subsequently the content and style representations) of the content, style, and generated images.

In [12]:
```
def get_model():
  """ Creates our model with access to intermediate layers.

  This function will load the VGG19 model and access the intermediate la
yers.
  These layers will then be used to create a new model that will take in
put image
  and return the outputs from these intermediate layers from the VGG mod
el.

  Returns:
    returns a keras model that takes image inputs and outputs the style
 and
      content intermediate layers.
  """
  # Load our model. We load pretrained VGG, trained on imagenet data
  vgg = tf.keras.applications.vgg19.VGG19(include_top=False, weights='im
agenet')
  vgg.trainable = False
  # Get output layers corresponding to style and content layers
  style_outputs = [vgg.get_layer(name).output for name in style_layers]
  content_outputs = [vgg.get_layer(name).output for name in content_laye
rs]
  model_outputs = style_outputs + content_outputs
  # Build model
  return models.Model(vgg.input, model_outputs)
```

Define and create our loss functions (content and style distances)

Content Loss

Our content loss definition is actually quite simple. We'll pass the network both the desired content image and our base input image. This will return the intermediate layer outputs (from the layers defined above) from our model. Then we simply take the euclidean distance between the two intermediate representations of those images.

More formally, content loss is a function that describes the distance of content from our output image x and our content image, p. Let C_{nn} be a pre-trained deep convolutional neural network. Again, in this case we use VGG19 (https://keras.io/applications/#vgg19). Let X be any image, then $C_{nn}(X)$ is the network fed by X. Let $F_{ij}^{l}(x) \in C_{nn}(x)$ and $P_{ij}^{l}(p) \in C_{nn}(p)$ describe the respective intermediate feature representation of the network with inputs x and p at layer l. Then we describe the content distance (loss) formally as:

$$L_{content}^{l}(p, x) = \sum_{i,j} (F_{ij}^{l}(x) - P_{ij}^{l}(p))^2$$

Computing content loss

We will actually add our content losses at each desired layer. This way, each iteration when we feed our input image through the model (which in eager is simply `model(input_image)` !) all the content losses through the model will be properly compute and because we are executing eagerly, all the gradients will be computed.

In [13]:
```
def get_content_loss(base_content, target):
  return tf.reduce_mean(tf.square(base_content - target))
```

Style Loss

Computing style loss is a bit more involved, but follows the same principle, this time feeding our network the base input image and the style image. However, instead of comparing the raw intermediate outputs of the base input image and the style image, we instead compare the Gram matrices of the two outputs.

Mathematically, we describe the style loss of the base input image, x, and the style image, a, as the distance between the style representation (the gram matrices) of these images. We describe the style representation of an image as the correlation between different filter responses given by the Gram matrix G^l, where G^l_{ij} is the inner product between the vectorized feature map i and j in layer l. We can see that G^l_{ij} generated over the feature map for a given image represents the correlation between feature maps i and j.

To generate a style for our base input image, we perform gradient descent from the content image to transform it into an image that matches the style representation of the original image. We do so by minimizing the mean squared distance between the feature correlation map of the style image and the input image. The contribution of each layer to the total style loss is described by

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G^l_{ij} - A^l_{ij})^2$$

where G^l_{ij} and A^l_{ij} are the respective style representation in layer l of x and a. N_l describes the number of feature maps, each of size $M_l = height * width$. Thus, the total style loss across each layer is

$$L_{style}(a, x) = \sum_{l \in L} w_l E_l$$

where we weight the contribution of each layer's loss by some factor w_l. In our case, we weight each layer equally ($w_l = \frac{1}{|L|}$)

Computing style loss

Again, we implement our loss as a distance metric .

In [14]:
```
def gram_matrix(input_tensor):
  # We make the image channels first
  channels = int(input_tensor.shape[-1])
  a = tf.reshape(input_tensor, [-1, channels])
  n = tf.shape(a)[0]
  gram = tf.matmul(a, a, transpose_a=True)
  return gram / tf.cast(n, tf.float32)

def get_style_loss(base_style, gram_target):
  """Expects two images of dimension h, w, c"""
  # height, width, num filters of each layer
  # We scale the loss at a given layer by the size of the feature map an
d the number of filters
  height, width, channels = base_style.get_shape().as_list()
  gram_style = gram_matrix(base_style)

  return tf.reduce_mean(tf.square(gram_style - gram_target))# / (4. * (c
hannels ** 2) * (width * height) ** 2)
```

Apply style transfer to our images

Run Gradient Descent

We'll define a little helper function that will load our content and style image, feed them forward through our network, which will then output the content and style feature representations from our model.

In [15]:
```
def get_feature_representations(model, content_path, style_path):
  """Helper function to compute our content and style feature representa
tions.

  This function will simply load and preprocess both the content and sty
le
  images from their path. Then it will feed them through the network to
 obtain
  the outputs of the intermediate layers.

  Arguments:
    model: The model that we are using.
    content_path: The path to the content image.
    style_path: The path to the style image

  Returns:
    returns the style features and the content features.
  """
  # Load our images in
  content_image = load_and_process_img(content_path)
  style_image = load_and_process_img(style_path)

  # batch compute content and style features
  style_outputs = model(style_image)
  content_outputs = model(content_image)


  # Get the style and content feature representations from our model
  style_features = [style_layer[0] for style_layer in style_outputs[:num
_style_layers]]
  content_features = [content_layer[0] for content_layer in content_outp
uts[num_style_layers:]]
  return style_features, content_features
```

Computing the loss and gradients

Here we use **tf.GradientTape** (https://www.tensorflow.org/programmers_guide/eager#computing_gradients) to compute the gradient. It allows us to take advantage of the automatic differentiation available by tracing operations for computing the gradient later. It records the operations during the forward pass and then is able to compute the gradient of our loss function with respect to our input image for the backwards pass.

In [16]:
```
def compute_loss(model, loss_weights, init_image, gram_style_features, c
ontent_features):
  """This function will compute the loss total loss.

  Arguments:
    model: The model that will give us access to the intermediate layers
    loss_weights: The weights of each contribution of each loss functio
n.
      (style weight, content weight, and total variation weight)
    init_image: Our initial base image. This image is what we are updati
ng with
      our optimization process. We apply the gradients wrt the loss we a
re
      calculating to this image.
    gram_style_features: Precomputed gram matrices corresponding to the
      defined style layers of interest.
    content_features: Precomputed outputs from defined content layers of

      interest.

  Returns:
    returns the total loss, style loss, content loss, and total variatio
nal loss
  """
  style_weight, content_weight = loss_weights

  # Feed our init image through our model. This will give us the content
 and
  # style representations at our desired layers. Since we're using eager
  # our model is callable just like any other function!
  model_outputs = model(init_image)

  style_output_features = model_outputs[:num_style_layers]
  content_output_features = model_outputs[num_style_layers:]

  style_score = 0
  content_score = 0

  # Accumulate style losses from all layers
  # Here, we equally weight each contribution of each loss layer
  weight_per_style_layer = 1.0 / float(num_style_layers)
  for target_style, comb_style in zip(gram_style_features, style_output_
features):
    style_score += weight_per_style_layer * get_style_loss(comb_style[0
], target_style)

  # Accumulate content losses from all layers
  weight_per_content_layer = 1.0 / float(num_content_layers)
  for target_content, comb_content in zip(content_features, content_outp
ut_features):
    content_score += weight_per_content_layer* get_content_loss(comb_con
tent[0], target_content)

  style_score *= style_weight
  content_score *= content_weight
```

```
  # Get total loss
  loss = style_score + content_score
  return loss, style_score, content_score
```

Then computing the gradients is easy:

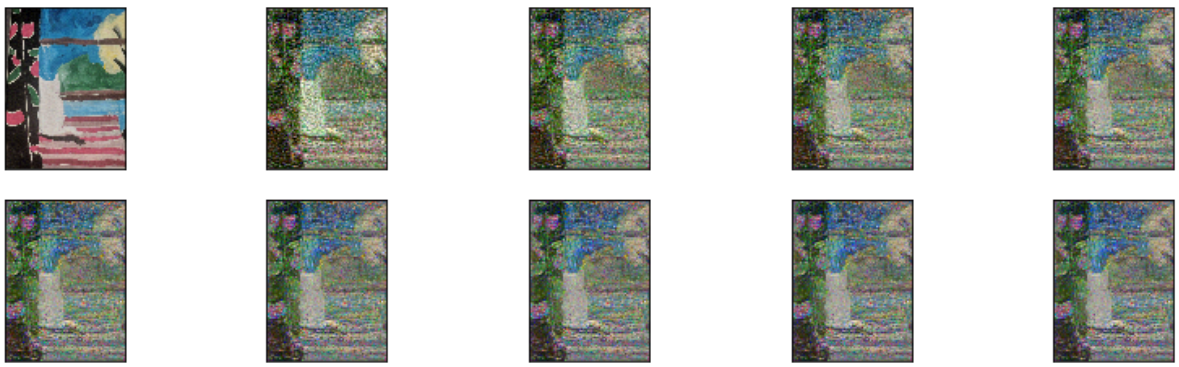
In [17]:
```
def compute_grads(cfg):
  with tf.GradientTape() as tape:
    all_loss = compute_loss(**cfg)
  # Compute gradients wrt input image
  total_loss = all_loss[0]
  return tape.gradient(total_loss, cfg['init_image']), all_loss
```

Optimization loop

In [18]:
```
import IPython.display

def run_style_transfer(content_path,
                       style_path,
                       num_iterations=1000,
                       content_weight=1e3,
                       style_weight=1e-2):
  # We don't need to (or want to) train any layers of our model, so we s
et their
  # trainable to false.
  model = get_model()
  for layer in model.layers:
    layer.trainable = False

  # Get the style and content feature representations (from our specifie
d intermediate layers)
  style_features, content_features = get_feature_representations(model,
content_path, style_path)
  gram_style_features = [gram_matrix(style_feature) for style_feature in
 style_features]

  # Set initial image
  init_image = load_and_process_img(content_path)
  init_image = tfe.Variable(init_image, dtype=tf.float32)
  # Create our optimizer
  opt = tf.train.AdamOptimizer(learning_rate=5, beta1=0.99, epsilon=1e-1
)

  # For displaying intermediate images
  iter_count = 1

  # Store our best result
  best_loss, best_img = float('inf'), None

  # Create a nice config
  loss_weights = (style_weight, content_weight)
  cfg = {
      'model': model,
      'loss_weights': loss_weights,
      'init_image': init_image,
      'gram_style_features': gram_style_features,
      'content_features': content_features
  }

  # For displaying
  num_rows = 2
  num_cols = 5
  display_interval = num_iterations/(num_rows*num_cols)
  start_time = time.time()
  global_start = time.time()

  norm_means = np.array([103.939, 116.779, 123.68])
  min_vals = -norm_means
  max_vals = 255 - norm_means

  imgs = []
```

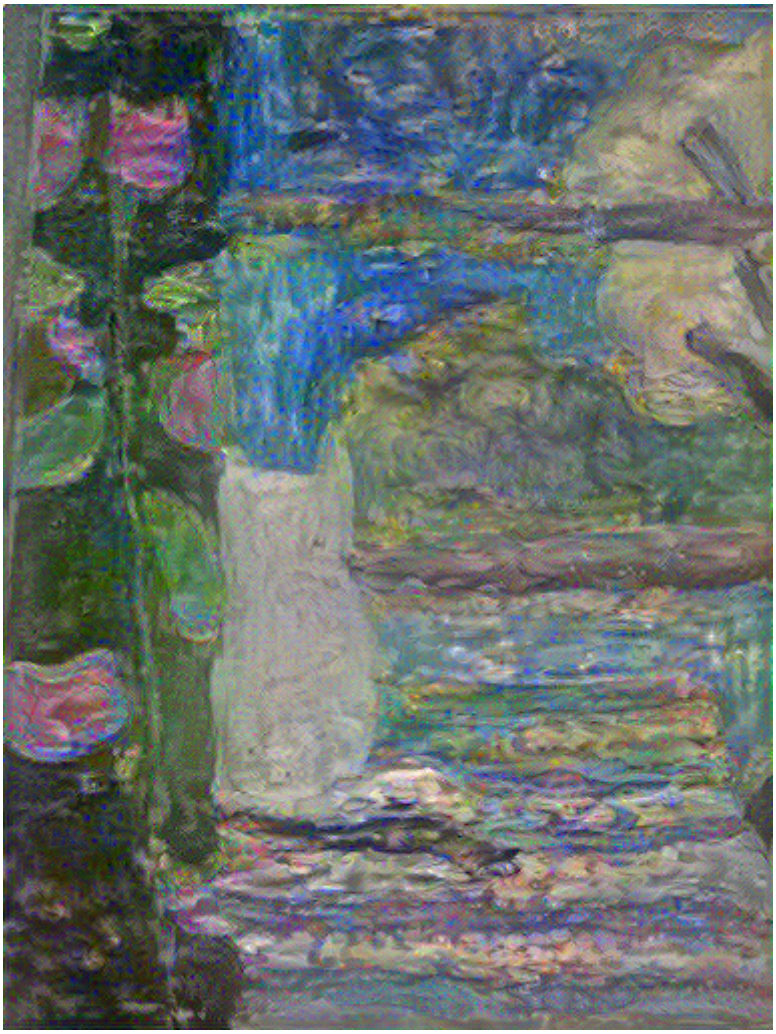
```
  for i in range(num_iterations):
    grads, all_loss = compute_grads(cfg)
    loss, style_score, content_score = all_loss
    opt.apply_gradients([(grads, init_image)])
    clipped = tf.clip_by_value(init_image, min_vals, max_vals)
    init_image.assign(clipped)
    end_time = time.time()

    if loss < best_loss:
      # Update best loss and best image from total loss.
      best_loss = loss
      best_img = deprocess_img(init_image.numpy())

    if i % display_interval== 0:
      start_time = time.time()

      # Use the .numpy() method to get the concrete numpy array
      plot_img = init_image.numpy()
      plot_img = deprocess_img(plot_img)
      imgs.append(plot_img)
      IPython.display.clear_output(wait=True)
      IPython.display.display_png(Image.fromarray(plot_img))
      print('Iteration: {}'.format(i))
      print('Total loss: {:.4e}, '
            'style loss: {:.4e}, '
            'content loss: {:.4e}, '
            'time: {:.4f}s'.format(loss, style_score, content_score, tim
e.time() - start_time))
  print('Total time: {:.4f}s'.format(time.time() - global_start))
  IPython.display.clear_output(wait=True)
  plt.figure(figsize=(14,4))
  for i,img in enumerate(imgs):
      plt.subplot(num_rows,num_cols,i+1)
      plt.imshow(img)
      plt.xticks([])
      plt.yticks([])

  return best_img, best_loss
```

In [19]:
```
best, best_loss = run_style_transfer(content_path,
                                     style_path, num_iterations=1000)
```



In [20]:
```
Image.fromarray(best)
```

Out[20]:



To download the image from Colab uncomment the following code:

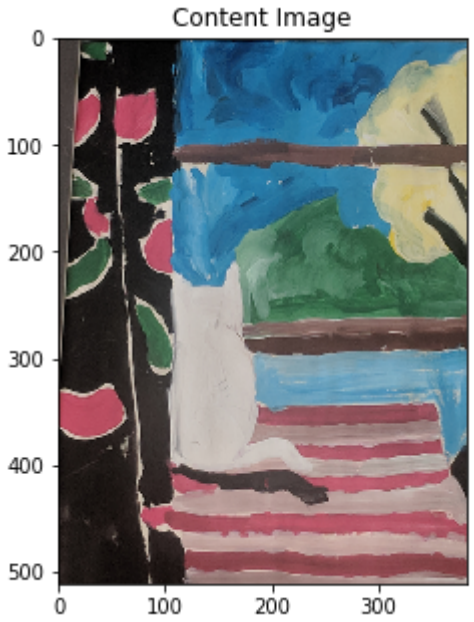
In [21]:
```
best_image = Image.fromarray(best)
best_image.save("best_cat_lily.jpg")
```

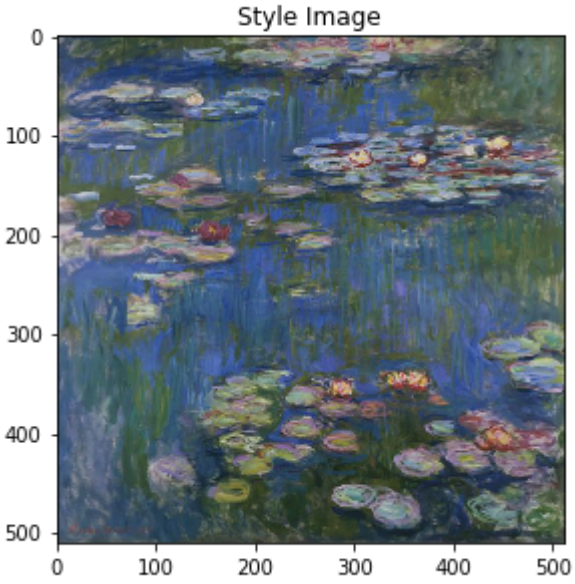
Visualize outputs

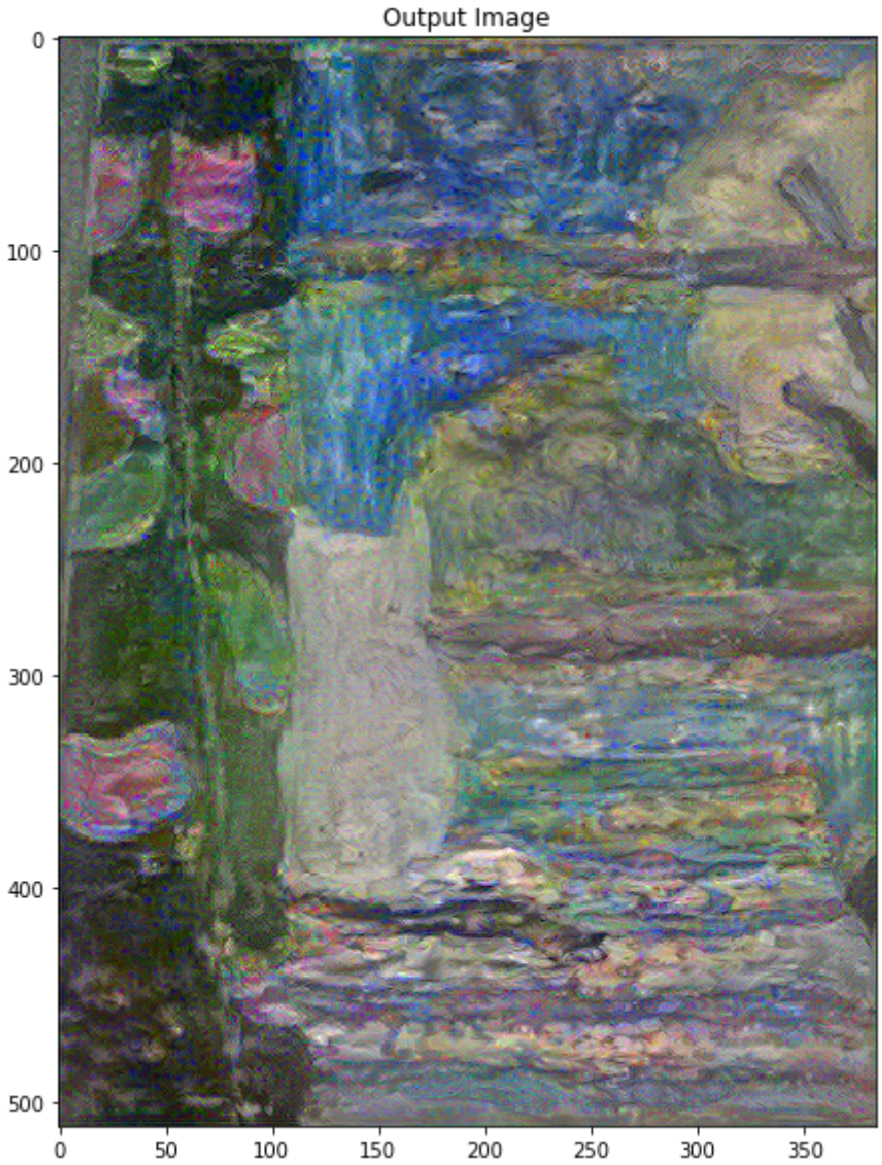
We "deprocess" the output image in order to remove the processing that was applied to it.

In [22]:
```
def show_results(best_img, content_path, style_path, show_large_final=Tr
ue):
  plt.figure(figsize=(10, 5))
  content = load_img(content_path)
  style = load_img(style_path)

  plt.subplot(1, 2, 1)
  imshow(content, 'Content Image')

  plt.subplot(1, 2, 2)
  imshow(style, 'Style Image')

  if show_large_final:
    plt.figure(figsize=(10, 10))

    plt.imshow(best_img)
    plt.title('Output Image')
    plt.show()
```

In [23]:
```
show_results(best, content_path, style_path)
```







Try it on other images

In [28]:
```
best_pond_monet, best_loss = run_style_transfer('/tmp/nst/IMG_20181003_124856.jpg',
                                                '/tmp/nst/Monet_Water_Lilies_1916.jpg')
```

In [29]:
```
show_results(best_pond_monet,
             '/tmp/nst/IMG_20181003_124856.jpg',
             '/tmp/nst/Monet_Water_Lilies_1916.jpg')
Image.fromarray(best_pond_monet).save('/tmp/nst/best_pond_monet.jpg')
```