19AIE205 Python for Machine Learning

Project Report – 2020

Submission Date: December 5, 2020

(Neural Style Transfer)

Submitted By:

Sai Siddharth Cilamkoti

(AM.EN.U4AIE19055)

Neural Style Transfer

Problem Definition/ Abstract

Neural Style Transfer is an optimization technique that takes three images they are, content image, style image and the image that we want to style, blend them together such that the image output should look like the content image but painted in the style of the style image. This is the application of deep neural networks and deep learning in the field of art.

Datasets

There exists no specific dataset to complete style transfer but, the images that we choose should have a base content image like any normal photograph and the style image has to be a one such that it has an artistic effect and features.
The images that I have used are:

Content Image

Style Image

Prepare Data

To obtain the best possible results of the algorithm, I have used the architecture of the VGG19 Convolutional Neural Network. So, I had to preprocess both the content image and style image in such a way that these images can be run through the VGG19 network for optimization. So the images are normalized by mean = [103.939, 116.779, 123.68] with three channels i.e BGR.

To achieve this I used the command from tensorflow:
img = tf.keras.applications.vgg19.preprocess_input(img)
In order to view the final output image we also have to deprocess it to get the most visual effect out of it and we also have to restrict the pixel values in between 0 to 255. The function that I wrote to deprocess is as follows

```
def deprocess_img(processed_img):
  x = processed_img.copy()
  if len(x.shape) == 4:
    x = np.squeeze(x, 0)
  assert len(x.shape) == 3,
  if len(x.shape) != 3:
    raise ValueError("Invalid input to deprocessing image")

  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
```

Summarization

- Data Visualization:

  The snippets of code that I have used to load the images and display them are:

```
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)

  img = np.expand_dims(img, axis=0)
  return img
```

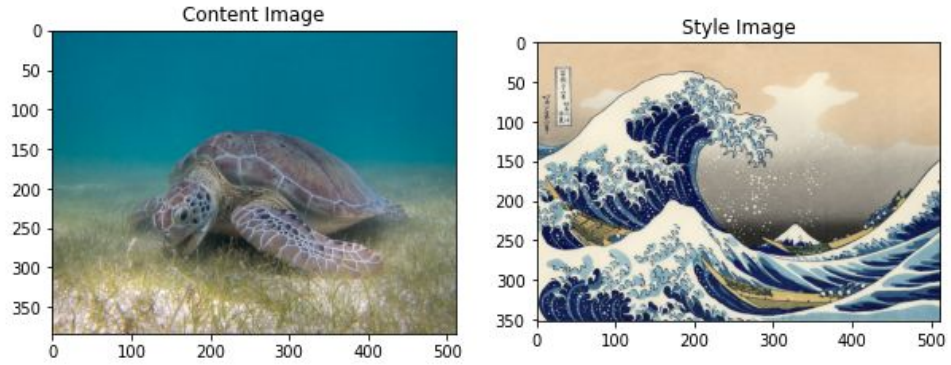
```
def imshow(img, title=None):
  out = np.squeeze(img, axis=0)
  out = out.astype('uint8')
  plt.imshow(out)
  if title is not None:
    plt.title(title)
  plt.imshow(out)
```

```
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()
```



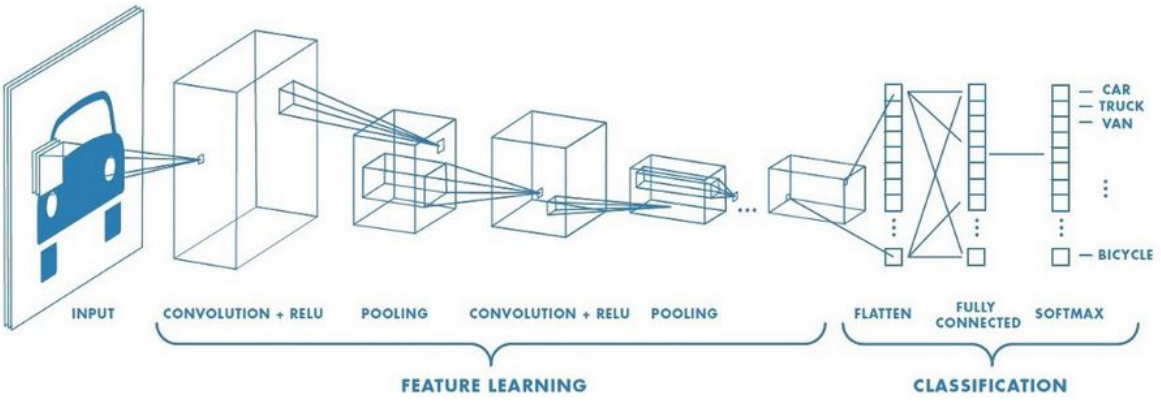
Python packages

The most important python package I have used is **Tensorflow**, this is a free library for machine learning and deep learning. Next, **Matplotlib** is the library I used to display and plot images according to the space constraint. **Numpy**, for tensor operations. I also used **Image** library from Pillow package which is used for manipulating images of different formats.

Explain the algorithm used

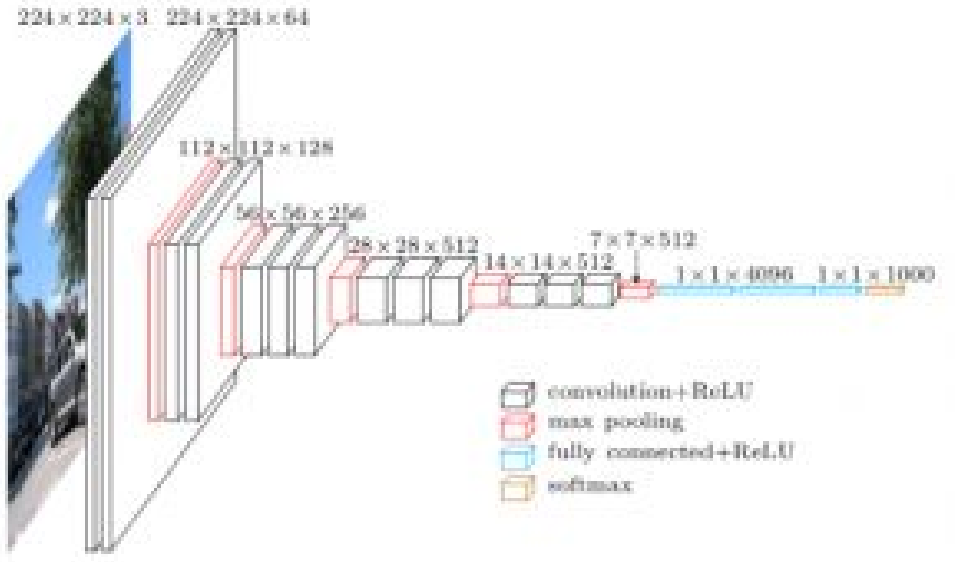
Style transfer uses a Convolutional Neural Network(CNN). It is an algorithm which can take in an input image, assign weights and biases to various aspects/objects, pattern in the image and be able to differentiate one from another.

The below image shows the flow of image, convolution operation and the final classification part of an image. A CNN has a filter matrix which is called a kernel, with which the convolution operation is performed on the image followed by pooling.



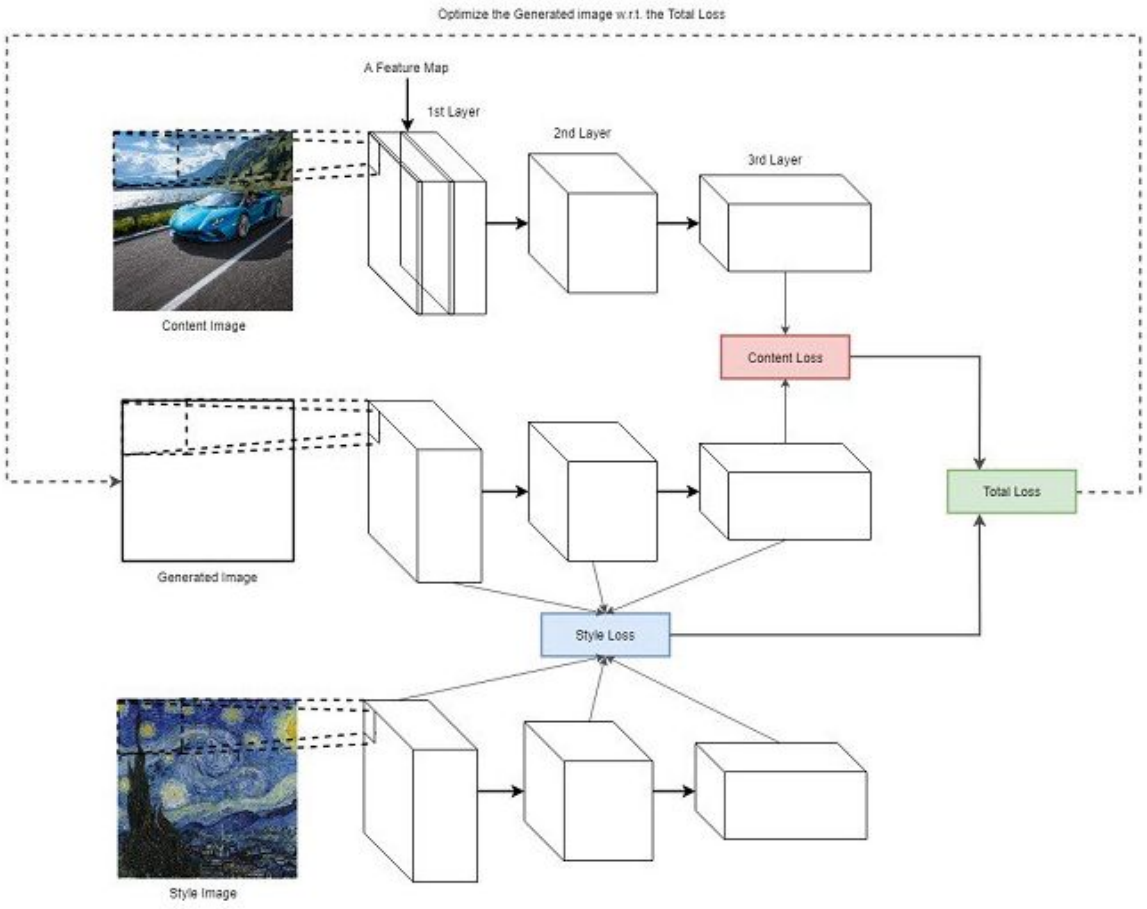
In a classical CNN training we will be training to update the weights of the neural network, but in **style transfer** the weights remain constant but the activations of the neurons in the neural network(i.e the image) are optimized and updated.

For the maximum optimized results, I chose to use a VGG-19 CNN architecture. VGG-19 architecture was proposed by Karen Simonyan and Andrew Zisserman of University of Oxford in 2014. The input to VGG based convNet is a 224*224 RGB image. Preprocessing layer takes the RGB image with pixel values in the range of 0–255 and subtracts the mean image values which is calculated over the entire ImageNet training set.



This is the pictorial architecture of VGG-19.

The high level architecture of a neural style transfer model is shown below



High level architecture of NST model

In style transfer, there are two cost functions, one style loss term and a content loss term.

The content loss function ensures that the activations of the higher layers are similar between the content image and the generated output image. Whereas, the style loss function makes sure that the correlation of activations in all the layers are similar between the style image and the generated image.

The expression for the content loss function is,

$$L_{content} = \frac{1}{2} \sum_{i,j} (A_{ij}^{l}(g) - A_{ij}^{l}(c))^2$$

Here, A^l {ij}(Image) is the activation of the ***l*** th layer,***i*** th feature map and ***j*** th position obtained using the Image.

This function captures the root mean squared error between the activations produced by the generated image and the content image.

This is coded as following,

```
def get_content_loss(base_content, target):
  return tf.reduce_mean(tf.square(base_content - target))
```

In defining the Style loss, we use the style information which is measured as the amount of correlation present between features maps in a given layer. a loss is defined as the difference of correlation present between the feature maps computed by the generated image and the style image. Mathematically, the style loss is defined as,

$$L_{style} = \sum_l w^l L^l_{style} \text{ where,}$$

$$L^l_{style} = \frac{1}{M^l} \sum_{ij} (G^l_{ij}(s) - G^l_{ij}g)^2 \text{ where,}$$

$$G^l_{ij}(I) = \sum_k A^l_{ik}(I) A^l_{jk}(I).$$

w^l (chosen uniform in this tutorial) is a weight given to each layer during loss computation and ***M^l*** is a hyperparameter that depends on the size of the ***l*** th layer. However in this implementation, we are not using ***M^l*** as that will be absorbed by another parameter when defining the final loss.

The code for this is given as follows,

```
def gram_matrix(input_tensor):
  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):
  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. * (channels ** 2) * (width * height) ** 2)
```

Here, the Gram matrix essentially captures the “distribution of features” of a set of feature maps in a given layer. By trying to minimise the style loss between two images, we are essentially matching the distribution of features between the two images.

The final loss function is defined as,

$$L = \alpha L_{content} + \beta L_{style}$$

where ***α*** and ***β*** are user-defined hyperparameters. Here ***β*** has absorbed the ***M^l*** normalisation factor defined earlier. By controlling ***α*** and ***β*** you can control the amount of content and style injected to the generated image.

The code for this is,

```
def compute_loss(model, loss_weights, init_image, gram_style_features, content_features):
  style_weight, content_weight = loss_weights
  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

  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)

  weight_per_content_layer = 1.0 / float(num_content_layers)
  for target_content, comb_content in zip(content_features, content_output_features):
    content_score += weight_per_content_layer* get_content_loss(comb_content[0], target_content)

  style_score *= style_weight
  content_score *= content_weight

  loss = style_score + content_score
  return loss, style_score, content_score
```

So our final aim is to reduce the cost function(**L**) by using **gradient descent** and optimize the output image.

Experimental result obtained

The output image is optimized for 1000 iterations, the effect of content image and style image is seen gradually over the iterations.

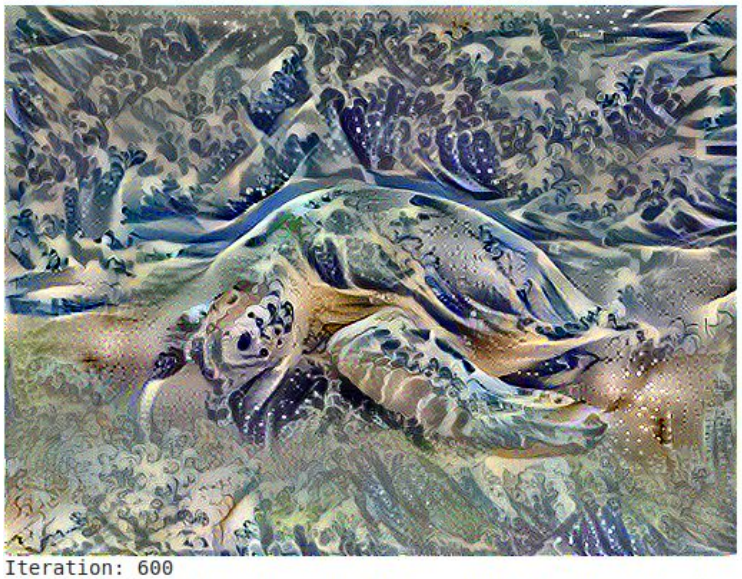
The output image at the end of 100th iteration looks is shown below,

```
Iteration: 100
Total loss: 1.8465e+07, style loss: 1.6870e+07, content loss: 1.5952e+06, time: 0.0578s
```

At the end of 300th iteration,

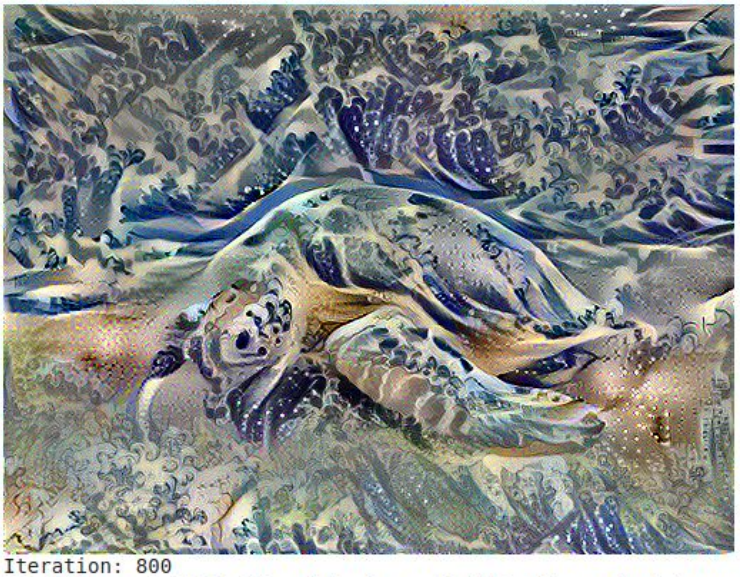
```
Iteration: 300
Total loss: 4.4306e+06, style loss: 3.2332e+06, content loss: 1.1974e+06, time: 0.0560s
```

At the end of 600th iteration,



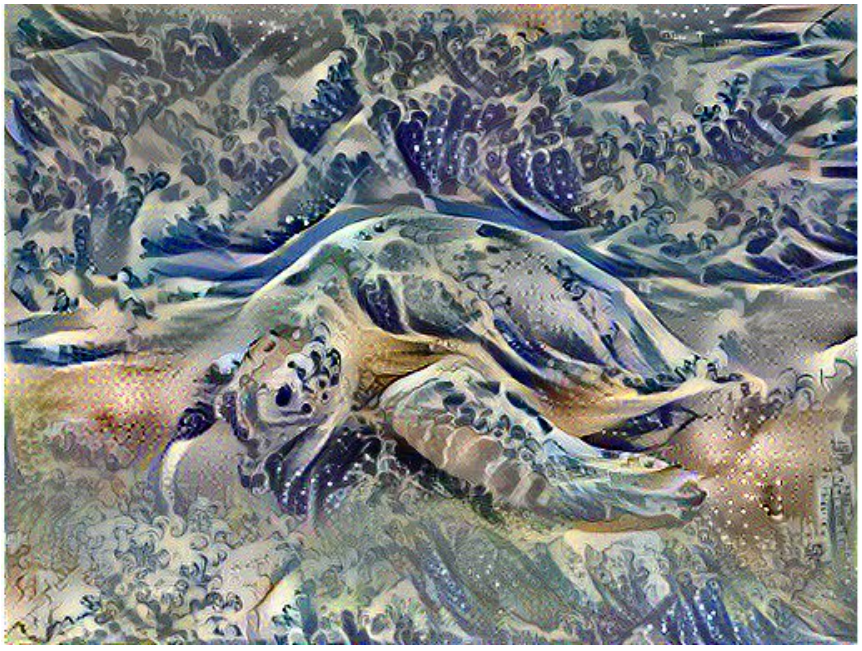
```
Iteration: 600
Total loss: 1.9661e+06, style loss: 1.1505e+06, content loss: 8.1568e+05, time: 0.0537s
```

At the end of 800th iteration,

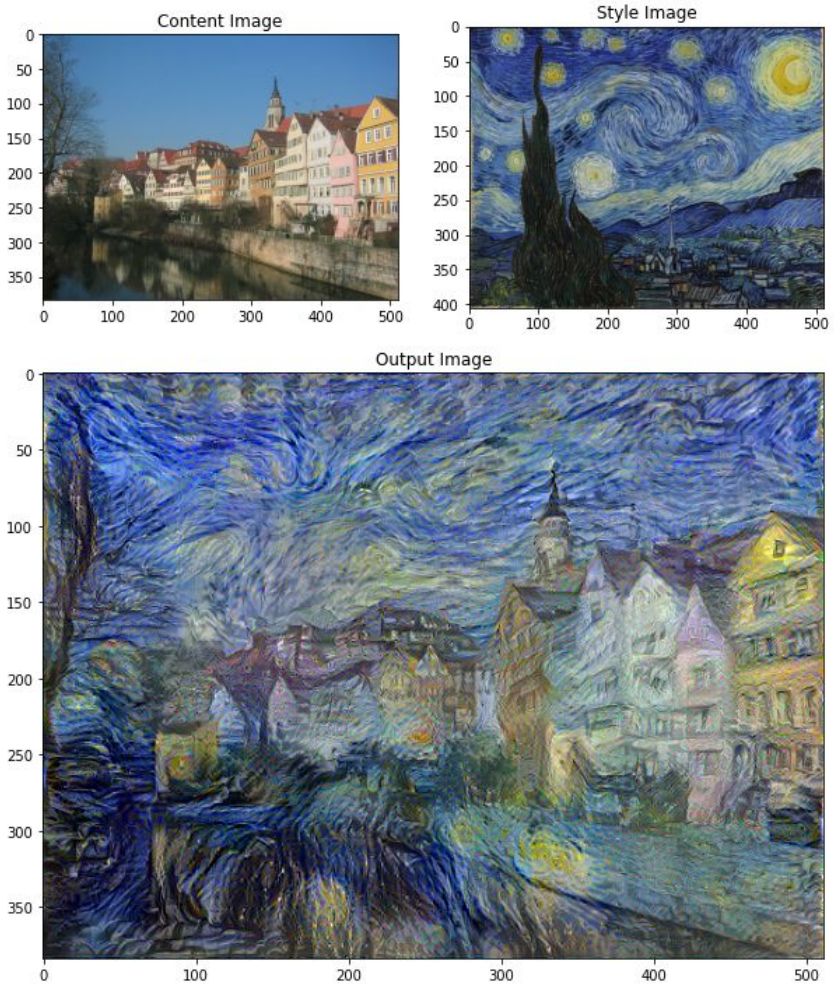


```
Iteration: 800
Total loss: 1.5486e+06, style loss: 8.4426e+05, content loss: 7.0438e+05, time: 0.0548s
```

And finally the styled output image is



Example 2:



Conclusion

The main aim of this project was to explore the application of machine learning and deep learning in any field possible. The art produced by these deep learning algorithms are at par with the art painted by many professionals. The major takeaway from this project for me was the application of CNNs.

Reference

https://towardsdatascience.com/light-on-math-machine-learning-intuitive-guide-to-neural-style-transfer-ef88e46697ee

https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Gatys_Image_Style_Transfer_CVPR_2016_paper.pdf