Implementation of paper A Neural Algorithm of Artistic Style
This paper is of great importance in the field of neural transfer, since it was the first one that described how you can separate content and style from an image with the same deep learning model. This is my interpretation and implementation of it.
The paper describes how, by getting the outputs of different feature maps alongside the deep neural network, we can get different representations of content and styles of an image. We then modify a white image by applying gradients via backpropagation to each pixel to reduce the distance between the white picture and the content and style representations of different pictures. The process to do this is described below.

Content image

White image

Style image

The neural style transfer architecture. I'll describe each part in detail

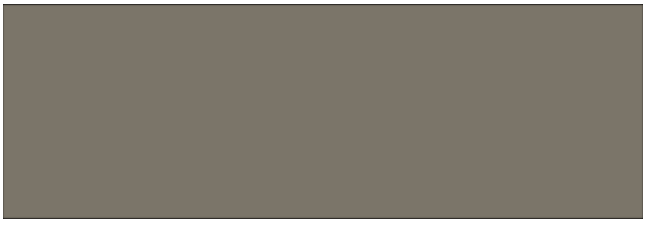
```
class neural_transfer_model(tf.keras.Model):
    def __init__(self,pre_trained_model,rows,cols,content_layers,style_layers,content_weight,style_weight
        super().__init__()

        self.rows = rows
        self.cols = cols

        self.pre_trained_model = pre_trained_model
        self.white_image = tf.Variable(tf.ones(shape = (1,rows,cols,3)))

        self.content_layer = self.pre_trained_model.get_layer(content_layers).output
        self.style_layer = [self.pre_trained_model.get_layer(layer).output for layer in style_layers]

        self.content_model = Model(inputs = self.pre_trained_model.inputs, outputs = self.content_layer)
        self.style_model = Model(inputs = self.pre_trained_model.inputs, outputs = self.style_layer)

        self.content_weight = content_weight
        self.style_weight = style_weight


    def compile(self, optimizer):
        super().compile()
        self.optimizer = optimizer

    def gram_matrix(self,x):  #Detailed in notebook style_Loss
        features = K.batch_flatten(K.permute_dimensions(x[0],(2,0,1)))
        gram = K.dot(features,K.transpose(features))
        return gram

    def style_loss(self,style,white_image):    #Detailed in notebook style_Loss
        style_gram_matrix = self.gram_matrix(style)
        white_image_gram_matrix = self.gram_matrix(white_image)
        size = self.rows * self.cols
        return tf.reduce_sum(tf.square(style_gram_matrix - white_image_gram_matrix)) / (36 * (size**2))
```

```
def content_loss(self,image):    #Detailed in notebook content_Loss
    base_pred = self.content_model(image)
    white_pred = self.content_model(self.white_image)
    return tf.reduce_sum(tf.square(white_pred - base_pred))


def train_step(self, data):

    content_data = data[0][0]
    style_data = data[0][1]

    with tf.GradientTape() as tape:

        s_loss = tf.zeros(shape=()) #for summing each layer loss

        style_fowards = self.style_model(style_data)
        style_white_noise_fowards = self.style_model(self.white_image)

        for layer_index in range(len(style_fowards)):
            current_layer_loss = self.style_loss(style_fowards[layer_index],style_white_noise_fowards[layer_index])
            #1/amount of layers * current layer loss to give same importance to each layer loss
            s_loss += (1 / len(style_fowards)) * current_layer_loss


        c_loss = self.content_loss(content_data)

        total_loss = self.content_weight * c_loss + self.style_weight * s_loss

    grads = tape.gradient(total_loss, [self.white_image])[0]
    self.optimizer.apply_gradients([(grads, self.white_image)])
    return {"total_loss": total_loss,"style_loss": s_loss,"content_loss": c_loss}
```

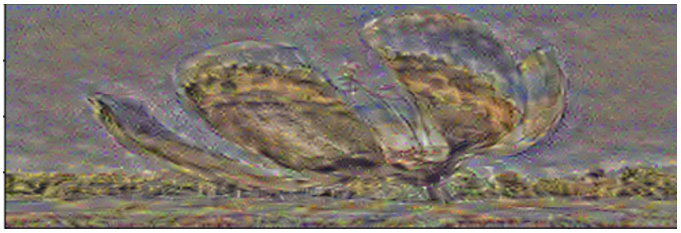
Neural style transfer model class

Content

The paper implies that the feature maps of higher layers in the network have the content representation of the image. It describes what and where things are, but not the exact pixel value as the layers in lower levels. Given a feature map of the content image and a feature map of the white image, we minimize the mean square error between the two matrices via backpropagation, propagating the gradients to each pixel in the white image.



Original white image



White image after training on content image

```
self.pre_trained_model = pre_trained_model
self.white_image = tf.Variable(tf.ones(shape = (1,rows,cols,3)))
```

We load the pretrained model (vgg16) and create the white image as a tensor variable when instantiating the class

```
self.content_model = Model(inputs = self.pre_trained_model.inputs, outputs = self.content_layer)
```

I used keras api function to create a graph with the vgg16 inputs and feature map (output) of the desired layer

$$\mathcal{L}_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} \left(F_{ij}^{l} - P_{ij}^{l} \right)^2 \ .$$

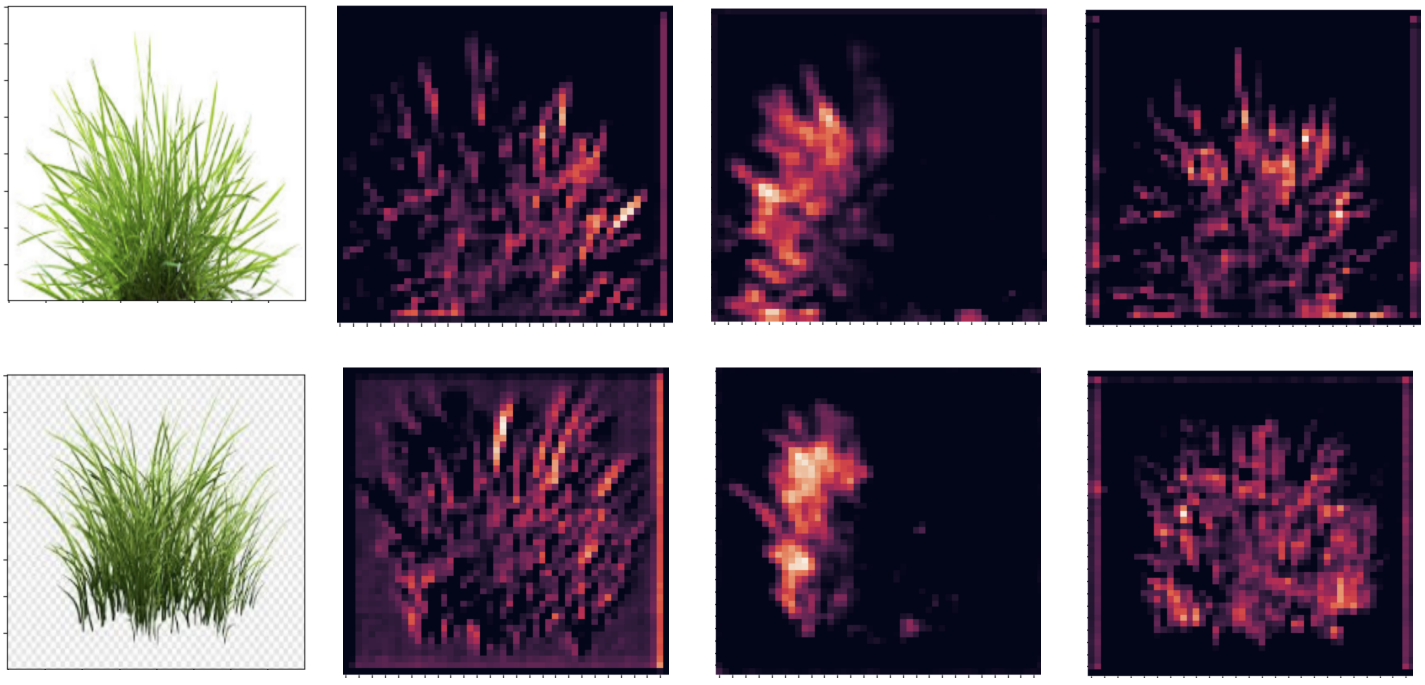
Content loss described in the paper

```
def content_loss(self,image):    #Detailed in notebook content_Loss
    base_pred = self.content_model(image)
    white_pred = self.content_model(self.white_image)
    return tf.reduce_sum(tf.square(white_pred - base_pred))
```

Implementation of content loss

Style

The style loss is very similar to the content. Instead of calculating the square mean error between feature maps, we calculate it between Gram Matrices. The concept of the gram matrix is that, having two pictures similar to each other, each feature map in a specific layer will “see” similar things. Using layer "block3_conv1" we get the following feature maps representation



Original pictures Feature map 1 Feature map 2 Feature map 3

We can see how the first three feature maps have similar representations of the images. By calculating the dot product of each feature map with the others, we make a gram matrix of n*n, where n is the number of feature maps in that specific layer. The diagonal is the dot product of each feature map with itself. By doing this we can observe that positions of high values in the gram matrix means that two feature maps dot product produce high value, therefore the two feature maps have high values. If we minimize the mean square error of the gram matrix of the style picture and the white image, the gram matrix of the white image picture will try to assimilate to the gram matrix of the style picture, therefore capturing the style of this one.

Gram Matrix

Gram matrix consist of calculating the dot product of each feature map with the rest, thus giving as a matrix of "importance" if you like. Dot product of bigger feature maps values will result in big values for that specific place of the matrix. The diagonal of the matrix is the dor product of each feature map with itself. Below is the standard code for calulating the gram matrix

```
feature_maps = tf.constant([[[1, 2, 3], [4, 5, 6],[7, 8, 9]],
           [[10, 20, 30], [40, 50, 60],[70, 80, 90]],[[100, 200, 300], [400, 500, 600],[700, 800, 900]]])
feature_maps #each dimension in axis 2 would be a feature map extracted from some layer in the pretrained model
```

```
<tf.Tensor: shape=(3, 3, 3), dtype=int32, numpy=
array([[[  1,   2,   3],
        [  4,   5,   6],
        [  7,   8,   9]],

       [[ 10,  20,  30],
        [ 40,  50,  60],
        [ 70,  80,  90]],

       [[100, 200, 300],
        [400, 500, 600],
        [700, 800, 900]]])>
```

```
# transpose columns to be rows, and each dimension in axis 2 is
#the concatenation of the first columns of each feature map
permuted_channels = tf.keras.backend.permute_dimensions(feature_maps, pattern=(2,0,1))

flatten = tf.keras.backend.batch_flatten(permuted_channels) #flatten each feature map into one vector

 #transpose the matrix formed by the transposed vector. (the amount of feature maps is the amount of rows)
transpose = tf.keras.backend.transpose(flatten)
```

```
permuted_channels
```

```
<tf.Tensor: shape=(3, 3, 3), dtype=int32, numpy=
array([[[  1,   4,   7],
        [ 10,  40,  70],
        [100, 400, 700]],

       [[  2,   5,   8],
        [ 20,  50,  80],
        [200, 500, 800]],

       [[  3,   6,   9],
        [ 30,  60,  90],
        [300, 600, 900]]])>
```

```
flatten
```

```
<tf.Tensor: shape=(3, 9), dtype=int32, numpy=
array([[  1,   4,   7,  10,  40,  70, 100, 400, 700],
       [  2,   5,   8,  20,  50,  80, 200, 500, 800],
       [  3,   6,   9,  30,  60,  90, 300, 600, 900]])>
```

```
transpose
```

```
<tf.Tensor: shape=(9, 3), dtype=int32, numpy=
array([[  1,   2,   3],
       [  4,   5,   6],
       [  7,   8,   9],
       [ 10,  20,  30],
       [ 40,  50,  60],
       [ 70,  80,  90],
       [100, 200, 300],
       [400, 500, 600],
       [700, 800, 900]])>
```

```
# Dot product of the flattened matrix of feature maps and the transpose of the matrix.
#With this we have the dot product of each feature map with itself and the other features maps
gram_matrix = tf.keras.backend.dot(flatten,transpose)
#each column in the transposed matrix and each row in the flattened matrix
```

```
# Each number is the dor product between the feature maps.
#909090 would be the dot product of the first feature map
#(the first dimension in axis 2 of the tensor) with the third feature map (third dimension in axis 2 of tensor)
gram_matrix
```

```
<tf.Tensor: shape=(3, 3), dtype=int32, numpy=
array([[ 666666,  787878,  909090],
       [ 787878,  939393, 1090908],
       [ 909090, 1090908, 1272726]])>
```

```
# dot product of feature map 1 with itself
1*1 + 4*4 + 7*7 + 10*10 + 40*40 + 70*70 + 100 * 100 + 400*400 + 700 * 700
```

```
666666
```

```
# dot product of feature map 3 with feature map 2
3*2 + 6*5 + 9*8 + 30*20 + 60*50 + 90*80 + 300 * 200 + 600*500 + 900 * 800
```

```
1090908
```

$$G_{ij}^{l} = \sum_{k} F_{ik}^{l} F_{jk}^{l}.$$

Style loss for each layer

```
def gram_matrix(self,x):  #Detailed in notebook style_Loss
    features = K.batch_flatten(K.permute_dimensions(x[0],(2,0,1)))
    gram = K.dot(features,K.transpose(features))
    return gram

def style_loss(self,style,white_image):    #Detailed in notebook style_Loss
    style_gram_matrix = self.gram_matrix(style)
    white_image_gram_matrix = self.gram_matrix(white_image)
    size = self.rows * self.cols
    return tf.reduce_sum(tf.square(style_gram_matrix - white_image_gram_matrix)) / (36 * (size**2))
```

Implementation of style loss for each layer

A^l and G^l their respective style representations in layer l. The contribution of that layer to the total loss is then

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^l - A_{ij}^l\right)^2 \tag{4}$$

and the total loss is

$$\mathcal{L}_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l \tag{5}$$

Total style loss weighted for each layer

```
s_loss = tf.zeros(shape=()) #for summing each layer loss

style_fowards = self.style_model(style_data)
style_white_noise_fowards = self.style_model(self.white_image)

for layer_index in range(len(style_fowards)):
    current_layer_loss = self.style_loss(style_fowards[layer_index],style_white_noise_fowards[layer_index])
    #1/amount of layers * current layer loss to give same importance to each layer loss
    s_loss += (1 / len(style_fowards)) * current_layer_loss
```

Implementation code for the total style loss

White image after a few epochs training with style loss

Content + Style loss:
If we sum up both the content loss and the style loss, we can make the white noise image become a combination of both pictures, with the content of the content style, and the gram matrix of the style picture, therefore with the style of it.
The paper uses the conv4_2 layer for the content loss, and conv1_1, conv2_1, conv3_1, conv4_1 and conv5 1 for style loss. Each layer of the style loss represents 1/amount_of_layers_used to the total style loss, so we multiply each style loss of each style layer by this amount. Also, the authors multiply each loss by a scalar to manipulate how much

each loss weighs to the final output. “The ratio α/β was either 1×10−3 (Fig 2 B,C,D) or 1 × 10−4 (Fig 2 E,F)”. I found a ratio of 1x10-2 to compose reasonable results.

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

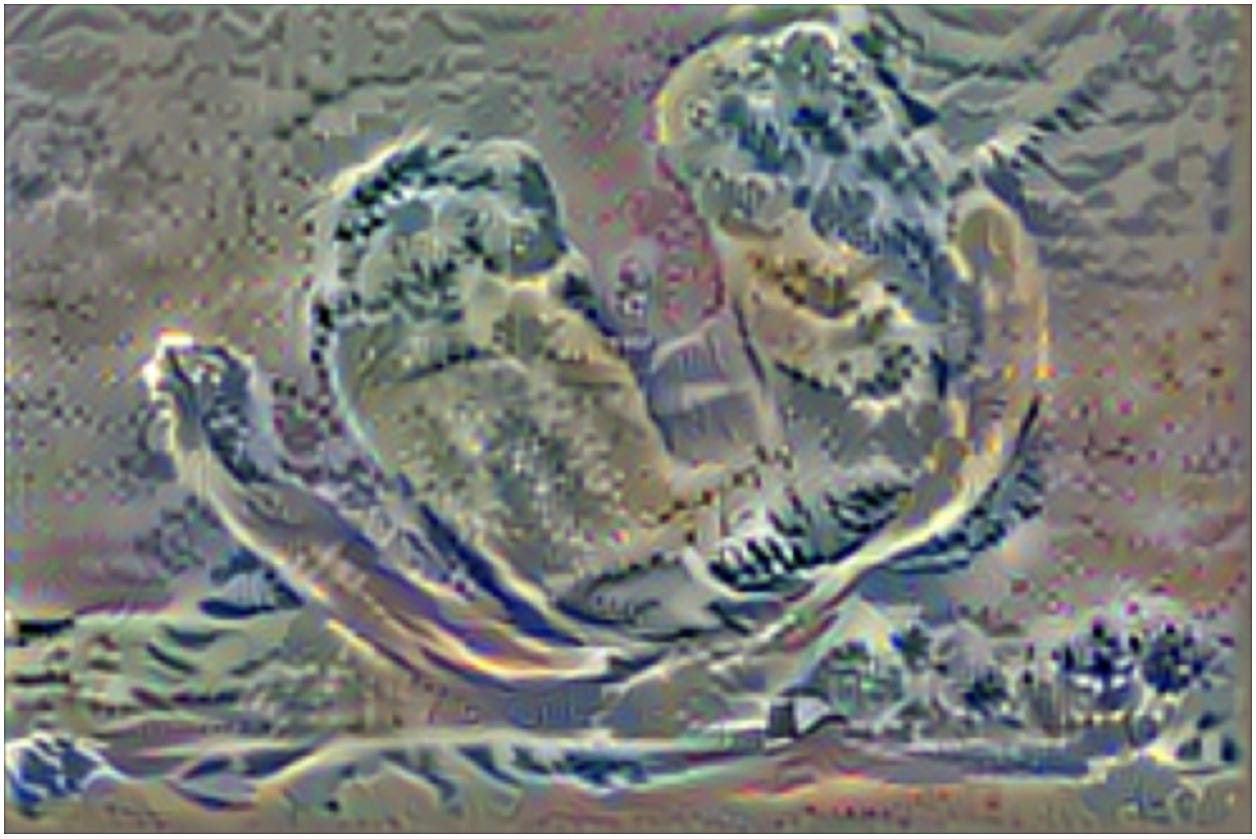
Total loss

```
  def train_step(self, data):

      content_data = data[0][0]
      style_data = data[0][1]

      with tf.GradientTape() as tape:

          s_loss = tf.zeros(shape=()) #for summing each layer loss

          style_fowards = self.style_model(style_data)
          style_white_noise_fowards = self.style_model(self.white_image)

          for layer_index in range(len(style_fowards)):
              current_layer_loss = self.style_loss(style_fowards[layer_index],style_white_noise_fowards[layer_index])
              #1/amount of layers * current layer loss to give same importance to each layer loss
              s_loss += (1 / len(style_fowards)) * current_layer_loss


          c_loss = self.content_loss(content_data)

          total_loss = self.content_weight * c_loss + self.style_weight * s_loss

      grads = tape.gradient(total_loss, [self.white_image])[0]
      self.optimizer.apply_gradients([(grads, self.white_image)])
      return {"total_loss": total_loss,"style_loss": s_loss,"content_loss": c_loss}
```

Complete training step with total loss

```
def get_image_tensor(path,rows,cols):
    """open an image and process it like vgg16 training set"""
    image = cv2.imread(path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    image = cv2.resize(image,(cols, rows))
    img_to_tensor = tf.convert_to_tensor(image, dtype=tf.float32)
    processed_image = tf.keras.applications.vgg16.preprocess_input(img_to_tensor)
    return processed_image



def deprocess_image(x,rows,cols):
    """convert a tensor processed by vgg19 into a valid image by applying the inverse of the preprocess function"""
    x = x.reshape((rows, cols, 3))
    # Remove zero-center by mean pixel
    x[:, :, 0] += 103.939
    x[:, :, 1] += 116.779
    x[:, :, 2] += 123.68
    # 'BGR'->'RGB'
    x = x[:, :, ::-1]
    x = np.clip(x, 0, 255).astype("uint8")
    return x
```

Utils functions to preprocess image into tensors and invert the preprocess images into valid images



Final result