Importing Modules

Let us begin by importing the necessary modules. We shall implement the project using Tensorflow 2.

**INSTRUCTIONS**

* Import tensorflow as tf, and check its version.
* import << your code comes here >> as << your code comes here >>
* print(tf.\_\_version\_\_)
* Import import IPython.display as display.
* import IPython.display as << your code comes here >>
* Import matplotlib.pyplot as plt
* import << your code comes here >> as << your code come here >>
* Import numpy as np.
* import << your code comes here >> as << your code comes here >>
* Import import PIL.Image.
* import << your code comes here >>
* Import time module.

import << your code comes here >>

Load the Images

Let us first load and visualize the content and style images we want to work with.

We shall do that by:

* defining the function load\_img to load an image and limit its maximum dimension to 512 pixels.
* creating a simple function imshow to display an image

We shall create the function load\_img in this slide, and create the imshow function in the next slide.

**Note:**

* tf.io.read\_file(path\_to\_img) reads the entire contents of the input filename.

It returns a Tensor of type string which contains bit-representations of the image.

* tf.image.decode\_image(img, channels=3) detects whether an image is a BMP, GIF, JPEG, or PNG, and performs the appropriate operation to convert the input bytes string into a Tensor of type uint8. For uint8, the minimum value is 0 and the maximum value is 255.

Images that are represented using floating-point values are expected to have values in the range [0,1).

Image data stored in integer data types are expected to have values in the range [0,MAX], where MAX is the largest positive representable number for the data type.

This op converts between data types, scaling the values appropriately before casting.

* tf.image.convert\_image\_dtype(img, tf.float32) converts the image tensor values to the specified dtype, scaling its values if needed.
* tf.shape(input\_tensor) returns the shape of a tensor.
* tf.cast casts a tensor to a new type.
* tf.cast(tf.shape(img)[:-1], tf.float32) converts all the dimensions - except the last dimension - of the img to float32 data type.
* tf.image.resize resizes image to a specified size.
* tf.newaxis is used to increase the dimension of the existing array by one more dimension.

**INSTRUCTIONS**

* Mention the content-image path "/cxldata/dlcourse/dog.jpg" in the variable content\_path.
* << your code comes here >> = "/cxldata/dlcourse/dog.jpg"
* Mention the style-image path "/cxldata/dlcourse/moon.jpg" in the variable style\_path.
* << your code comes here >> = "/cxldata/dlcourse/moon.jpg"
* We shall now define the function load\_img as follows:
  + Set max\_dim to 512 to set the maximum dimensions of the input image.
  + Read the image from the given path using tf.image.decode\_image.
  + Convert the image pixels to float32 using tf.image.convert\_image\_dtype.
  + Get the maximum dimension long\_dim from the shape of the input image. The max\_dim is divided by this long\_dim to get the scale measure so that the scale could be used to resize the image.
  + The shape is multiplied by scale and the integer result is used along with tf.image.resize to resize the image.

The above steps are Pythonically implemented as follows. So use the below code to load the image.

def load\_img(path\_to\_img):

max\_dim = 512

img = tf.io.read\_file(path\_to\_img)

img = tf.image.decode\_image(img, channels=3)

img = tf.image.convert\_image\_dtype(img, tf.float32)

shape = tf.cast(tf.shape(img)[:-1], tf.float32)

long\_dim = max(shape)

scale = max\_dim / long\_dim

new\_shape = tf.cast(shape \* scale, tf.int32)

img = tf.image.resize(img, new\_shape)

img = img[tf.newaxis, :]

return img

* Now call the load\_image function to load the content and style images by passing the paths to each of the images as input arguments respectively.
* content\_image = << your code comes here >>(content\_path)

style\_image = << your code comes here >>(style\_path)

Visualize the Images

We shall visualize the images that we have loaded previously. We shall define the imshow function to do the same.

**Note:**

* tf.squeeze removes dimensions of size 1 from the shape of a tensor.

**INSTRUCTIONS**

* We shall define the imshow function. We shall pass the image and the title as input arguments to the function.
  + We shall squeeze the fourth dimension, if any.
  + We shall also display the title of the image, if any.
  + def imshow(image, title=None):
  + if len(image.shape) > 3:
  + image = tf.squeeze(image, axis=0)
  + plt.imshow(image)
  + if title:
  + plt.title(title)
* Let us display the content and style images side-by-side. Call the imshow function to display the content and style images.
* plt.subplot(1, 2, 1)
* << your code comes here >>(content\_image, 'Content Image')
* plt.subplot(1, 2, 2)
* << your code comes here >>(style\_image, 'Style Image')
* Let us also print the shapes of the content and style images using shape.
* print("Content image shape: ", << your code comes here >>)

print("Style image shape: ", << your code comes here >>)

Loading Pre-trained VGG19

Let us load VGG19 previously trained to classify Imaagenet data. Let us test run it on our image to ensure it's used correctly.

**Note**:

* tf.keras.applications are canned architectures with pre-trained weights.
* tf.keras.applications.VGG19 is VGG19 model for Keras.
* tf.keras.applications.vgg19.preprocess\_input returns the images converted from RGB to BGR, then each color channel is zero-centered with respect to the ImageNet dataset, without scaling.
* tf.image.resize resizes image to size using the specified method.
* tf.keras.applications.vgg19.decode\_predictions decodes the prediction of an ImageNet model.
* vgg.layers returns the list of all the layers in the vgg model.

**INSTRUCTIONS**

* Get the preprocessed form of the content\_image using tf.keras.applications.vgg19.preprocess\_input.
* x = << your code comes here >>(content\_image\*255)
* We will have to set include\_top=True since we want to cross-check if the network is able to correctly predict our content\_image. When setting include\_top=True and loading imagenet weights, input\_shape should be (224, 224, 3). So, let us resize it using tf.image.resize.
* x = tf.image.resize(x, (224, 224))
* Now, let us instantiate the VGG19 model as follows, using tf.keras.applications.VGG19:
* vgg = << your code comes here >>(include\_top=True, weights='imagenet')
* Pass x, that is the preprocessed and resized content image, to the vgg and get the prediction\_probabilities. We expect the shape of this prediction\_probabilities to be of 1000 dimensions, as VGG19 on the Imagenet database is trained to classify 1000 classes.
* prediction\_probabilities = vgg(<< your code comes here >>)
* print(prediction\_probabilities.shape)
* Let us print the top 5 predicted classes of our content image using tf.keras.applications.vgg19.decode\_predictions.
* predicted\_top\_5 = tf.keras.applications.vgg19.decode\_predictions( prediction\_probabilities.numpy() )[0]
* print([(class\_name, prob) for (number, class\_name, prob) in predicted\_top\_5])

Observe that the classes predicted are all different breeds of dogs (you could google these classes to cross-check though :-D ).

This assures that the network is able to recognize that it is an image of a dog! Voila, the network is already powerful enough to recognize the main features of the given image! So we can assuredly go forward to use this to extract the features (content from content image and style from the style image) of the input image!

* Now, let us load the VGG19 network without the classification head (by setting include\_top to False) , just to see the list of all the layer names.
* vgg = tf.keras.applications.VGG19(include\_top=<< your code comes here >>, weights='imagenet')
* for layer in vgg.layers:
* print(layer.name)
* Now, let us choose intermediate layers from the network to represent the style and content of the image:
* content\_layers = ['block5\_conv2']
* style\_layers = ['block1\_conv1',
* 'block2\_conv1',
* 'block3\_conv1',
* 'block4\_conv1',

'block5\_conv1']

Getting model with the specified VGG19 Layers

We have defined from which layers we are going to extract the content of the image, and from which layers we are going to extract the style of the image. We shall define a function vgg\_layers to do the same.

We will be getting an instance VGG19, and we will be getting the weights of these layers so that these weights act as the feature extractors and these features will be used by use as discussed previously.

**Note:**

* Say vgg is an instance of tf.keras.applications.VGG19. Then, vgg.get\_layers(layer\_name).output returns the weights of the given layer. layer\_name is the layer name in string format.
* tf.keras.Model takes the input layer and list of other layers( regarded as output layers) as input arguments, and returns the model with these layers.

**INSTRUCTIONS**

* Define the vgg\_layers function and pass the layers as input argument. In this function, we will:
  + get the instance vgg of tf.keras.applications.VGG19(include\_top=False, weights='imagenet'). Remember, we have to set include\_top=False, as this is not a classification problem, but we just want to use the network as a feature extractor to extract the content and the style.
  + set the trainable to False since we are not training the network but we will be using the same pre-trained.
  + use vgg.get\_layer(name).output to get weights of that layer.
  + finally, we will return the model with the specified layer weights.

All the above steps are Pythonically implemented in the below function. Use the below code to do the same.

def vgg\_layers(layer\_names):

vgg = tf.keras.applications.VGG19(include\_top=False, weights='imagenet')

vgg.trainable = False

outputs = [vgg.get\_layer(name).output for name in layer\_names]

model = tf.keras.Model([vgg.input], outputs)

return model

* Now, call the vgg\_layer function and pass the style\_layers as the input argument to the function to get the style extractor model.
* style\_extractor = << your code comes here >>(style\_layers)
* Now, pass the style\_image\*255 to this style\_extractor. This returns the layer-wise names and outputs.
* style\_outputs = style\_outputs = style\_extractor(style\_image\*255)
* Let us look at the statistics of each layer's output:
* for name, output in zip(style\_layers, style\_outputs):
* print(name)
* print(" shape: ", output.numpy().shape)
* print(" min: ", output.numpy().min())
* print(" max: ", output.numpy().max())

print(" mean: ", output.numpy().mean())

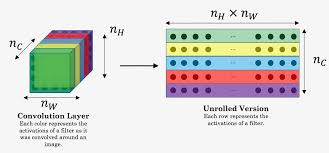
print()

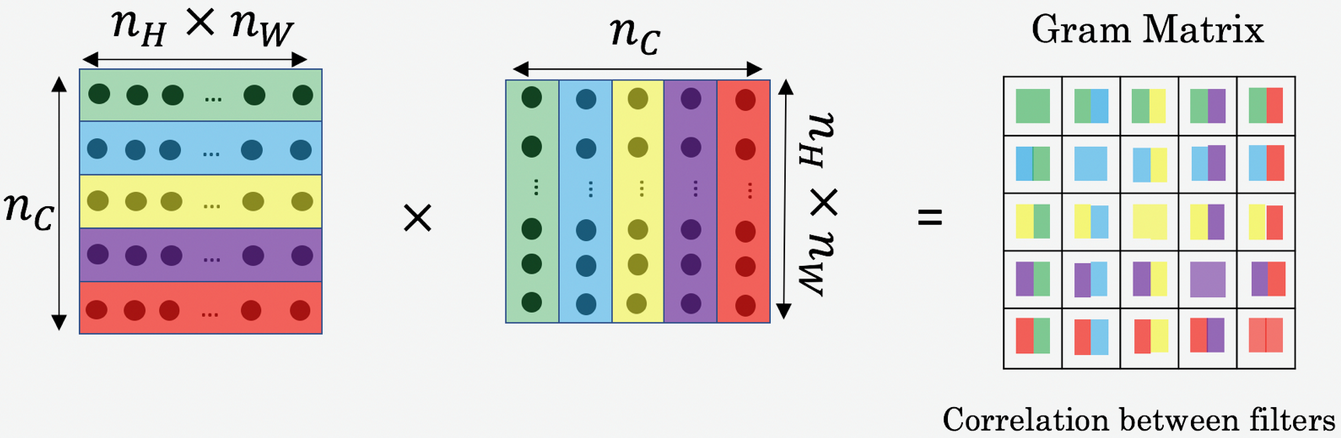
Calculating Style

The content of an image is represented by the values of the intermediate feature maps.

On the other hand, the style of an image can be described by the means and correlations across the different feature maps. We calculate a Gram matrix that includes this information by taking the outer product of the feature vector with itself at each location and averaging that outer product over all locations. This Gram matrix can be calculated for a particular layer as:

enter image description here





This can be implemented concisely using the tf.linalg.einsum function:

We shall now define a function to calculate the gram matrix, given the style activations from a layer.

**INSTRUCTIONS**

* Define the function gram\_matrix and pass the input\_tensor, which would potentially be the output activations from an intermediate style layer of the input image. We shall implement the following steps in the function:
  + Calculate the numerator of the above-mentioned gram matrix formula.
  + Divide the thus obtained result with the multiplication of the width and height of the image.

Use the below code to calculate the gram matrix of given input tensor:

def gram\_matrix(input\_tensor):

result = 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)

return result/(num\_locations)

Extracting style and content - 1

We are going to define a class to extract the style and content of a given image.

* So basically, we build a model that returns the style and content tensors.
* Inside of Keras the Model class is the root class used to define a model architecture. Since Keras utilizes object-oriented programming, we can actually subclass the Model class and then insert our architecture definition.
* Model subclassing is fully-customizable and enables you to implement your own custom forward-pass of the model.
* We are going to define our custom style-content extractor for the given image by subclassing tf.keras.models.Model. We do that by:
  + Define \_\_init()\_\_:
    - Call the super().\_\_init\_\_(), the constructor of tf.keras.models.Model which is the parent class.
    - Next, we shall set self.vgg to the vgg\_layers function which we have previously defined. This returns the custom model with the specified style layers and content layers.
    - Define the layers for content and style extraction in our custom model.
    - It is important to set the trainable to False, as we want to use the same VGG19 weights trained on ImageNet Database.
  + Define call method:

call method is regarded as the forward pass of the model. We would customize it.

In our scenario, we define call such that we will be returned the gram-matrices representing the style of the image and, the content of the image will be returned. We shall implement the following steps in the call function:

* + - We would first scale the input image values to the range [0,255].
    - Then, we shall preprocess the image using tf.keras.applications.vgg19.preprocess\_input.
    - Next, we shall pass this preprocessed input to our custom model - self.vgg - we defined with the specified style and content layers using vgg\_layers funtion. This returns the outputs, which contains the style and content matrices for our input image.
    - Now that we have got the style representation matrices, we shall proceed to calculate the gram-matrices of each of the style layers. We shall call gram\_matrix function to do this.
    - Finally, we shall return a dictionary holding the content representations and the layer-wise gram-matrices for style representations of the given input image.

**Note:**

* super().\_\_init\_\_() calls our parent constructor. From there on, our layers are defined as instance attributes. Attributes in Python use the self keyword and are typically (but not always) defined in a constructor.
* tf.keras.applications.vgg19.preprocess\_input returns preprocessed NumPy array or a tf.Tensor with type float32. The images are converted from RGB to BGR, then each color channel is zero-centered with respect to the ImageNet dataset, without scaling.
* call : Once the layers of our choice are defined, we can then define the network topology/graph inside the call function which is used to perform a forward-pass.

**INSTRUCTIONS**

* Use the following code to define the StyleContentModel, which returns the style and content representations of the given input image. Each instruction in the below code is just a Pythonic implementation of the above-mentioned description. So, make sure to understand each and every line.
* class StyleContentModel(tf.keras.models.Model):
* def \_\_init\_\_(self, style\_layers, content\_layers):
* super().\_\_init\_\_()
* self.vgg = vgg\_layers(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):
* 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}

Extracting style and content - 2

We shall now use the class StyleContentModel defined in the previous slide.

When called on an image, this model returns the gram matrix (style) of the style\_layers and content of the content\_layers:

**Note:**

* tf.constant() creates a constant tensor from a tensor-like object.
* The output returned by the result items are of type tensorflow.python.framework.ops.EagerTensor. So we shall convert that into NumPy array and find the mathematical statistics - like minimum value, mean value, etc - of each output.

**INSTRUCTIONS**

* Instantiate the class StyleContentModel and pass the style\_layers and content\_layers.
* extractor = << your code comes here >>(style\_layers, content\_layers)
* Now pass the tf.constant(content\_image) to the extractor and get the results - which holds the content and style representations of the content\_image.
* results = << your code comes here >>(tf.constant(content\_image))
* Let us see the statistics of the gram-matrices returned for the content image.
* for name, output in sorted(results['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()
* Let us see the statistics of the content matrices returned for the content image.
* for name, output in sorted(results['content'].items()):
* print(" ", name)
* print(" shape: ", output.numpy().shape)
* print(" min: ", output.numpy().min())
* print(" max: ", output.numpy().max())

print(" mean: ", output.numpy().mean())

Getting the Target Content and Style

With StyleContentModel which is a style and content extractor, we can now implement the style transfer algorithm. Do this by calculating the mean square error for your image's output relative to each target, then take the weighted sum of these losses.

Let us set the style and content target values. We shall do this by extracting the style of the style image and content of the content image.

**INSTRUCTIONS**

* Get the target content from content\_image using extractor as follows:
* content\_targets = extractor(content\_image)['content']
* Now similarly, get the target style from style\_image using extractor.

style\_targets = << your code comes here >>(style\_image)['style']

Defining some Hyper-parameters

Let us define the following hyper-parameters we would be using:

* style\_weight
* content\_weight
* an optimizer, here we shall use Adam and set its hyper-parameters values like the learning rate.

We shall also create a function clip\_0\_1 that would clip the values of image pixels to be in between 0 and 1 since this is a float image.

We shall also define the variable image which we would be using further to update its pixels throughout the train-steps in the coming slides. We shall assign the tf.Variable(content\_image) to the image. We use tf.Variable since the pixel values of this image are to be updated through the gradient descent.

**Note:**

* tf.clip\_by\_value clips tensor values to a specified min and max.

**INSTRUCTIONS**

* Define and. We do this to optimize using a weighted combination of the two losses to get the total loss:
* style\_weight=1e-2
* content\_weight=1e4
* Use the tf.optimizers.Adam optimizer and set the learning\_rate to 0.02, beta\_1 to 0.99, and epsilon to 1e-1.
* opt = << your code comes here >>(learning\_rate=0.02, beta\_1=0.99, epsilon=1e-1)
* Define the function clip\_0\_1:
* def clip\_0\_1(image):
* return tf.clip\_by\_value(image, clip\_value\_min=0.0, clip\_value\_max=1.0)
* Use tf.Variable to declare the image

image = << your code comes here >>(content\_image)

Defining Loss Function

* Let us now define the style loss and content loss for the input image. We would be using the style\_targets and content\_targets to do this.
* To do this, we shall define a function style\_content\_loss and implement the following steps:
  + Store the content representation and gram matrices of the style representations of the input image.
  + Calculate the mean squared difference between the gram matrices of the respective layers of the input image from the target representations. Add these average squared distances and scale this loss with style\_weight to obtain the style\_loss.
  + Calculate the squared difference between the content representations of the input image from the target representations. Add these average squared distances and scale this loss with content\_weight to obtain the content\_loss.
  + Add the style\_loss and content\_loss to obtain the total loss loss.

**Note:**

* tf.reduce\_mean computes the mean of elements across dimensions of a tensor.
* tf.add\_n adds all input tensors element-wise.

**INSTRUCTIONS**

* Use the following code:
* num\_content\_layers = len(content\_layers)
* num\_style\_layers = len(style\_layers)
* def style\_content\_loss(outputs):
* style\_outputs = outputs['style']
* content\_outputs = outputs['content']
* style\_loss = tf.add\_n([tf.reduce\_mean(
* (style\_outputs[name] - style\_targets[name])\*\*2)
* for name in style\_outputs.keys()] )
* style\_loss \*= style\_weight / num\_style\_layers
* content\_loss = tf.add\_n([tf.reduce\_mean(
* (content\_outputs[name]-content\_targets[name])\*\*2)
* for name in content\_outputs.keys()])
* content\_loss \*= content\_weight / num\_content\_layers
* loss = style\_loss + content\_loss

return loss

Converting Tensor to Image

* Let us define a function tensor\_to\_image to convert the input tensor to an image format.
* We do that as follows:
  + Make the pixel values from [0 , 1] to [0, 255].
  + Convert the pixels from float type to int type.
  + Get the first item(the image with 3 channels) if the tensor shape is greater than 3. In our exercise, the input tensor will be 4, where the first dimension is always 1. It is so because some of the functions we are using will be expecting the input tensors to be of size 4, for processing purposes.
  + Use PIL.Image.fromarray(tensor) to convert the tensor to image.

**INSTRUCTIONS**

* Use the below code:
* def tensor\_to\_image(tensor):
* tensor = tensor\*255
* tensor = np.array(tensor, dtype=np.uint8)
* if np.ndim(tensor)>3:
* assert tensor.shape[0] == 1
* tensor = tensor[0]

return PIL.Image.fromarray(tensor)

# Define the Training Step

Now it's the time to define the training step. We shall use tf.GradientTape to update the image.

Let us define train\_step(image) function which performs the calculation of gradient and updation of image pixel values for each train step epoch.

In defining the train\_step function, the following steps are implemented:

* Calculate the outputs which are the style and content representations of the input image, using the extractor which is the object of StyleContentModel. Then, call the function style\_content\_loss function to get the weighted-loss of the input image. Record all these operations using tf.GradientTape().
* Based on the thus obtained loss, calculate the gradients, using tape.gradient(loss, image).
* Then, apply these gradients using opt.apply\_gradients.
* Finally, update the image as per the gradients and clip the pixel values to be in 0-1 range.

**Note:**

* @tf.function converts a Python function to its graph representation for Faster execution, especially if the function consists of many small ops. The pattern to follow is to define the training step function, that's the most computationally intensive function, and decorate it with @tf.function.
* tf.GradientTape() records the list of the operations, so that these could be used for automatic differentiation during optimization. It is very highly recommended to go through [the official docs](https://www.tensorflow.org/guide/autodiff) in order to gain a bigger picture of this.
* optimizer.apply\_gradients applies the gradients.

**INSTRUCTIONS**

* Use the following code:
* @tf.function()
* def train\_step(image):
* with tf.GradientTape() as tape:
* outputs = extractor(image)
* loss = style\_content\_loss(outputs)
* grad = tape.gradient(loss, image)
* opt.apply\_gradients([(grad, image)])
* image.assign(clip\_0\_1(image))
* Now run a few steps to test:
* train\_step(image)
* train\_step(image)
* train\_step(image)
* tensor\_to\_image(image)

# Optimization

* Since it's working, perform a longer optimization.
* We shall implement the optimization as follows:
  + Let us go perform 5 epochs, each having 10 steps.
  + In each step, we will be calling the train\_step function which implements the actual optimization of the image pixels by calling various fore-defined functions for each iteration.
  + For each epoch, let us display the image with the updated pixels.
  + Let us also calculate the amount of time taken to perform this optimization. Let us use the time module.

**Note:**

* time.time() method returns the time as a floating-point number expressed in seconds since the epoch, in UTC.

We shall store the starting time in using time.time() in the beginning of the optimization code, and end=time.time(). Then, end-start gives the elapsed seconds taken to perform the optimization.

* We imported IPython.display as display in the beginning of the exercise. Now,
  + display.clear\_output() clears the output of the current cell receiving the output. It has a boolean parameter wait. If it is turned True, it means to wait to clear the output until the new output is available to replace it.
  + display.display() displays a Python object in all frontends.

**INSTRUCTIONS**

* Use the below code for optimization:
* import time
* start = time.time()
* epochs = 5
* steps\_per\_epoch = 10
* step = 0
* for n in range(epochs):
* for m in range(steps\_per\_epoch):
* step += 1
* train\_step(image)
* print(".", end='')
* display.clear\_output(wait=True)
* display.display(tensor\_to\_image(image))
* print("Train step: {}".format(step))
* end = time.time()
* print("Total time: {:.1f}".format(end-start))
* Due to the time and resource constraints, we have selected the specified number of epochs and steps.

Let us view the output image after training for 1000 iterations. Use the code below for the same:

stylized\_image = load\_img("/cxldata/dlcourse/output-stylized-image.png")

plt.figure(figsize=(6,4))

plt.axis('off')

imshow(stylized\_image, 'Output Stylized Image')

We could observe the style of the painting to be imparted into the content image, thus making us feel as if the dog image was drawn using the style of the style image. Hurray! You have now become a digital artist who could intelligently use Deep Learning to draw images of various styles. Congrats!