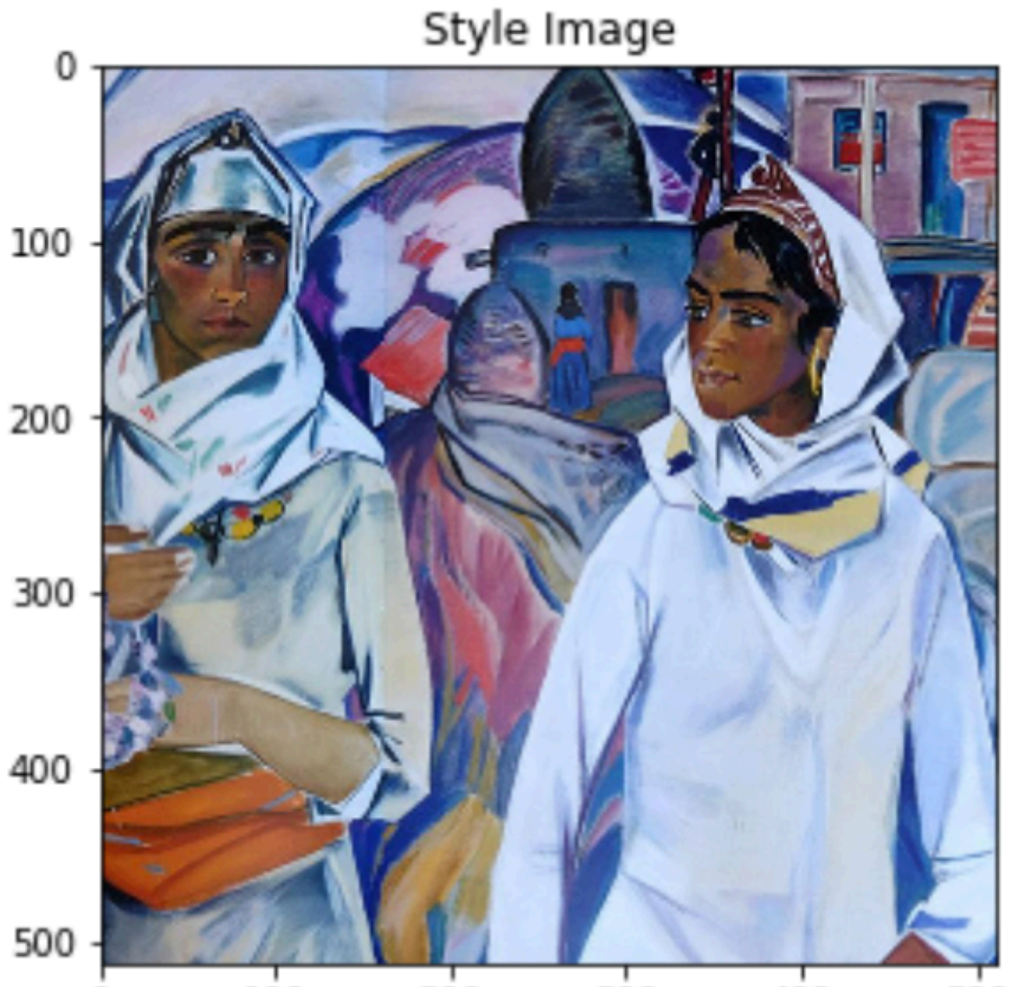
Neural Style Transfer

Recreating Nadir Abdurahmanov's style

As a trained painter, I loved looking at other painters' works. Still, I was not too fond of classes when we needed to recreate them as it was very tedious (both analyzing and implementing wise) and not very original. So, I always wanted superpowers to apply some style with a whim of a brush. Once I discovered the world of machine learning and generative adversarial networks, I realized I could finally become a wizard.

Hence, for my final project, I wanted to do something with generative art to get a taste of it as we didn't cover it in the main portion of the class. After some wandering around Google and seeing enough replications of Van Gogh or Picasso's styles, I decided it would be fun to do the same with one of my favourite Azerbaijani artists, Nadir Abdurahmanov. My favourite series of his work are paintings of Talysh women doing daily chores (me being Talysh may or may not play a role here). Some examples are:

So for the final project, I decided to learn how I can transfer style from one image to another to create some new artwork in Nadir's style! Hence, I will be implementing neural style transfer based on an excellent guide in [Keras (https://keras.io/examples/generative/neural_style_transfer/)](https://keras.io/examples/generative/neural_style_transfer/).



What is Neural Style Transfer?

Before diving into code, I have read over some papers (https://arxiv.org/abs/1508.06576) and medium artciles (https://towardsdatascience.com/light-on-math-machine-learning-intuitive-guide-to-neural-style-transfer-ef88e46697ee) describing what is Neural Transfer Network to understand better what is happening underneath the model.

Neural Style Transfer is an optimization technique used to blend two images, a content image and a style reference image. In the mixing process, the method makes sure that the input image looks like the content image but is made in the style of the style reference image. So, a style reference is put on top of the content image as an Instagram (or any other social media) filter on top of our faces.

How does this "transfer" work?

Neural style transfer uses two distance functions to work. The first one is to measure the difference in the content of the two images, content loss function $L_{content}$, while the second one measures the difference between the style of two images, L_{style}. Hence, this is an optimization problem where:

1) We are given content and style reference images;

2) We transform the input image by minimizing the content distance with the content image and its style distance with the style image.

So for an successful neural style transfer model, we would need to minimize both loss functions. We can use backpropagation to create a synthesized image that matches the content of the content image and the style of the style reference image.

Setup

In [1]:
```
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 pathlib
import IPython.display
import tensorflow as tf

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
```

Model

Based on the A Neural Algorithm of Artistic Style's paper (https://arxiv.org/abs/1508.06576), we can use the VGG19 model for neural style transfer. VGG19, similar to VGG16, is a pre-trained convolutional neural network model used for feature mapping in style transfer. Hence, we will use this CNN to extract the feature maps/kernels of content, style and finally generated images and freeze parameters during training. CNN is an appropriate model here as each layer learns a feature representation by detecting some patterns. Hence, CNN will learn to encode what both input images represent from top layers with simple general features to deeper and hidden layers with more complex features. So, as we have hierarchical feature extraction, we can select which layers will be used for content features and which will be used for layer features to ensure that generated image is in the shape of the content image only with some stylistic elements of style image.

Since we have two inputs (content image is used for both content and input) and would need to work with different layers of CNN for content and style, we will use Kera's Fucntional API (https://keras.io/guides/functional_api/) for determining the output activations.

Layer Selection for content and style features

As mentioned above, we don't need each layer for content and style as we want a blend, so we will select intermediate layers which would represent most of the content and some of the style images. Hence, later, we will match the style and content at those defined layers given an input image.

In [8]:
```
# Content layer
content_layer = ['block5_conv2']

# Style layers
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1'
               ]

num_cl = len(content_layer)
num_sl= len(style_layers)
```

In [9]:
```
def load_model():
    """
        Loads CNN model with access to its layers.

        Returns:
                returns a Keras model that takes image inputs and outputs the desired layers
                for style and content.
    """
    # Load pretrained on imagenet data VGG19 model
    vgg = tf.keras.applications.vgg19.VGG19(include_top=False, weights='imagenet')
    # We won't be training any layers for style transfer
    vgg.trainable = False
    # Extract layers for style and content
    style_outputs = [vgg.get_layer(name).output for name in style_layers]
    content_outputs = [vgg.get_layer(name).output for name in content_layer]
    model_outputs = style_outputs + content_outputs
    # Build model
    return models.Model(vgg.input, model_outputs)
```

Loss fucntions

Content Loss Function

The content loss function ensures that the activations of the higher layers, in our case block5_conv2, are similar between the content image and the generated image; in other words, the content in the content image is captured in the generated image. We have already selected a higher level, the top-most layer in CNN, which captures the content.

To calculate the loss function, we take the Euclidean distance between the higher-layer representations of those two images.

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

The content loss

Here, $A^l_{ij}(I)$ is the activation of the **l**th layer, **i**th feature mao and **j**th position obtained using the image **I**. So, $L_{content}$ is the root mean squared error between the activations produced by the content and generated images.

We will apply backpropagation to minimize the content loss by changing the input (to become generated) image until it has similar output in the defined higher layer as the content image.

In [10]:
```
def content_loss(base, genereted):
    """
        Calulate the loss function to maintain the "content" of the
        content image in the generated image
    """
    return tf.reduce_mean(tf.square(base - genereted))
```

Style Loss Function

From VGG19, we use each layer to extract the features for the Style Loss Function. Here, the style information is the amount of correlation between features maps in a given layer. While the loss is the difference of correlation between the features maps of generated and styles images. The mathematical functions (image is from here (https://towardsdatascience.com/light-on-math-machine-learning-intuitive-guide-to-neural-style-transfer-ef88e46697ee)):

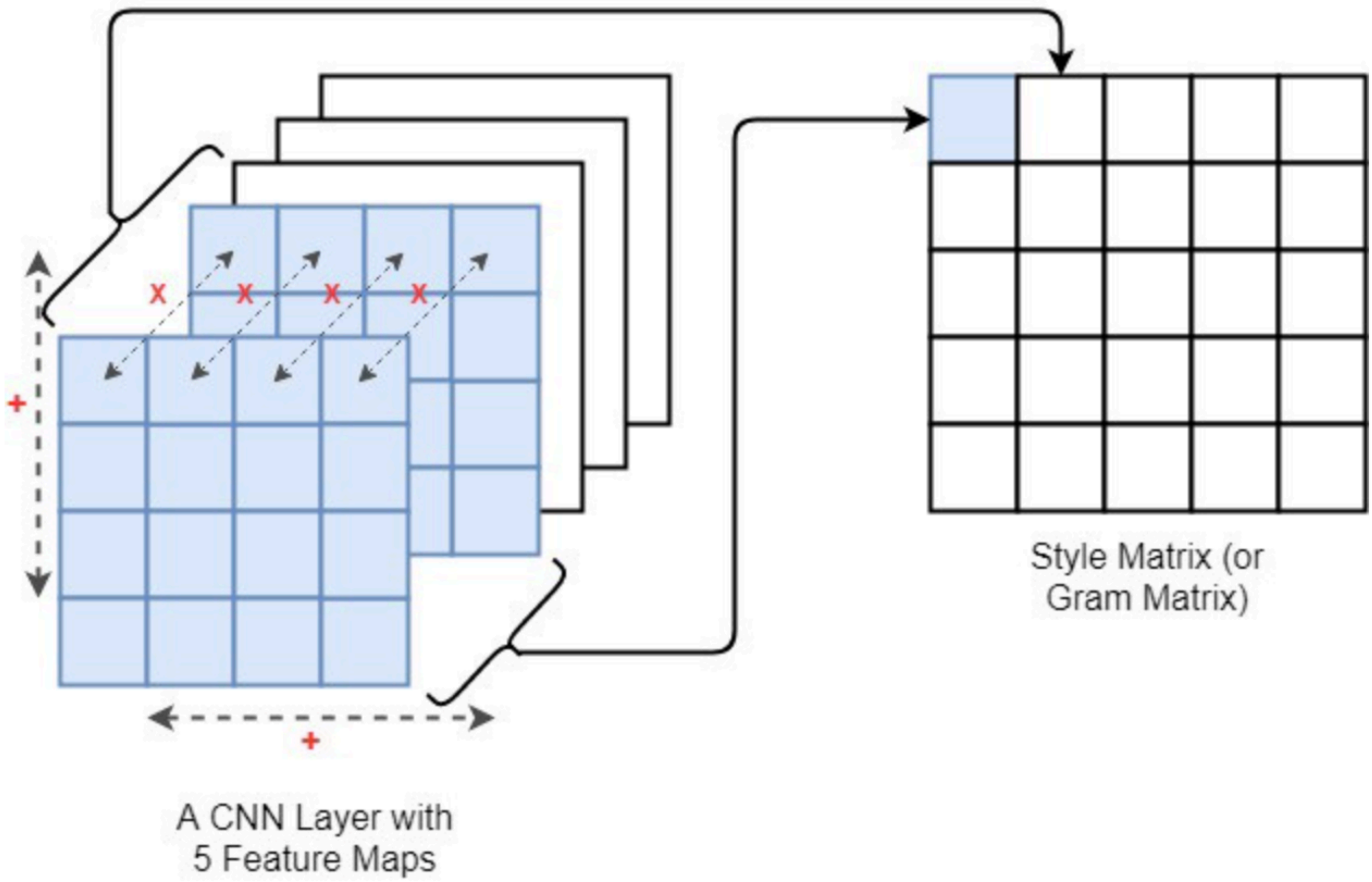
$$L_{style} = \sum_l w^l L^l_{style} \text{ where,}$$

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

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

Here, w^l is a weight given to each layer during loss computing, which we assume as uniform.

The goal of the style loss function is to compute a style or Gram matrix (image source (https://towardsdatascience.com/light-on-math-machine-learning-intuitive-guide-to-neural-style-transfer-ef88e46697ee)) for the generated and style image and find the distance between them. To create a style for our input image, we make a gradient descent from the content image to transform it into an image that matches the style representation of the original image. Hence, the style loss is computed as the root mean square difference between the two style matrices.



How the Style Matrix is Computed for a CNN Layer with 5 Feature Maps

The red cross shows element-wise multiplication in the matrices, and the red plus sign shows summing across both width heights of the feature maps.

In [11]:
```python
def gram_matrix(input_tensor):
    """
        This function computes gram matrix of an image tensors by
        performing feature-wise outer product
    """
    channels = int(input_tensor.shape[-1])

    features = tf.reshape(input_tensor, [-1, channels])
    n = tf.shape(features)[0]
    gram = tf.matmul(features, tf.transpose(features))
    return gram / tf.cast(n, tf.float32)

def style_loss(style, genereted_gram):
    """
        This function computes the style loss based on the gram matrices of
        feature maps from the style reference image and from the generated image
    """
    # Scale the loss at a given layer by the size of the feature map and the number of filters
    height, width, channels = style.get_shape().as_list()
    gram_style = gram_matrix(style)

    return tf.reduce_mean(tf.square(gram_style - genereted_gram))
```

Now, we need to load content and style image, feed them forward through the network, which will output the content and style feature representations from our model.

In [12]:
```
def get_features(model, content_path, style_path):
    """

        This function computes content and style features.

        Return:
             the style and content features.
    """
    # Process the content and style images
    content_image = process_img(content_path)
    style_image = process_img(style_path)

    # Compute iteretively content and style features
    style_outputs = model(style_image)
    content_outputs = model(content_image)


    # Extract the style and content features from the model
    style_features = [style_layer[0] for style_layer in style_outputs[:num_sl]]
    content_features = [content_layer[0] for content_layer in content_outputs[num_sl:]]

    return style_features, content_features
```

In [13]:
```
def compute_loss(model, loss_weights, base_image, gram_features, content_features):
    """

        This function computes the total loss.
        Return:
                total loss, style loss, content loss
    """

    style_weight, content_weight = loss_weights

    model_outputs = model(init_image)
    style_output_features = model_outputs[:num_sl]
    content_output_features = model_outputs[num_sl:]

    style_score = 0
    content_score = 0

    # Compute style losses from all layers
    weight_per_style_layer = 1.0 / float(num_sl)
    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)

    # Compute 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_output_features):
        content_score += weight_per_content_layer* get_content_loss(comb_content[0], target_content)

    style_score *= style_weight
    content_score *= content_weight

    # Add total variation loss
    loss = style_score + content_score
    return loss, style_score, content_score
```

In [14]:
```
def compute_loss_and_grads(cfg):
    """
       This function using tf.GradientTape computes the gradient.

       Return:
           gradient and total loss
    """
    with tf.GradientTape() as tape:
        all_loss = compute_loss(**cfg)
    total_loss = all_loss[0]
    return tape.gradient(total_loss, cfg['init_image']), all_loss
```

Training loop for transfer style model

I have chosen to use Adam's optimizer for our model to minimize our loss. Even though this is not generally the best performing optimizer on the style transfer, it is one of the quickest ones that serve this assignment's purpose. We iteratively update our generated image so that it minimizes our loss by training the input image to reduce loss. To achieve this, we calculate loss and gradients. We repeatedly run descent steps in the loop to minimize the loss and save the results after each epoch.

For the weights of content and style, a higher content/style ratio will yield an output image more representative of the original target image, while the opposite will yield an output image with stronger stylistic features. So, we set values for ratio on the higher end so our images preserve most of their intial shapes.

In [15]:
```
def style_transfer(content_path,
                   style_path,
                   num_iterations=1000,
                   content_weight=1e3,
                   style_weight=1e-2):


    model = get_model()
    for layer in model.layers:
        layer.trainable = False

    # Extract the style and content features
    style_features, content_features = get_features(model, content_path, style_path)
    gram_features = [gram_matrix(style_feature) for style_feature in style_features]

    # Set base image
    initial_image = process_img(content_path)
    initial_image = tf.Variable(initial_image, dtype=tf.float32)
    # Build our optimizer
    opt = tf.optimizers.Adam(learning_rate=5, beta_1=0.99, epsilon=1e-1)

    iter_count = 1

    # Store best optimized loss value and associated image
    best_loss, best_img = float('inf'), None

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

    # Params for displaying
    num_rows = 2
    num_cols = 5
    display_interval = num_iterations/(num_rows*num_cols)
    # Record running time
    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_loss_and_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()


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

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

            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, time.time() - start_time))


    print('Total time: {:.4f}s'.format(time.time() - global_start))

    # Plot intermediary outputs
    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
```

Trying out the style transfer!

Ex. 1

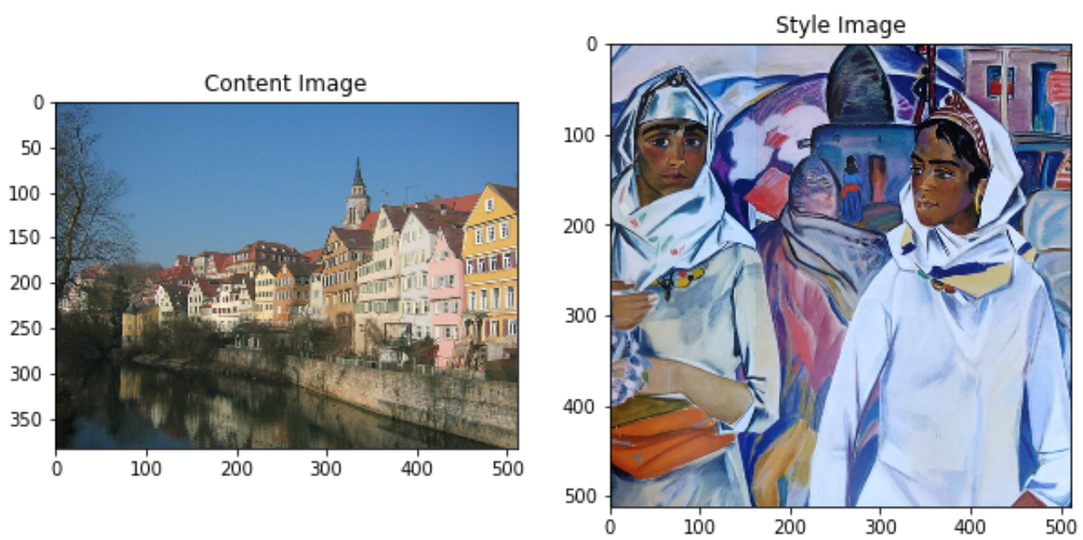
In [17]:
```
content_path = pathlib.Path('Tuebingen_Neckarfront.jpeg')
style_path = pathlib.Path('style_1.jpeg')
```

In [18]:
```
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()
```



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

```
Iteration: 900
Total loss: 9.0889e+05, style loss: 3.9538e+05, content loss: 5.1352e+05, time: 0.0514s
Total time: 3011.8428s
```

Ex. 2

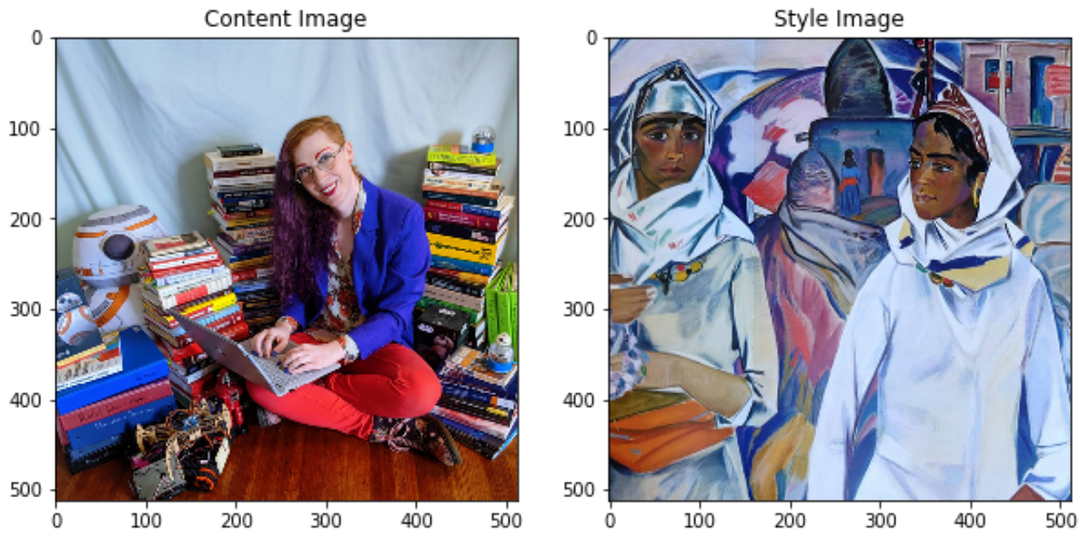
In [22]:
```
content_path_2 = pathlib.Path('content_2.png')
style_path_2 = pathlib.Path('style_1.jpeg')
```

In [23]:
```
plt.figure(figsize=(10,10))

content = load_img(content_path_2).astype('uint8')
style = load_img(style_path_2).astype('uint8')

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

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



In [24]:
```
best, best_loss = run_style_transfer(content_path_2,
                                     style_path_2,
                                     num_iterations=1000)
```

```
Iteration: 900
Total loss: 8.8154e+05, style loss: 3.8266e+05, content loss: 4.9888e+05, time: 0.0622s
Total time: 4060.3526s
```

I like this output! You can notice that even eyebrow color has been modified to fit the style of the reference image.

Ex. 3

In [26]:
```
content_path_2 = pathlib.Path('content_2.png')
style_path_3 = pathlib.Path('style_3.png')
```

In [27]:
```
plt.figure(figsize=(10,10))

content = load_img(content_path_2).astype('uint8')
style = load_img(style_path_3).astype('uint8')

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

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

In [28]:
```
best, best_loss = run_style_transfer(content_path_2,
                                     style_path_3,
                                     num_iterations=1000)
```

```
Iteration: 900
Total loss: 7.9256e+05, style loss: 2.4731e+05, content loss: 5.4525e+05, time: 0.0659s
Total time: 4230.2761s
```

As we can see from our results, we have achieved style transfer but at the same time some of the features such as eyes were reformed in the generated images (hence, we have higher loss for content than style). Both content and style loss are pretty low which would mean more epochs would only obscure further the generated images even if we ge lower loses. This was a greeat start to the NST and results turned better hat I thought!

References

Gatys, L., Ecker, A., & Bethge, M. (2015). A Neural Algorithm of Artistic Style. Retrieved 22 April 2022, from https://arxiv.org/abs/1508.06576 (https://arxiv.org/abs/1508.06576)

Intuitive Guide to Neural Style Transfer. (2020). Retrieved 22 April 2022, from https://towardsdatascience.com/light-on-math-machine-learning-intuitive-guide-to-neural-style-transfer-ef88e46697ee (https://towardsdatascience.com/light-on-math-machine-learning-intuitive-guide-to-neural-style-transfer-ef88e46697ee)

Neural Style Transfer Tutorial -Part 1. (2019). Retrieved 22 April 2022, from https://towardsdatascience.com/neural-style-transfer-tutorial-part-1-f5cd3315fa7f (https://towardsdatascience.com/neural-style-transfer-tutorial-part-1-f5cd3315fa7f)

Team, K. (2022). Keras documentation: Neural style transfer. Retrieved 22 April 2022, from https://keras.io/examples/generative/neural_style_transfer/ (https://keras.io/examples/generative/neural_style_transfer/)

Team, K. (2022). Keras documentation: The Functional API. Retrieved 22 April 2022, from https://keras.io/guides/functional_api/ (https://keras.io/guides/functional_api/)

Understanding the VGG19 Architecture. (2020). Retrieved 22 April 2022, from https://iq.opengenus.org/vgg19-architecture/ (https://iq.opengenus.org/vgg19-architecture/)

Appendix

Data Processing

Helper functions to load and prep the images for the model feeding.

In [30]:
```
def load_img(path_to_img):
    # Open and resize images
    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
```

In [31]:
```
def imshow(img, title=None):
    # Remove the batch dimension
    output = np.squeeze(img, axis=0)
    # Normalize the images
    output = output.astype('uint8')
    plt.imshow(out)
    if title is not None:
        plt.title(title)
    plt.imshow(out)
```

In [32]:
```
def process_img(path_to_img):
    # Format pictures into appropriate tensors
    img = load_img(path_to_img)
    img = tf.keras.applications.vgg19.preprocess_input(img)
    return img
```

In [33]:
```
def deprocess_img(processed_img):

    # Convert a tensor into a valid image
    x = processed_img.copy()
    if len(x.shape) == 4:
        x = np.squeeze(x, 0)

    # 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
```

In [ ]:
```
```