CS4043D Image Processing

Course Project

Date : 3rd April 2024

Neural Style Transfer

Group: E

Rithika Kathirvel	B200055CS
Pavanitha B	B200702CS
Aswin Sreekumar	B200737CS
Divya Krishnan	B200873CS

Department of Computer Science and Engineering
National Institute of Technology, Calicut

Declaration

This report has been prepared on the basis of our group's work. All the other published and unpublished source materials have been cited in this report.

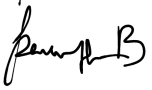
Student Name :	Rithika Kathirvel	Digital Signature:	
	Pavanitha B		
	Aswin Sreekumar		
	Divya Krishnan		

Table of Contents

Abstract

Neural image style transfer is an optimization technique that takes two images - a content image and a style reference image (such as an artwork by a famous painter) and blends them so the output image looks like the content image, but “painted” in the style of the style reference image. Neural style transfer is a growing topic of research. It is currently being used in diversified domains like gaming, virtual reality, photo and video editors, and art. We aim to implement 2 approaches: Gatys style transfer [1] and Fast style transfer [2] for neural image style transfer leveraging advanced image processing and machine learning techniques. With these implementations, we plan to thoroughly analyse and compare the efficiency and effectiveness of both approaches.

Project Specification

Neural Image Style transfer : Given two images namely, content image and style reference image,create a visually appealing image as output that retains the recognizable content of the original but incorporates the stylistic elements of a reference image. More formally,given a content image C and a style image S, the objective is to generate an image G that retains the content of C while adopting the visual style of S. It involves finding the optimal pixel values for G,using techniques like gradient descent, in such a way that a predefined loss function, which combines content and style losses, is minimised .The final output is an optimised image that preserves the content of the content image while adopting the style characteristics of the style image.

Mathematically, it involves finding the generated image G that minimises the loss function:

$$\mathbf{L_{Total}(G)=\alpha L_{content}(C,G)+\beta L_{style}(S,G)}$$ [5]

$L_{Content}(C,G)$ measures the content difference between C and the generated image G.
$L_{Style}(S,G)$ measures the style difference between S and the generated image G.
α and β are hyperparameters that control the relative importance of content and style in the generated image.

We explore 2 approaches : Gatys style transfer and fast style transfer. We also aim to provide insights into the strengths and limitations of each approach, enabling informed decision-making for selecting the most suitable method based on specific requirements and constraints.

Chapter 1: Introduction

1.1 Background

Neural style transfer has arisen as a potent tool within the domain of computer vision and image processing, facilitating the creation of images that fuse the content from one source with the artistic style of another. Conventional approaches to artistic style transfer typically depended on manually crafted algorithms and rules, which were constrained in their capacity to accurately replicate the complex details of artistic styles. Neural style transfer signifies a transformative shift in this field, utilising deep learning methodologies to comprehend and apply complex artistic styles from reference images onto diverse content images. It finds applications across various domains, including digital art creation, photo editing, graphic design, and even fashion and interior design.

Some existing approach for neural style transfer include :

1. Texture transfer: This technique is focused on transferring texture patterns from a style image to a content image. This involves methods like texture synthesis using Markov Random Fields (MRFs), which aim to generate new textures based on statistical properties extracted from the style image.

2. Filters and Image Processing techniques: Traditional image processing techniques such as convolutional filters, morphological operations and histogram matching were sometimes used for simple style transfer tasks. Less computational intensive making them more suitable for real-time applications.

3. Image Analogies: Image analogies aimed to transfer the style of one image onto another by finding correspondences between image patches and transferring their appearance. This approach relies on techniques like patch-based texture synthesis and matching.

4. Non-Photorealistic Rendering(NPR): It generates images that mimicked various artistic styles, such as painting or sketching. It uses rendering processes like stroke-based rendering or image abstraction. NPR allows real-time NST.

5. Other existing approaches : Other approaches include use of Pytorch, TensorHub and Magenta for quicker development of “fast style Transfer”. Although not as customisable as traditional NST approaches like the ones suggested above, the use of tensor hub and magenta allows real time style transfer from arbitrary style images.

1.2 Challenges

In both Gatys style transfer and Fast style transfer, computational resources can become a hindrance especially when handling high-resolution images or intricate styles. Striking a balance between retaining content and infusing style is a complex task, complicated by issues such as overfitting and

the loss of meaningful information. Moreover, the subjective nature of artistic style presents difficulties in objectively quantifying and assessing the quality of stylized outputs.

In fast style transfer balancing perceptual loss, style loss, and content loss to achieve both rapid execution and accurate stylization requires extensive experimentation and tuning. Training fast style transfer models typically involves large datasets of content and style images, which can be computationally intensive and time consuming to process and train on. Additionally, CNN can sometimes produce artifacts, such as blurriness, that can reduce the quality of the stylized image. In fast style transfer each style requires its own weights for the model which means it requires a lot of space to save weights for each type of style. In short, fast style transfer is one model per style. For different styles, different training should be done to obtain the appropriate weights which is very time consuming and resource intensive.

Furthermore in fast style transfer, the original implementation of this approach uses VGG-16. Use of VGG-16 as the CNN does not capture the more complex patterns and features in the content and style data. Hence the team used VGG-19 as it is a deeper convolution network. The additional layers in VGG19 allow it to capture more complex patterns and features in the data. However this comes with a challenge. Deeper networks typically require more computational resources and take longer to train. VGG19 is more computationally intensive compared to VGG16 due to its additional layers. Deeper networks have a higher risk of overfitting. In practice, this means that VGG19 might require more careful tuning and regularisation to prevent overfitting.

Chapter 2: Literature Review

2.1 A Neural Algorithm of Artistic Style by Leon A. Gatys, Alexander S. Ecker, Matthias Bethge [1]

The paper presents a novel approach,Neural Style Transfer, that combines Convolutional Neural Networks with optimization algorithms to generate artistic images that blend the content of one image with the style of another. The authors use VGG-Network(a high performance CNN) to extract content and style representations. Content is captured from the feature responses of higher layers of the network.On top of the filter responses of each layer ,a new feature space is built to capture the style of input image.Style representation captures correlation between different features in different layers of the CNN.During the image synthesis ,an optimisation algorithm is used to minimise the content loss and style loss. The key finding of this paper is that the representations of content and style in the Convolutional Neural Network are separable.The experimental results presented in the paper show the effectiveness of the approach.The superiority of this approach over the already existing methods is also highlighted in the paper.

Drawbacks in the approach of this paper is:

- Loss of Content Details: While the style is transferred effectively, some finer details of the content image might be lost or distorted in the process, especially when using strong style representations.
- Memory Requirements: The optimization process also requires storing intermediates representations of the style and content images , which can consume a large amount of money, particularly for high-resolution images.
- Artifacts and Distortions: The generated image may contain artifacts or distortions, especially around edges or in regions with intricate textures, potentially diminishing its visual quality.
- Computational Intensity: Implementing this algorithm can be computationally intensive for larger images and more complex styles.

2.2 Perceptual Losses For Real-Time Style Transfer And Super-Resolution by Justin Johnson, Alexandre Alahi, Li Fei-Fei [2]

In this paper, they have discussed the image transformation tasks. For solving this task, they have come across two approaches. One approach is to train a feed-forward convolution neural network using a per-pixel loss function to measure the difference between output and ground truth images. Second approach is to generate a high-quality image using perceptual loss functions. Here they have combined the benefits of both the approaches. They experimented on two tasks: Style transfer and Single-image super-resolution. Consequently, they were able to better reconstruct the fine features and sharp edges in single-image super-resolution and achieve comparable performance and faster speed in style transfer as compared to existing methods.

Drawbacks in the approach of this paper is:

- Loss of Fine Details: There may be loss of fine details in the content image, especially when applying strong stylistic effects. In some circumstances, this loss may be more noticeable due to the nature of the perceptual loss function used.

- Limited Style Variation and Generalization: Approach relies on fixed pre-trained styles, limiting customization without retraining. Also, certain styles may not generalise well to all content types, with potential limitations in applying styles to diverse images.
- Artifacts and Inconsistencies: The approach may yield artifacts or inconsistencies in images, like unnatural textures or colour shifts, especially in complex or stylistically intense areas.

2.3 Image Neural Style Transfer With Global And Local Optimization Fusion by Hui-Huang Zhao, Paul L Rosin, Yu Kun Lai, Mu-Gang Lin, Qin-Lin Liu [3]

In this paper, they have presented a new image synthesis method for image style transfer. This is achieved by combining both local and global style losses. The global style loss helps avoid patch transfer errors such as mouth, eyes and moustache being transferred into wrong places. The local style loss helps better retain detailed styles. Consequently, experimenting on various images showed that the proposed method can preserve the structure and colour of the content image while transferring styles and reducing artefacts.

Drawbacks in the approach of this paper is:

- Complexity: The approach involves a combination of global and local optimization techniques, which can make the implementation more complex.
- Loss of Content Details: Depending on the strength of the style representation and optimization parameters, the approach may result in a loss of fine content details in the generated images, particularly in regions where global and local optimizations conflict.
- Artifacts and Inconsistencies: The fusion of global and local optimization techniques may introduce new artifacts or inconsistencies in the generated images, especially in regions with complex textures or patterns.
- Limited Style Variation and Generalization: The approach's effectiveness may be limited by the styles used for training or the representation of those styles. Achieving highly customised or novel styles require retraining of the model which is challenging. Also, generalisation to all content images or styles, especially with limited training data diversity or quantity is challenging.

Chapter 3: Proposed Methodology

Gatys style Transfer Approach :

1. Decide content, style image and preprocess the images : Content and style images are decided. Convert the images into RGB format, resize them into maximum dimensions while maintaining aspect ratio,convert image to numpy array and create a batch of a single image since neural networks take the input in batches.

2. Feature extraction model and convolutional layers : We define a function that takes the pre-trained model (VGG) that will be used to extract features from the layer from which we want to extract features. Higher convolutional layers are used as a content layer because they learn complex and high-level features. For style layers, various layers at different scales to capture feature maps at different spatial scales.We utilise these maps to compute loss values. VGG model which outputs all feature maps from the layers defined when an image is passed through it. Using these we calculate the content loss and style loss.

3. Define the output image : Copy contents of the content image into it for faster convergence because the content is already present in the image and we get an appealing image in less number of optimization epochs.

4. Defining the content, style loss and loss function : Content Loss is the mean square error between two images. Similarly Style loss is a gram matrix to calculate correlation or similarity between feature maps of two images. The loss function is defined as the style and content loss of our style image and content image respectively and compares with the style and content loss of our target placeholder image which will be our final styled image which has the content of content image and style of style image.

5. Optimise the output image : Adam optimizer which uses gradient descent method is used for optimising the output image by decreasing loss value from loss function. The optimization of loss function(style + content loss) leads to highly pixelated and noisy image.To prevent this we introduce total variation loss to act as regularizer which smoothens generated image and ensures spatial continuity.

6. Final Output after iterative optimization : Define the number of epochs and steps per epochs and for every epoch, calculate loss and optimise our output image using Adam optimizer.The final image is displayed once required optimization is achieved.

Fast style Transfer Approach :

1. Creating Autoencoder : The network has an encoder-decoder architecture with residual layers. For creating autoencoders, we first define the required layers for Reflection Padding, Instance Normalisation, Conv Layer, Residual Layer, and Upsample Layer. With 3 X ConvLayer, 5 X Residual Layer, 3 X Upsample Layer, we create the autoencoder.

2. Dataset for Training : Define a style image which is fixed and a dataset like COCO which loads a lot of content images for training purposes. The COCO dataset is downloaded in zip format. For training the model , we create a tensorflow dataset which loads all images from the path specified and resize them to be of the same size , batching and prefetching.

3. Creating and Training the loss model : The goal is to learn the input-output mapping for various styled images so that a new content image is styled in real time. So for this we create and train a loss model. We are implementing Loss model using VGG-19. The original Implementation uses VGG-16. We are using vgg19 for improved speed and efficiency. We send a batch of input training images of various content images into an autoencoder which provides us output. The training process involves feeding batches of diverse content images into an autoencoder, which generates corresponding styled output images. The pretrained VGG-19 deep neural network extracts the features of the content and style images at different layers. These are used to compute content and style losses. The weighted average of the content loss and style loss is defined as perpetual loss. To minimise perpetual loss, we are using Adam optimizer with learning rate 1e-3 as suggested in the paper. We define a single training step. The training function calls the training step iterative as per the number of epochs defined.

4. Loading the model into autoencoder and styling images : Once we reached the desired level of training, We load the saved model checkpoint into the autoencoder for styling. Load a new content image that we want to style and in one forward pass of the model, we get a generated styled image. We can use that network for styling any image in one go without the need of iterative optimization.

Summarising the proposed methodology in the first approach, Decide content image and style image, from which we aim to generate an output image.We preprocess the images to convert it into RGB format, re-size it into maximum dimensions while maintaining aspect ratio and converting an image to numpy array and creating a batch of a single image since neural networks take the input in batches. Utilising a pre-trained model using which we obtain feature maps at various layers. We define a function that takes the pre-trained model (VGG) that will be used to extract features from images and the layer names from which we want to extract features. We use these maps to compute loss values. VGG model which outputs all feature maps from the layers defined when an image is passed through it. Using these we calculate the content loss and style loss. Content Loss is the mean square error between two images. It tells us how close pixels of two images are. Similarly Style loss is a gram matrix to calculate correlation or similarity between feature maps of two images. Using these loss values, we define a loss function which is employed to optimise our output image. Loss function combines the style and content loss of our style image and content image respectively and compares with the style and content loss of our target image. This target image after optimization will be our final output image of the content image in the style of style image. This is passed as input parameter to the loss optimizer function along with adam optimizer to decrease loss value. The optimization by loss function leads to highly pixelated and noisy images which is prevented by total variation loss. The loss optimizer function optimises the image using gradients method. The final image is displayed once required optimization is achieved.

In the second approach - fast style transfer, the primary focus is on training a feedforward network that can efficiently apply artistic styles to images. Model is an encoder-decoder architecture with residual layers. Input images are fed into the encoder, which extracts features, and then passed through the decoder to generate an output image of the same size as the input. We train the model using perceptual loss, which is calculated similarly to Gatys style transfer. We deploy a pre-trained

model (such as VGG19) to get feature maps from the style and content layers defined. Calculate style loss and content loss using these features. We use a dataset containing diverse images, such as the COCO dataset or other collections. These images should cover various contents like people, animals, vehicles, etc. Additionally, select a style image whose style you want the autoencoder to learn. This can be a painting or sketch. During training, feed batches of input images into the autoencoder. The output of the autoencoder serves as the styled image. Pass these output images through the loss model to extract features from different layers. Use these features to compute style loss and content loss. The weighted sum of these losses produces the perceptual loss, which guides the training process. Train the model iteratively, adjusting the weights of the encoder-decoder network to minimise the perceptual loss. This process continues until the model learns to generate stylized images effectively. After training, we can use the model for styling any number of images in the style of the style image used for training that model in one pass without the need of optimization for each image.

Once both the approaches are implemented, we analyse and compare the strengths and limitations of each approach and draw conclusions to make informed decisions for selecting the most suitable approach based on resources, requirements and other constraints.

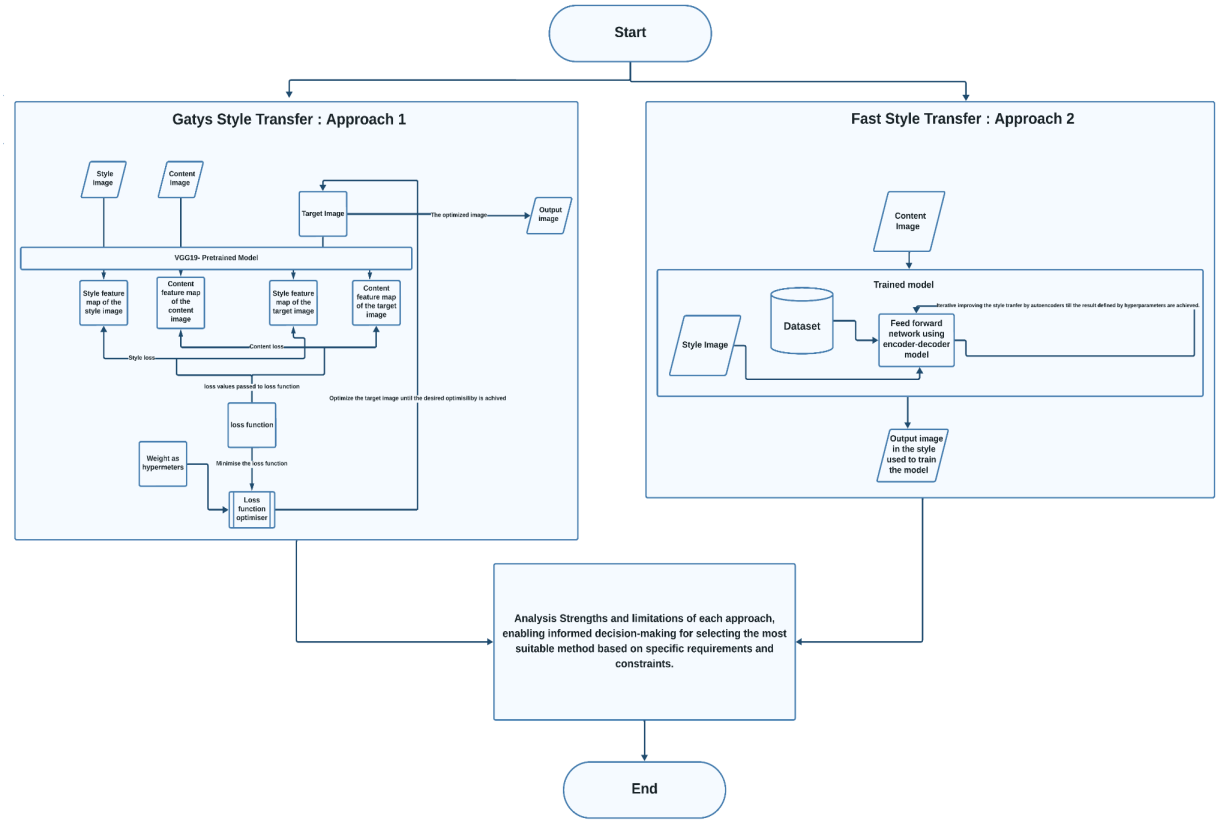


Figure 1: Graphical representation of the proposed solution.

3.1 Tools and Techniques

Gatys Style Transfer Approach:

1. System and GPU requirements : For styling images any system will work but size of output image is limited as per system memory. GPU is not a requirement for styling.

2. Language requirements : Python 3.6 or higher in Windows 10 or higher, or Linux.

3. Modules and Libraries : The necessary modules are Numpy, tensorflow, tensorflow.keras as high-level neural network library, pillow, and time module.

4. Feature Extraction : Here we make use of the VGG-19 model. VGG-19 is a deep Convolutional Neural Network(CNN) architecture with 19 convolutional layers.

4. Loss functions : There are 2 losses namely, content loss and style loss. Using MSE and gram matrices, we can find the closeness of features (style and texture) between two images.

5. Adam optimizer : Adam Optimiser helps in minimising the loss function. Adam uses Gradient Descent over iterations(few thousands) to update the random image matrix as close as both content and style image.

Fast Style Transfer Approach:

1. System and GPU requirements : Any system will work but size of output image is limited as per system. GPU is not a must for styling but for training GPU is must with tensorflow-gpu and cuda installed. We can use colab or kaggle kernels if the local machine doesn't have good computation capability.

2. Language of Implementation : Python 3.6 or higher in Windows 10 or higher, or Linux.

3. Modules and Libraries : NumPy (numpy): NumPy is a powerful library for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays, TensorFlow (tensorflow): TensorFlow is an open-source machine learning framework developed by Google. It provides support for building and training various machine learning models, including neural networks, through the use of computational graphs, TensorFlow.keras (tensorflow.keras): Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow. It provides an easy-to-use interface for building and training neural networks, making it a popular choice for deep learning practitioners, Pillow (PIL): Pillow is a Python Imaging Library (PIL) fork that adds support for opening, manipulating, and saving many different image file formats. It's commonly used for image processing tasks in Python, Matplotlib (matplotlib): Matplotlib is a comprehensive library for creating static, animated, and interactive visualisations in Python. It can be used to generate plots, histograms, power spectra, bar charts, error charts, scatterplots, etc, Requests (requests), Base64 (base64), IO (io): These modules are used for handling HTTP requests, encoding and decoding data using Base64, and

working with input/output operations, respectively. They are particularly useful for downloading and loading images from URLs, OS (os): The os module in Python provides a way of using operating system dependent functionality. It can be used to perform various operating system-related tasks such as file operations, directory operations, etc, Time (time): The time module provides various time-related functions in Python.

3. Feedforward network : Encoder-decoder architecture with residual layers.

4. Loss Model : For calculating style loss and content loss we implement a loss model using vgg19 as a pre-trained model. The original implementation uses vgg16. With weighted averaging of style and content losses, we define perceptual loss.

5. Adam Optimiser : Adam optimiser is used to minimise perpetual loss against the model's trainable parameters.

6. TensorFlow dataset : Tensorflow dataset loads all images from the path specified,and resize them to be of the same size for efficient batch training and implements batching and prefetching.

7. Training technique : Single step training function with enabled mixed-precision training as it offers significant computational speedup by performing operations in half-precision format.

8. Instance normalisation [4]: The autoencoders use instance normalisation instead of Batch Normalisation for better and robust results.

9. Configuring the dataset for efficiency : Preprocessing, and batching the dataset before training increases the efficiency of the model. We are training models with fixed-size images but we can generate images of any size because all layers in the model are convolutional layers.

Chapter 4: Implementation and Code

4.1 Documentation

In this section we have given the pseudocode and the description on programs, modules, user defined functions, main library functions used. Input and output of each user defined functions.

Gatys style Transfer Approach :

Step 1 : Import necessary modules and libraries

Libraries :

- numpy
- tensorflow
- tqdm.notebook
- PIL

Modules used : -

- tensorflow.keras.applications.vgg19
- tensorflow.keras.models
- time
- IPython.display

Step 2 : Define functions to load an image from path specified,denormalize the image,convert it to a numpy array and display the image.

- Function to load image
  Input: image path,max dimension(default:512)
  Output: preprocessed image array suitable for input into a neural network model.

```
Func load_image(image_path,max_dim=512):
    Open the image at the image_path
    Convert image to RGB
    Resize it with max_dim maintaining aspect ratio
    Convert image to numpy array
    Normalize pixel values
    Create a batch of single image
    Return image
```

- Function to process image
  Input: processed image
  Output: denormalized image,a numpy array with pixel values in the range [0, 255]

```
Func deprocess_image(img):
    Denormalize the image
    Return numpy array
```

- Function to convert array to image
  Input:a numpy array representing the image and a variable deprocessing set to false by default
  Output: image(if deprocessing is true it will first deprocess and then return the image)

```
Func array_to_img(array, deprocessing=False):
    If deprocessing:
        deprocess_image(array)
    If array_dim>3:
        Assert 1st dim represents batch
        Extract single batch
    Convert array to image and return image
```

- Function to display image
  Input: image as numpy array,deprocessing variable set to false by default
  Output: image displayed on screen

```
Func show_image(img, deprocessing=True):
    array_to_img(img, deprocessing)
    Display image
```

Step 3 : Load content and style image

```
content_image = load_image(content_img_path)
show_image(content_image)
style_image = load_image(style_img_path)
show_image(style_image)
```

Step 4 : Defining function to create a stylised model

- Function to create stylised model
  Input: pretrained neural network model,layer names from which we want to extract features
  Output: A new Model that takes the same input as the original model but outputs the activations of the specified layers

```
Func stylized_model(model, layer_names):
    Set trainable as False
    Get output of specified layers
    Create a new model with specified inputs and outputs
    Return new_model
```

Step 5 : Initiate a pretrained VGG and feature extraction layers.Initiate 2 models with content layers and style layers and store the feature map outputs to create the style transfer model.

```
vgg=vgg19.VGG19(weights='imagenet',include_top=False)
content_layers=['block5_conv2']
style_layers=['block1_conv1',
              'block2_conv1',
              'block3_conv1',
              'block4_conv1',
              'block5_conv1']
```

```
content_model = stylized_model(vgg, content_layers)
style_model = stylized_model(vgg, style_layers)
content_outputs = content_model(content_image)
style_outputs = style_model(style_image)
model = stylized_model(vgg, style_layers  + content_layers)
```

Step 6 : Obtain the output feature maps as a dictionary using the style transfer model and image as input.Extract feature maps from content and style image

- Function to obtain output feature maps as a dictionary
  Input:style transfer model,image
  Output: feature maps for content and style layers as dictionary

```
Func get_output_dict(model, inputs):
    Scale inputs to [0,255]
    Preprocess inputs using VGG-19 preprocess function
    Determine number of style layers
    Pass the preprocessed inputs through the model to get outputs
    Split the output into content and style components
    Create dictionaries for style and content outputs
    Return the output dictionaries
```

Step 7 : Define the style loss and content loss functions.Also define the loss function which merges the style and content loss

- Content_loss function:
  Input : placeholder(Activation tensor of a content layer for the generated image,content(Activation tensor of the same content layer for the original content image)
  Output : content loss(a scalar representing the content loss between the placeholder and content activations)

- Gram_matrix function:
  Input : Activation tensor of a layer
  Output : Gram matrix computed from the input tensor.

- Style_loss function:
  Input : placeholder(Activation tensor of a style layer for the generated image),style(Activation tensor of the same style layer for the style reference image)
  Output : style loss(a scalar representing the style loss between the placeholder and style activations)

```
Func content_loss(placeholder, content):
    return mean squared error between placeholder and content image
Func  gram_matrix(x):
    Compute Gram matrix using Einstein summation
    Return the normalized Gram matrix
Func style_loss(placeholder,style):
    s = gram_matrix(style)
    p = gram_matrix(placeholder)
    Compute the Mean square difference between s and p and return
```

- Function to define loss function

Input : outputs(dictionary containing the activations of the content and style layers for the generated image),content_outputs(dictionary containing the activations of the content layers for the original content image),style_outputs(dictionary containing the activations of the style layers for the style reference image),content_weight(weight of the content loss in the overall loss calculation),style_weight(weight of the style loss in the overall loss calculation)
Output : loss(computed loss value,which is a combination of content loss and style loss)

```
content_targets = get_output_dict[model,content_image]('content')
style_targets = get_output_dict[model,style_image]('style')
```

```
Func loss_function(outputs, content_outputs, style_outputs,
content_weight, style_weight):
   final_content = outputs['content']
   final_style = outputs['style']
   Compute content loss
   Compute style loss
   Combine content and style losses using weight parameters
   return total loss
```

Step 8 : Define output image(simply copy contents of the content image into it for faster convergence) and an optimizer.

```
output_image = tf.Variable(content_image, dtype=tf.float32)
optimizer=tf.optimizers.Adam(learning_rate=0.02, beta_1=0.99, epsilon=1e-1)
```

Step 9 : Define a loss optimizer function which includes total variation loss as a regulariser.We calculate gradients using tape gradient. With these gradients, we optimise our image using the optimizer.apply_gradients method.

- Loss optimizer function
  Input:image that we want to optimise,optimizer,content weight(weight assigned to the content loss term in the total loss),style weight(assigned to the style loss term in the total loss),total variation weight(weight assigned to the total variation loss term in the total loss)
  Output:loss(The computed loss value after applying the optimization step)

```
Func loss_optimizer(image, optimizer, content_weight,
style_weight,total_variation_weight):
   With GradientTape as tape:
      outputs = get_output_dict(model,image)
      loss = loss_function(outputs, content_targets, style_targets,content_weight,style_weight)
      Add total variation term to loss
   Compute gradients of loss w.r.t image
   Apply gradients to  image using optimizer
   Clip the pixel values in[0,1] range
   Return loss
```

Step 10 : Define the weights for each loss, number of epochs,steps per epoch,optimize the loss using adam optimizer and save the output image

```
total_variation_weight=0.0004
style_weight=1e-2
content_weight=1e4
epochs=10
steps_per_epoch=100
```

```
For each epoch:
  For each step in epoch:
      Optimize the loss function
Convert the output image array to image object
Save the final image
```

Fast style Transfer Approach :

Step 1 : The necessary modules are Numpy for arrays manipulation, tensorflow for tensor operations, tensorflow.keras as high-level neural network library for tensorflow for creating neural networks, pillow for converting an image to numpy array and vice versa, and saving output image, time module for calculating the time of each iteration, matplotlib for displaying images and graphs in notebook. Request, base64, io for downloading and loading images from URL and os module. tensorflow.keras.applications.vgg19: This module provides the VGG19 model pre-trained on the ImageNet dataset. matplotlib.rcParams['figure.figsize'] = (12, 12) line sets the default size of figures (plots) created by Matplotlib to a width of 12 inches and a height of 12 inches. matplotlib.rcParams['axes.grid'] = False line disables the grid lines on the axes of plots/

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.applications import vgg19
from tensorflow.keras.models import load_model,Model
from PIL import Image
import time
import matplotlib.pyplot as plt
import matplotlib
import requests
import base64
import os
from pathlib import Path
from io import BytesIO
matplotlib.rcParams['figure.figsize'] = (12,12)
matplotlib.rcParams['axes.grid'] = False
```

Step 2 : Define Utility functions to load an image from path specified and convert it to numpy array, plot a single as well as batches of images in grid, and convert a numpy array to image.

For loading the image from the image path (url or local path) which is our input we open the image from the path using the image library in python and resize according to the dimensions and resize specified if any. We then convert it to RGB and return the numpy array of the image as output.

To plot a single as well as batches of images in a grid, we make use of the matplotlib library in python. To convert from numpy array [input] to image, we first check if the array is in uint8 format as all images are in unit8 format. We then extract the single array from single or batch arrays as array and convert it to an image using the image library in python and return the output.

```
Func load_image(image_path, dim=None, resize=None)
Open img from image_path
If dim
   If resize
      Resize image to specified dimensions
   Else
      Resized to fit within the specified dimensions maintaining the aspect ratio
Convert img to RGB
Return numPy array of img
```

```
Func array_to_img(numPy_Array_of_img)
Check if data in uint8 format
Check the dimension of numPy_Array_of_img
If dim>3
   Assert there is only one image in the batch and extract it.
Convert array into image
```

```
Func show_img(image)
Check dimension of image
If Dim > 3
   Squeeze the image along the first axis to remove the batch dimension
Plot the image using matplotlib's imshow() function
```

```
Func show_images_grid (images, nums_rows=1)
Calculate the number of images
If number_of_image >1
   Calculate the number of columns
   Create grids of subplots with the specified number of rows and columns.
   Flatten the axis
   Resize the figure
   Iterates over the images and plots them on the corresponding subplots.
Else
   Display the single image.
```

Fig 1,2,3,4 : Pseudocode of the utility functions.

Step 2: Define loss model. We use the vgg19 pre-trained model. Then we define the layers and define a class that has methods for accessing feature maps. Using this class, we define the loss model as below.

```
Class LossModel :
Def _init_ (self, pretained_model, content_layer, style_layer)
      Initializes an instance of the LossModel class with the help of the arguments.
      Construct a model using the member function of this class get_model()
Def get_model(self)
   Freezes the layers of the vgg19 model by setting trainable to False.
   Combine the content_layer and style_layer into one list "Layer_list"
   Extracts the outputs of the layers  present in "Layer_list"
   Creates a new model with the same input but with outputs from the selected layers only.
   Return new_model
Def get_activation(self,inputs)
    Scale input to 0-255 range
    Pass through the loss model (self.loss_model) to get the outputs of the selected layers.
    Split output into style and content part.
    Compute activation for style and content.
    Return dictionary containing style and content activation.
```

```
Func Loss_model
vgg=vgg19.VGG19(weights='imagenet',include_top=False)
content_layers=['block4_conv2']
style_layers=['block1_conv1',
        'block2_conv1',
        'block3_conv1',
        'block4_conv1',
        'Block5_conv1']
Loss_model = LossModel(vgg, content_layers, style_layers)
```

Fig 5,6 : Pseudocode of the Loss Model class and the function for defining Loss Model.

Step 3 : Define content_loss, style_loss and perpetual loss as follows. The content_loss function calculates the content loss between a placeholder tensor and a content tensor. It asserts that the shapes of the placeholder and content tensors are the same. The content loss is computed as the mean squared difference between the placeholder and content tensors. Further, the style_loss function computes the style loss between a placeholder tensor and a style tensor. It asserts that the shapes of the placeholder and style tensors are the same. The style loss is computed as the mean squared difference between the Gram matrices of the style and placeholder tensors. The Gram matrix is computed using the Einstein summation ('einsum') method, which effectively calculates the inner products between the feature maps in the tensor. The result is normalised by the total number of elements in the feature maps. Gram matrices are commonly used in style transfer to capture the style information of an image. Finally the preceptual_loss function computes the perceptual loss, which combines content and style losses. It takes as input the predicted activations for content and style layers, as well as the target activations for content and style layers. Content and style losses are calculated for each layer and summed up, weighted by their respective weights. The final perceptual loss is the sum of the weighted content and style losses.

```
Def content_loss :
  Assert placeholder tensor and content tensor has same shape
  Return the mean squared difference of the placeholder and content tensor.

Def style_loss :
  Assert placeholder tensor and styletensor has same shape
   Calculate gram square matrix of each
   Return the weighted mean square difference of the elements of both the matrices.
```

```
Def perpetual_loss :
Extracts the content and style activations from dictionary
Calculate contentloss by iterating over content activations for each layer using the content_loss()
Similarly compute styleloss
Scale both the losses by their respective weights
Sum them to get the total perpetual loss
Return perpetual_loss
```

Fig 7,8 : Pseudocode of the calculating content loss, Style loss and perpetual loss.

Step 4: Create an autoencoder. Define the necessary layers for the network namely, Reflection padding is a technique used to pad images symmetrically by reflecting the values at the boundaries. InstanceNormalization which normalises each channel of the input independently, ConvLayer implements a convolutional layer followed by instance normalisation and reflection padding, ResidualLayer implements a residual block, which consists of two convolutional layers with the same number of filters and kernel sizes, followed by a ReLU activation and an addition operation with the input (residual connection) and UpsampleLayer which implements an upsampling layer, which increases the spatial dimensions of the input tensor using upsampling, followed by convolution, instance normalisation, and reflection padding. Using these networks create a convolutional autoencoder of the Architecture: 3 X ConvLayer, 5 X ResidualLayer, 3 X UpsampleLayer. Define a style model using the styleTransferModel class. This model can take an input image and transform it into a stylized output image by applying a series of convolutional, residual, and upsampling operations.

```
Class styleTransferModel(tf.keras.Model)
  Def __init__ :
    Initializes the StyleTransferModel class from tf.keras.Model and define layers.
  Def call :
    Define the forward pass of the model.
    Output is scaled to the range [0, 255] using a tanh activation function.
     Return Output
  Def print_shape (self, input) :
     print the shape of each layer's output.
```

```
Func style_model()
    style_model = StyleTransferModel()
```

Fig 9,10 : Pseudocode of the styleTransferModel class and function to define the style Model

Step 5: Configure dataset into tensorflow dataset training as below. TensorFlow dataset loads all images from the specified path, resizes them to a common size for efficient batch training, and implements batching and prefetching.

```
Class TensorflowDatasetloader:
Download COCO dataset for training images in zip format.
Unzip the dataset
Resize all the images in the dataset
Batch and prefetch the data for efficient performance.
```

```
Func configure_dataset(style_image_path)
Loader = TensorflowDatasetloader("COCO", batch_size=4)
Style_image = load_image(style_image_path)
Return dataset
```

Fig 11,12 : Pseudocode for configuring the dataset

Step 6: Train the model using adam optimiser. Define content weight, style weight and total variation weight these are hyperparameters which we can tune. At each epoch, we are calling the train_step function defined and after every epoch, We save the model checkpoint for further inferencing and training. The train_step function iterates over the dataset for the specified number of steps per epoch.
For each batch of input images: It applies the style model to generate stylized images, Clips the pixel values of the outputs between 0 and 255, Calculates the perceptual loss and total variation loss based on the stylized images, Computes gradients of the total loss with respect to the trainable variables of the style model using tf.GradientTape method. It then, Updates the weights of the style model using

the optimizer. It saves the model weights periodically based on a checkpoint frequency. Finally, it returns the average loss across all batches processed during the step.

Def train_step(dataset, style_activations,steps_per_epoch,style_model, Loss_model, path_model_checkpoints, content_weight, style_weight, adam_optimiser)
Iterates over the dataset for a specified number of steps.
Computes the stylized images using the style model.
Computes the perceptual loss between the predicted activations of stylized images and content activations of input images.
Adds total variation loss for smoothening.
Computes gradients and updates the model weights using the optimizer.
Save model checkpoints to the path defined every 1000 steps.
Returns the mean_batch_loss

Func train_model ()
content_weight=1e1
style_weight=1e2
total_variation_weight=0.004
Epochs = 2
num_of_Image = len(Loader)
Steps_per_epochs = num_image/batch_size
For epoch in range (1, epochs+1)
    Call the train_step which runs till number of steps per epochs defined
    After, every epoch save model checkpoint for further inferencing and training.

Fig 13,14 : Pseudocode of the train_step and function to train the Model

Step 7 : After we have satisfactorily trained the model, we load the saved model into the autoencoder as below. We pass the content image we want to style. The content image is styled. The output is displayed.

Func style_new_content_Image ()
Load saved model checkpoint from the path defined into autoencoder
Load content_image for styling
Convert content_image into float
Styled_output_image = style_model(test_image) //output generated in one forward pass.
Clamp the pixels of Style_output_image between 0 to 255 and convert it to uint8.
Display the styled image

Fig 15 : Pseudocode for styling any new content image.

4.2 Source Code

In this section we have given the source code of the user defined functions as code snippets for both the approaches namely Gatys style transfer and Fast style transfer.

Gatys style Transfer Approach :

1. Importing the modules

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.applications import vgg19
from tensorflow.keras.models import load_model, Model
from PIL import Image
import time
import IPython.display as display
from tqdm.notebook import tqdm
from pathlib import Path
```

2. Defining the content and style image

```
content_img_path="content.jpg"
style_img_path="style.jpg"
```

3. Defining basic functions

```
def load_image(image_path,max_dim=512):
    img = Image.open(image_path)
    img = img.convert("RGB")
    img.thumbnail([max_dim,max_dim])
    img = np.array(img,dtype=np.float32)
    img = img / 255.0
    img = np.expand_dims(img,axis=0)
    return img

def deprocess_image(img):
    img = 255*img
    return np.array(img, np.uint8)

def array_to_img(array, deprocessing=False):
    if deprocessing:
        array=deprocess_image(array)
    if np.ndim(array)>3:
        assert array.shape[0]==1
        array=array[0]
    return Image.fromarray(array)

def show_image(img, deprocessing=True):
    image=array_to_img(img, deprocessing)
    display.display(image)

content_image = load_image(content_img_path)
print(content_image.shape)
show_image(content_image)

style_image = load_image(style_img_path)
print(style_image.shape)
show_image(style_image)

def stylized_model(model, layer_names):
    model.trainable=False
    outputs=[model.get_layer(name).output for name in layer_names]
    new_model=Model(inputs=model.input,outputs=outputs)
    return new_model
```

4. Pretrained model

```
vgg=vgg19.VGG19(weights='imagenet',include_top=False)
vgg.summary()
```

5. Defining content and style layers from pretrained model

```
content_layers=['block5_conv2']

style_layers=['block1_conv1',
              'block2_conv1',
              'block3_conv1',
              'block4_conv1',
              'Block5_conv1']

content_model = stylized_model(vgg, content_layers)
style_model = stylized_model(vgg, style_layers)

content_outputs = content_model(content_image)
for layer_name, outputs in zip(content_layers, content_outputs):
    print(layer_name)
    print(outputs.shape)

style_outputs = style_model(style_image)
for layer_name,outputs in zip(style_layers, style_outputs):
    print(layer_name)
    print(outputs.shape)
```

6. Creating the model for style transfer

```
model = stylized_model(vgg, style_layers + content_layers)

def get_output_dict(model, inputs):
    inputs = inputs*255.0
    preprocessed_input = vgg19.preprocess_input(inputs)
    style_length = len(style_layers)
    outputs = model(preprocessed_input)
    style_output,content_output =
outputs[:style_length],outputs[style_length:]
    content_dict = {name:value for name,value in
zip(content_layers,content_output)}
    style_dict = {name:value for name,value in
zip(style_layers,style_output)}
    return {'content':content_dict,'style':style_dict}


results = get_output_dict(model, style_image)

print("Content Image output Feature maps: ")
for layer_name,output in sorted(results['content'].items()):
    print(layer_name)
    print(output.shape)

for layer_name,output in sorted(results['style'].items()):
    print(layer_name)
    print(output.shape)
```

```
output_dict = get_output_dict(model, content_image)
content_targets = output_dict['content']
output_dict = get_output_dict(model, style_image)
style_targets = output_dict['style']
```

7. Loss functions

```
def content_loss(placeholder, content):
    return tf.reduce_mean(tf.square(placeholder - content))

def gram_matrix(x):
    gram=tf.linalg.einsum('bijc,bijd->bcd', x, x)
    return gram/tf.cast(x.shape[1]*x.shape[2],tf.float32)
def style_loss(placeholder,style):
    s = gram_matrix(style)
    p = gram_matrix(placeholder)
    return tf.reduce_mean(tf.square(s-p))

def loss_function(outputs, content_outputs, style_outputs, content_weight,
style_weight):
    final_content = outputs['content']
    final_style = outputs['style']
    num_style_layers = len(style_layers)
    num_content_layers = len(content_layers)

    c_loss = tf.add_n([content_loss(content_outputs[name],
final_content[name]) for name in final_content.keys()])
    c_loss *= content_weight / num_content_layers
    s_loss = tf.add_n([style_loss(style_outputs[name], final_style[name]) for
name in final_style.keys()])
    s_loss*= style_weight / num_style_layers

    loss = c_loss + s_loss
    return loss

output_image = tf.Variable(content_image, dtype=tf.float32)
```

8. Defining the optimizer

```
optimizer=tf.optimizers.Adam(learning_rate=0.02, beta_1=0.99, epsilon=1e-1

def clip_0_1(image):
    return tf.clip_by_value(image,clip_value_min=0.0, clip_value_max=1.0)


def loss_optimizer(image, optimizer, content_weight, style_weight,
total_variation_weight):
    with tf.GradientTape() as tape:
        outputs = get_output_dict(model,image)
        loss = loss_function(outputs, content_targets, style_targets,
content_weight, style_weight)
```

```
        loss += total_variation_weight * tf.image.total_variation(image)
    grad = tape.gradient(loss, image)
    optimizer.apply_gradients([(grad, image)])
    image.assign(clip_0_1(image))
    return loss
```

9. Defining weights for each of the losses

```
total_variation_weight=0.0004
style_weight=1e-2
content_weight=1e4
```

10. Defining the number of epochs and steps per epoch

```
epochs=10
steps_per_epoch=100
```

11. Optimising the output image

```
start=time.time()
for i in range(epochs):
    print(f"Epoch: {i+1}")
    for j in tqdm(range(steps_per_epoch)):
        curr_loss = loss_optimizer(output_image, optimizer, content_weight,
style_weight, total_variation_weight)

    print(f"Loss: {curr_loss}")
end=time.time()
print(f"Image successfully generated in {end-start:.1f} sec")
```

12. Saving the output image

```
show_image(output_image.numpy(), deprocessing=True)
final_image = array_to_img(output_image.numpy(), deprocessing=True)
final_image.save("output.jpg")
```

Fast Style Transfer Approach :

1. Utility functions :

```
def load_image(image_path, dim=None, resize=False):
img = Image.open(image_path)
if dim:
    if resize:
        img = img.resize(dim)
    else:
        img.thumbnail(dim)
img = img.convert("RGB")
return np.array(img)
```

```
def load_url_image(url, dim=None, resize=False):
 img_request = requests.get(url)
 img = Image.open(BytesIO(img_request.content))
 if dim:
     if resize:
         img = img.resize(dim)
     else:
         img.thumbnail(dim)
 img = img.convert("RGB")
 return np.array(img)


def array_to_img(array):
 array = np.array(array, dtype=np.uint8)
 if np.ndim(array) > 3:
     assert array.shape[0] == 1
     array = array[0]
 return Image.fromarray(array)


def show_image(image, title=None):
 if len(image.shape) > 3:
     image = tf.squeeze(image, axis=0)
 plt.imshow(image)
 if title:
     plt.title = title


def plot_images_grid(images, num_rows=1):
 n = len(images)
 if n > 1:
     num_cols = np.ceil(n / num_rows)
     fig, axes = plt.subplots(ncols=int(num_cols), nrows=int(num_rows))
     axes = axes.flatten()
     fig.set_size_inches((15, 15))
     for i, image in enumerate(images):
         axes[i].imshow(image)
 else:
     plt.figure(figsize=(10, 10))
     plt.imshow(images[0])
```

2. Pretrained Model Instantiation and Defining Content and Style layers from pretrained model

```
vgg = vgg19.VGG19(weights="imagenet", include_top=False)
vgg.summary()

content_layers = ["block4_conv2"]
style_layers = [
  "block1_conv1",
  "block2_conv1",
```

```
  "block3_conv1",
  "block4_conv1",
  "block5_conv1",
]
content_layers_weights = [1]
style_layers_weights = [1] * 5
```

3. Creating Loss model, defining losses and using them to calculate perceptual loss

```
class LossModel:
            def __init__(self, pretrained_model, content_layers,
style_layers):
        self.model = pretrained_model
        self.content_layers = content_layers
        self.style_layers = style_layers
        self.loss_model = self.get_model()

    def get_model(self):
        self.model.trainable = False
        layer_names = self.style_layers + self.content_layers
            outputs = [self.model.get_layer(name).output for name in
layer_names]
        new_model = Model(inputs=self.model.input, outputs=outputs)
        return new_model

    def get_activations(self, inputs):
        inputs = inputs * 255.0
        style_length = len(self.style_layers)
        outputs = self.loss_model(vgg19.preprocess_input(inputs))
               style_output, content_output = outputs[:style_length],
outputs[style_length:]
        content_dict = {
               name: value for name, value in zip(self.content_layers,
content_output)
        }
        style_dict = {
                name: value for name, value in zip(self.style_layers,
style_output)
        }
        return {"content": content_dict, "style": style_dict}
class LossModel:
            def __init__(self, pretrained_model, content_layers,
style_layers):
        self.model = pretrained_model
        self.content_layers = content_layers
        self.style_layers = style_layers
        self.loss_model = self.get_model()

    def get_model(self):
        self.model.trainable = False
        layer_names = self.style_layers + self.content_layers
```

```
            outputs = [self.model.get_layer(name).output for name in
layer_names]
        new_model = Model(inputs=self.model.input, outputs=outputs)
        return new_model

    def get_activations(self, inputs):
        inputs = inputs * 255.0
        style_length = len(self.style_layers)
        outputs = self.loss_model(vgg19.preprocess_input(inputs))
              style_output, content_output = outputs[:style_length],
outputs[style_length:]
        content_dict = {
               name: value for name, value in zip(self.content_layers,
content_output)
        }
        style_dict = {
                name: value for name, value in zip(self.style_layers,
style_output)
        }
        return {"content": content_dict, "style": style_dict}



loss_model = LossModel(vgg, content_layers, style_layers)


def content_loss(placeholder, content, weight):
    assert placeholder.shape == content.shape
    return weight * tf.reduce_mean(tf.square(placeholder - content))
def gram_matrix(x):
    gram = tf.linalg.einsum("bijc,bijd->bcd", x, x)
       return gram / tf.cast(x.shape[1] * x.shape[2] * x.shape[3],
tf.float32)
def style_loss(placeholder, style, weight):
    assert placeholder.shape == style.shape
    s = gram_matrix(style)
    p = gram_matrix(placeholder)
    return weight * tf.reduce_mean(tf.square(s - p))
def preceptual_loss(
    predicted_activations,
    content_activations,
    style_activations,
    content_weight,
    style_weight,
    content_layers_weights,
    style_layer_weights,
):
    pred_content = predicted_activations["content"]
    pred_style = predicted_activations["style"]
    c_loss = tf.add_n(
        [
            content_loss(
```

```
                        pred_content[name], content_activations[name],
content_layers_weights[i]
            )
            for i, name in enumerate(pred_content.keys())
        ]
    )
    c_loss = c_loss * content_weight
    s_loss = tf.add_n(
        [
            style_loss(
                            pred_style[name], style_activations[name],
style_layer_weights[i]
            )
            for i, name in enumerate(pred_style.keys())
        ]
    )
    s_loss = s_loss * style_weight
    return c_loss + s_loss
```

4. Defining the Layers and using them we create an encoder-decoder Style model

```
class ReflectionPadding2D(tf.keras.layers.Layer):
    def __init__(self, padding=(1, 1), **kwargs):
        super(ReflectionPadding2D, self).__init__(**kwargs)
        self.padding = tuple(padding)

    def call(self, input_tensor):
        padding_width, padding_height = self.padding
        return tf.pad(
            input_tensor,
            [
                [0, 0],
                [padding_height, padding_height],
                [padding_width, padding_width],
                [0, 0],
            ],
            "REFLECT",
        )

class InstanceNormalization(tf.keras.layers.Layer):
    def __init__(self, **kwargs):
        super(InstanceNormalization, self).__init__(**kwargs)

    def call(self, inputs):
        batch, rows, cols, channels = [i for i in inputs.get_shape()]
        mu, var = tf.nn.moments(inputs, [1, 2], keepdims=True)
        shift = tf.Variable(tf.zeros([channels]))
        scale = tf.Variable(tf.ones([channels]))
        epsilon = 1e-3
        normalized = (inputs - mu) / tf.sqrt(var + epsilon)
        return scale * normalized + shift
```

```
class ConvLayer(tf.keras.layers.Layer):
    def __init__(self, filters, kernel_size, strides=1, **kwargs):
        super(ConvLayer, self).__init__(**kwargs)
              self.padding = ReflectionPadding2D([k // 2 for k in
kernel_size])
            self.conv2d = tf.keras.layers.Conv2D(filters, kernel_size,
strides)
        self.bn = InstanceNormalization()

    def call(self, inputs):
        x = self.padding(inputs)
        x = self.conv2d(x)
        x = self.bn(x)
        return x
class ResidualLayer(tf.keras.layers.Layer):
    def __init__(self, filters, kernel_size, **kwargs):
        super(ResidualLayer, self).__init__(**kwargs)
        self.conv2d_1 = ConvLayer(filters, kernel_size)
        self.conv2d_2 = ConvLayer(filters, kernel_size)
        self.relu = tf.keras.layers.ReLU()
        self.add = tf.keras.layers.Add()

    def call(self, inputs):
        residual = inputs
        x = self.conv2d_1(inputs)
        x = self.relu(x)
        x = self.conv2d_2(x)
        x = self.add([x, residual])
        return x
class UpsampleLayer(tf.keras.layers.Layer):
      def __init__(self, filters, kernel_size, strides=1, upsample=2,
**kwargs):
        super(UpsampleLayer, self).__init__(**kwargs)
        self.upsample = tf.keras.layers.UpSampling2D(size=upsample)
              self.padding = ReflectionPadding2D([k // 2 for k in
kernel_size])
            self.conv2d = tf.keras.layers.Conv2D(filters, kernel_size,
strides)
        self.bn = InstanceNormalization()

    def call(self, inputs):
        x = self.upsample(inputs)
        x = self.padding(x)
        x = self.conv2d(x)
        return self.bn(x)

class StyleTransferModel(tf.keras.Model):
    def __init__(self, **kwargs):
                                              super(StyleTransferModel,
self).__init__(name="StyleTransferModel", **kwargs)
        self.conv2d_1 = ConvLayer(
```

```
                            filters=32,  kernel_size=(9,  9),  strides=1,
name="conv2d_1_32"
        )
        self.conv2d_2 = ConvLayer(
                             filters=64,  kernel_size=(3,  3),  strides=2,
name="conv2d_2_64"
        )
        self.conv2d_3 = ConvLayer(
                            filters=128,  kernel_size=(3,  3),  strides=2,
name="conv2d_3_128"
        )
           self.res_1 = ResidualLayer(filters=128, kernel_size=(3, 3),
name="res_1_128")
           self.res_2 = ResidualLayer(filters=128, kernel_size=(3, 3),
name="res_2_128")
           self.res_3 = ResidualLayer(filters=128, kernel_size=(3, 3),
name="res_3_128")
           self.res_4 = ResidualLayer(filters=128, kernel_size=(3, 3),
name="res_4_128")
           self.res_5 = ResidualLayer(filters=128, kernel_size=(3, 3),
name="res_5_128")
        self.deconv2d_1 = UpsampleLayer(
            filters=64, kernel_size=(3, 3), name="deconv2d_1_64"
        )
        self.deconv2d_2 = UpsampleLayer(
            filters=32, kernel_size=(3, 3), name="deconv2d_2_32"
        )
        self.deconv2d_3 = ConvLayer(
                             filters=3,  kernel_size=(9,  9),  strides=1,
name="deconv2d_3_3"
        )
        self.relu = tf.keras.layers.ReLU()

    def call(self, inputs):
        x = self.conv2d_1(inputs)
        x = self.relu(x)
        x = self.conv2d_2(x)
        x = self.relu(x)
        x = self.conv2d_3(x)
        x = self.relu(x)
        x = self.res_1(x)
        x = self.res_2(x)
        x = self.res_3(x)
        x = self.res_4(x)
        x = self.res_5(x)
        x = self.deconv2d_1(x)
        x = self.relu(x)
        x = self.deconv2d_2(x)
        x = self.relu(x)
        x = self.deconv2d_3(x)
        x = (tf.nn.tanh(x) + 1) * (255.0 / 2)
        return x
```

```
    def print_shape(self, inputs):
        print(inputs.shape)
        x = self.conv2d_1(inputs)
        print(x.shape)
        x = self.relu(x)
        x = self.conv2d_2(x)
        print(x.shape)
        x = self.relu(x)
        x = self.conv2d_3(x)
        print(x.shape)
        x = self.relu(x)
        x = self.res_1(x)
        print(x.shape)
        x = self.res_2(x)
        print(x.shape)
        x = self.res_3(x)
        print(x.shape)
        x = self.res_4(x)
        print(x.shape)
        x = self.res_5(x)
        print(x.shape)
        x = self.deconv2d_1(x)
        print(x.shape)
        x = self.relu(x)
        x = self.deconv2d_2(x)
        print(x.shape)
        x = self.relu(x)
        x = self.deconv2d_3(x)
        print(x.shape)
```

5. **Initialise the style model using the style model class defined and define the Input to the encoder-decoder Style model**

```
style_model = StyleTransferModel()
input_shape = (256, 256, 3)
batch_size = 4
style_model.print_shape(tf.zeros(shape=(1, *input_shape)))
```

6. **Define the training utility function to Train the style model to minimise the perpetual loss**

```
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)

def   train_step(dataset,style_activations,steps_per_epoch,style_model,
loss_model,optimizer,checkpoint_path="./",content_weight=1e4,style_wei
ght=1e-2,total_variation_weight=0.004,content_layers_weights=[1],style
_layers_weights=[1] * 5):
  batch_losses = []
  steps = 1
  save_path = os.path.join(checkpoint_path, f"model_checkpoint.ckpt")
    print("Model Checkpoint Path: ", save_path)
    for input_image_batch in dataset:
```

```
        if steps - 1 >= steps_per_epoch:
            break
        with tf.GradientTape() as tape:
            outputs = style_model(input_image_batch)
            outputs = tf.clip_by_value(outputs, 0, 255)
               pred_activations = loss_model.get_activations(outputs /
255.0)
                                                    content_activations    =
loss_model.get_activations(input_image_batch)[
                "content"
            ]
            curr_loss = preceptual_loss(
                pred_activations,
                content_activations,
                style_activations,
                content_weight,
                style_weight,
                content_layers_weights,
                style_layers_weights,
            )
                                curr_loss   +=   total_variation_weight   *
tf.image.total_variation(outputs)
        batch_losses.append(curr_loss)
                                    grad    =    tape.gradient(curr_loss,
style_model.trainable_variables)
                                      optimizer.apply_gradients(zip(grad,
style_model.trainable_variables))
    if steps % 1000 == 0:
            print("checkpoint saved ", end=" ")
            style_model.save_weights(save_path)
            print(f"Loss: {tf.reduce_mean(batch_losses).numpy()}")
        steps += 1
    return tf.reduce_mean(batch_losses)


from google.colab import drive
     drive.mount("/gdrive")


!wget http://images.cocodataset.org/zips/train2014.zip
!mkdir coco
!unzip -qq train2014.zip -d coco
!rm train2014.zip

class TensorflowDatasetLoader:
    def __init__(
             self, dataset_path, batch_size=4, image_size=(256, 256),
num_images=None
    ):
                         images_paths   =   [str(path)   for   path   in
Path(dataset_path).glob("*.jpg")]
        self.length = len(images_paths)
        if num_images is not None:
```

```
            images_paths = images_paths[0:num_images]
                                                        dataset       =
tf.data.Dataset.from_tensor_slices(images_paths).map(
            lambda path: self.load_tf_image(path, dim=image_size),
            num_parallel_calls=tf.data.experimental.AUTOTUNE,
        )
        dataset = dataset.batch(batch_size, drop_remainder=True)
        dataset = dataset.repeat()
                                                        dataset       =
dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
        self.dataset = dataset

    def __len__(self):
        return self.length

    def load_tf_image(self, image_path, dim):
        image = tf.io.read_file(image_path)
        image = tf.image.decode_jpeg(image, channels=3)
        image = tf.image.resize(image, dim)
        image = image / 255.0
        image = tf.image.convert_image_dtype(image, tf.float32)
        return image
```

Due to the unavailability of TPUs and GPUs, we weren't able to train on the full COCO Dataset. We had sliced the dataset into 100 images and 400 images and did two separate training. We compared and analysed the difference in output obtained due to the difference in number training image.

For 100 images :

```
import os
import random
import shutil

# Define paths
source_folder = 'coco/train2014/'
destination_folder = 'coco/traindata'
num_images_to_select = 100

# List all files in the source folder
all_images = os.listdir(source_folder)

# Randomly select 100 images
selected_images = random.sample(all_images, num_images_to_select)

# Create destination folder if it doesn't exist
if not os.path.exists(destination_folder):
    os.makedirs(destination_folder)

# Move selected images to the destination folder
for image in selected_images:
    source_path = os.path.join(source_folder, image)
    destination_path = os.path.join(destination_folder, image)
```

```
    shutil.move(source_path, destination_path)

print("Selected images moved to the destination folder.")
```

For 400 images :

```
import os
import random
import shutil

# Define paths
source_folder = 'coco/train2014/'
destination_folder = 'coco/traindata'
num_images_to_select = 400

# List all files in the source folder
all_images = os.listdir(source_folder)

# Randomly select 400 images
selected_images = random.sample(all_images, num_images_to_select)

# Create destination folder if it doesn't exist
if not os.path.exists(destination_folder):
    os.makedirs(destination_folder)

# Move selected images to the destination folder
for image in selected_images:
    source_path = os.path.join(source_folder, image)
    destination_path = os.path.join(destination_folder, image)
    shutil.move(source_path, destination_path)

print("Selected images moved to the destination folder.")
```

For 5000 images :

```
import os
import random
import shutil

# Define paths
source_folder = 'coco/train2014/'
destination_folder = 'coco/traindata'
num_images_to_select = 5000

# List all files in the source folder
all_images = os.listdir(source_folder)

# Randomly select 400 images
selected_images = random.sample(all_images, num_images_to_select)

# Create destination folder if it doesn't exist
if not os.path.exists(destination_folder):
    os.makedirs(destination_folder)
```

```
# Move selected images to the destination folder
for image in selected_images:
    source_path = os.path.join(source_folder, image)
    destination_path = os.path.join(destination_folder, image)
    shutil.move(source_path, destination_path)

print("Selected images moved to the destination folder.")
```

The batch size remains the same in each case :

```
loader = TensorflowDatasetLoader("coco/traindata/", batch_size=4)
Loader.dataset.element_spec
plot_images_grid(next(iter(loader.dataset.take(1))))
```

Load the style image (Van Gogh Starry night image is used as style image) :

```
Url                                                                    =
"https://upload.wikimedia.org/wikipedia/commons/thumb/e/ea/Van_Gogh_-_
Starry_Night_-_Google_Art_Project.jpg/300px-Van_Gogh_-_Starry_Night_-_
Google_Art_Project.jpg"
style_image      =      load_url_image(url,      dim=(input_shape[0],
input_shape[1]), resize=True)
style_image = style_image / 255.0

show_image(style_image)

style_image = style_image.astype(np.float32)
style_image_batch = np.repeat([style_image], batch_size, axis=0)
style_activations                                                      =
loss_model.get_activations(style_image_batch)["style"]
```

7. Use the training utility function to Train the style model to minimise the perpetual loss

```
epochs = 2
content_weight = 1e1
style_weight = 1e2
total_variation_weight = 0.004

num_images = len(loader)
steps_per_epochs = num_images // batch_size
print(steps_per_epochs)

model_save_path = "../gdrive/My Drive"
top_folder_name = "scream"
save_path = os.path.join(model_save_path, top_folder_name)
print(save_path)

os.makedirs(os.path.join(model_save_path,            top_folder_name),
exist_ok=True)
```

```
try:
                                                    policy            =
tf.keras.mixed_precision.experimental.Policy("mixed_float16")
    tf.keras.mixed_precision.experimental.set_policy(policy)
except:
    pass

try:
    tf.config.optimizer.set_jit(True)
except:
    pass
if                                os.path.isfile(os.path.join(save_path,
"model_checkpoint.ckpt.index")):
                        style_model.load_weights(os.path.join(save_path,
"model_checkpoint.ckpt"))
    print("resuming training ...")
else:
    print("training scratch ...")

epoch_losses = []
for epoch in range(1, epochs + 1):
    print(f"epoch: {epoch}")
    batch_loss = train_step(
        loader.dataset,
        style_activations,
        steps_per_epochs,
        style_model,
        loss_model,
        optimizer,
        save_path,
        content_weight,
        style_weight,
        total_variation_weight,
        content_layers_weights,
        style_layers_weights,
    )
                        style_model.save_weights(os.path.join(save_path,
"model_checkpoint.ckpt"))
          print("Model  Checkpointed  at:  ",  os.path.join(save_path,
"model_checkpoint.ckpt"))
    print(f"loss: {batch_loss.numpy()}")
    epoch_losses.append(batch_loss)
plt.plot(epoch_losses)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training Process")
plt.show()
if                                os.path.isfile(os.path.join(save_path,
"model_checkpoint.ckpt.index")):
                        style_model.load_weights(os.path.join(save_path,
"model_checkpoint.ckpt"))
    print("loading weights ...")
```

```
else:
    print("no weights found ...")
```

8. **Load the content image to be styled , use the style model to produce the styled image in one pass and display the output**

```
test_image_url = "https://github.com/hwalsuklee/tensorflow-fast-style-transfer/raw/master/content/chicago.jpg"

test_image = load_url_image(test_image_url, dim=(640, 480))
test_image = np.expand_dims(test_image, axis=0)
test_image = test_image.astype(np.float32)
predicted_image = style_model(test_image)
predicted_image = np.clip(predicted_image, 0, 255)
predicted_image = predicted_image.astype(np.uint8)
test_output = test_image.astype(np.uint8)
test_output = tf.squeeze(test_output).numpy()
predicted_output = tf.squeeze(predicted_image).numpy()
plot_images_grid([test_output, predicted_output])
```

Chapter 5: Evaluation and Results

5.1 Evaluation :

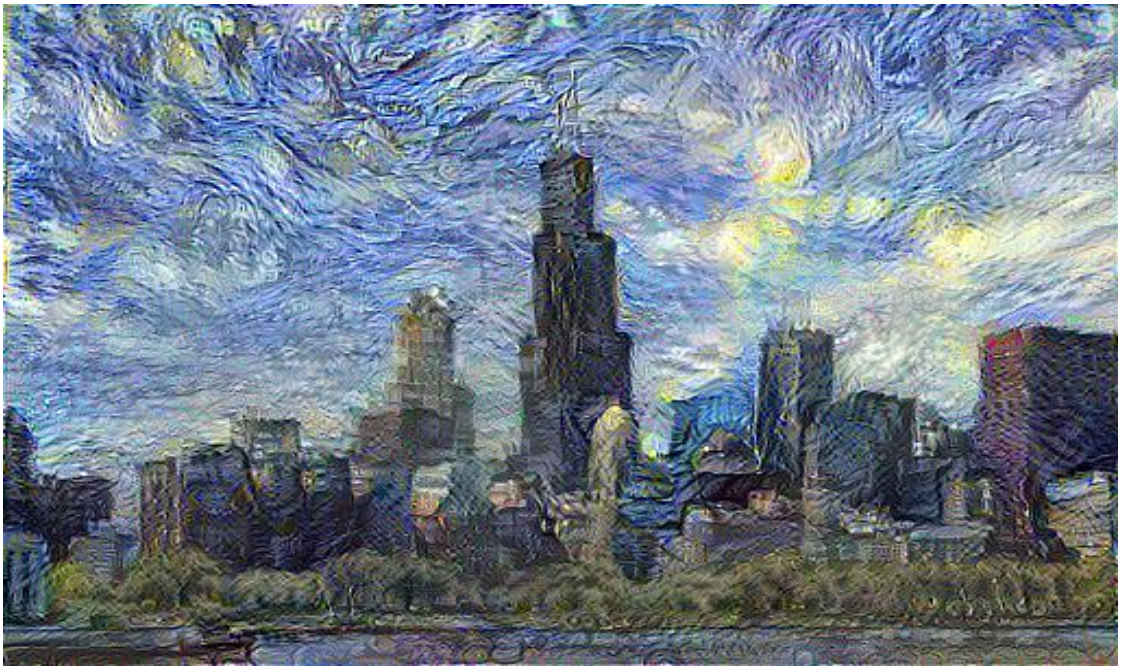
Gatys Style Transfer :

Evaluation of model's performance with 10 epochs and 100 steps per epoch

Content Image

Style Image

Result



- The loss function value computed after the first epoch was 16888666. It was minimised to 3958982 at the last step of the 10th epoch.
- The optimization process,which had 10 epochs and 100 steps per each epoch, took almost 3 hrs.

The observations are as follows :

1. Generating an image using Gatys style transfer takes time which depends on the compute power provided.

2. Gatys style transfer is usually not applied on real time videos since it is very slow.

3. The iteration process should be repeated separately for each pair of content and style images which can be time consuming.

4. There is no training involved in it ,we are just optimising the loss function which is a slow process

5. The output image depends on the weights given to the style image and content image. Hence it takes trial and error to arrive at the weights.

Fast Style Transfer :

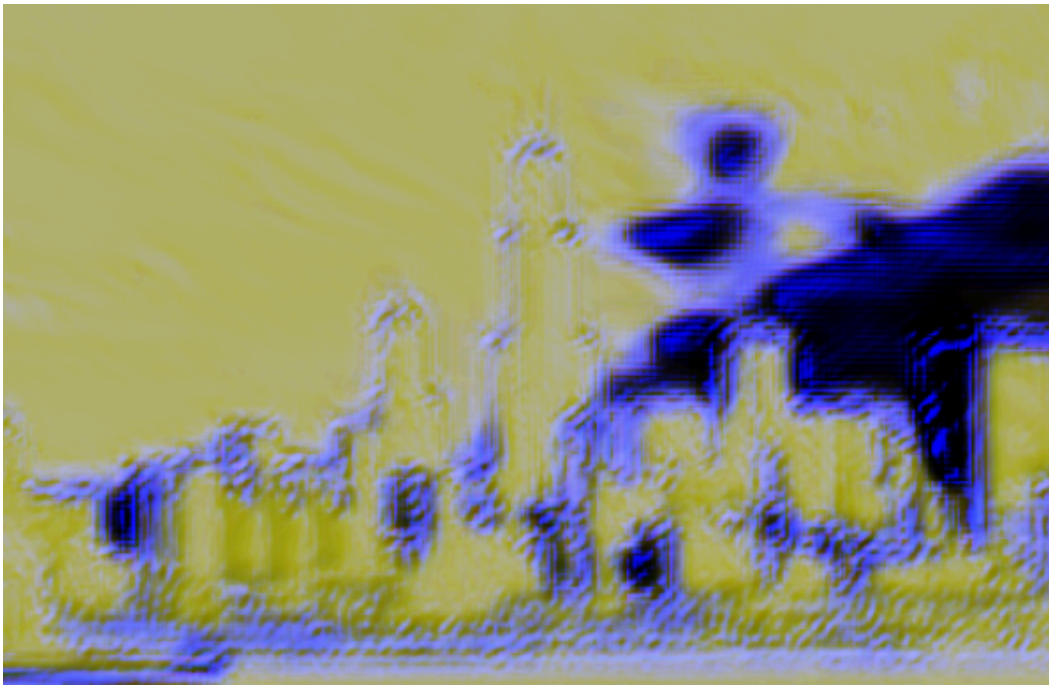
As mentioned in the challenges faced, due to limited availability of GPU(s) and TPU(s) the model was trained with 100 training images first and then trained on 400 training images. However to truly understand and evaluate the fast style approach, we need to train on a minimum of a few thousands of images.

Due to the limitations in the availability of hardware accelerators, training on more than 400 images took over 5 hrs on a normal office laptop and no conclusive output was achieved. Hence, we borrowed a gaming PC with 12GB of RAM and a graphics card with at least 8GB of memory built with an RTX 3050. The model was trained on 5000 training images .

The evaluation of model's performance in case of 100 images is as follows :

Style Image **Content Image**

Result



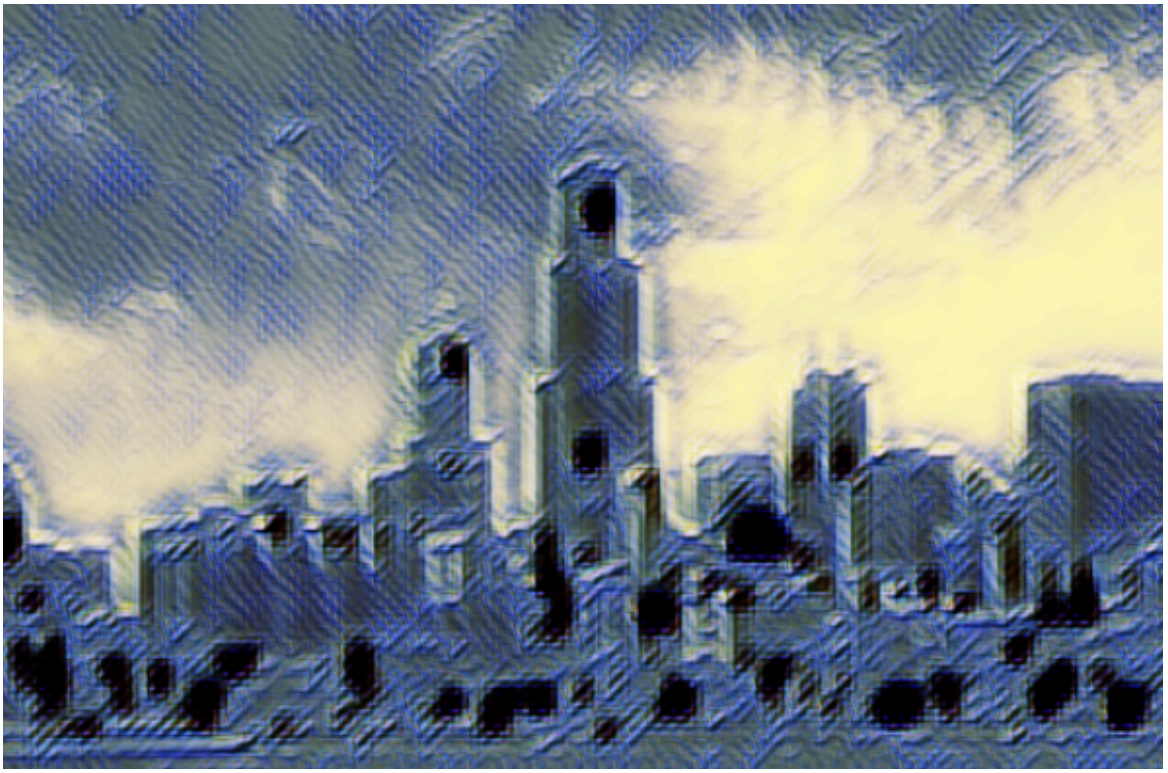
In the case of 100 training images, the styled output image has content loss and style loss. The training with the 100 training image took 45 mins.

The evaluation of model's performance in case of 400 images is as follows :

Style Image

Content Image

Result

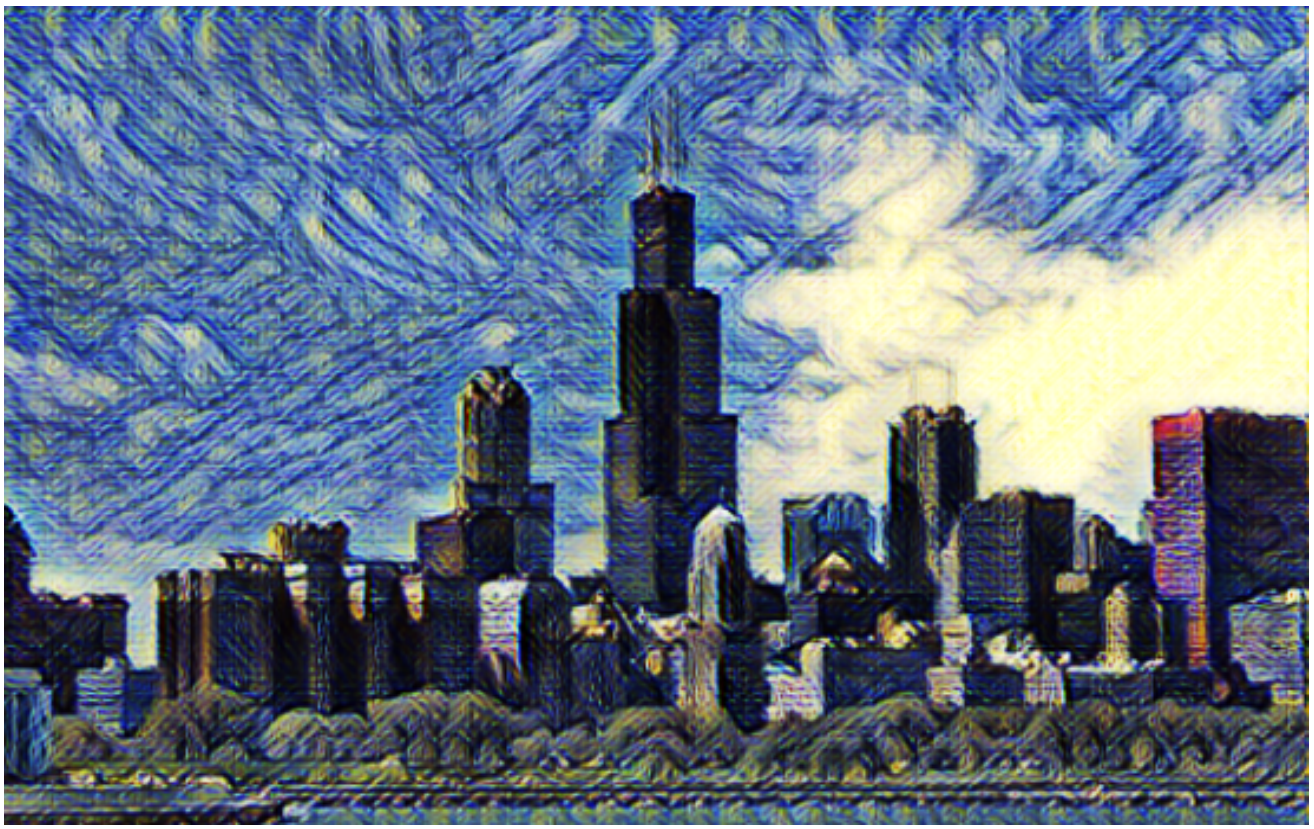


In the case of 400 training images, the styled output image has less content loss and less style loss compared to training with 100 images. The training with the 400 training image took 150 minutes.

The evaluation of model's performance while trained on 5000 images below: [The model was trained on a gaming PC with 12GB of RAM and a graphics card with at least 8GB of memory and built it with an RTX 3050.]

Style Image **Content Image**

Result



In the case of training the model without slicing the dataset, the styled output image has minimal content and style loss compared to training with 100 and 400 images. The training took around 15 hours.

Other content images can be styled in one-pass without iterative optimization. Some of the output styled images produced using the style model is given below :

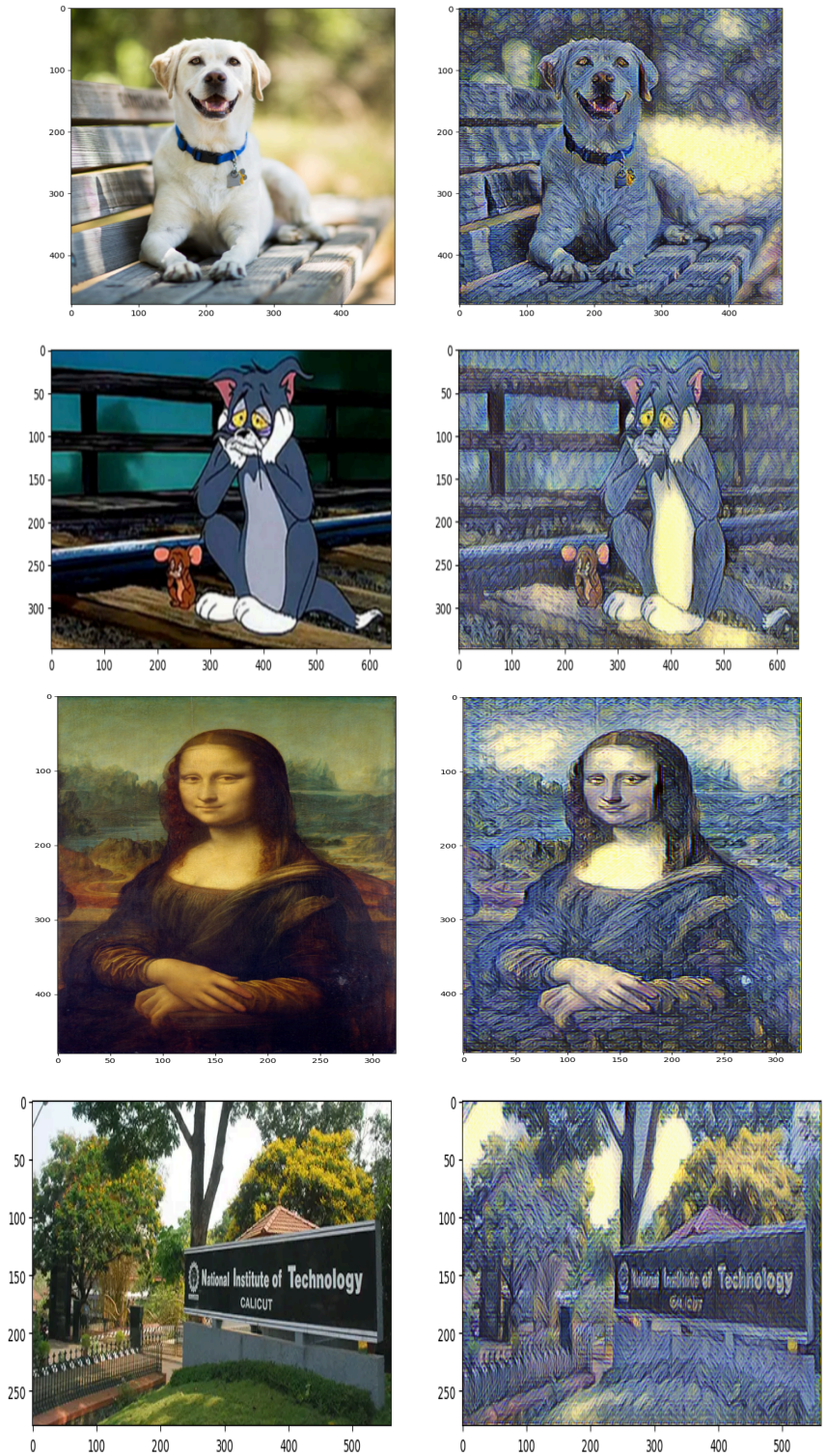


Fig : Dog, Tom & Jerry, Monalisa and NITC photos styled as starry night

Fig : Rithika Kathirvel’s Photo styled as Starry Night

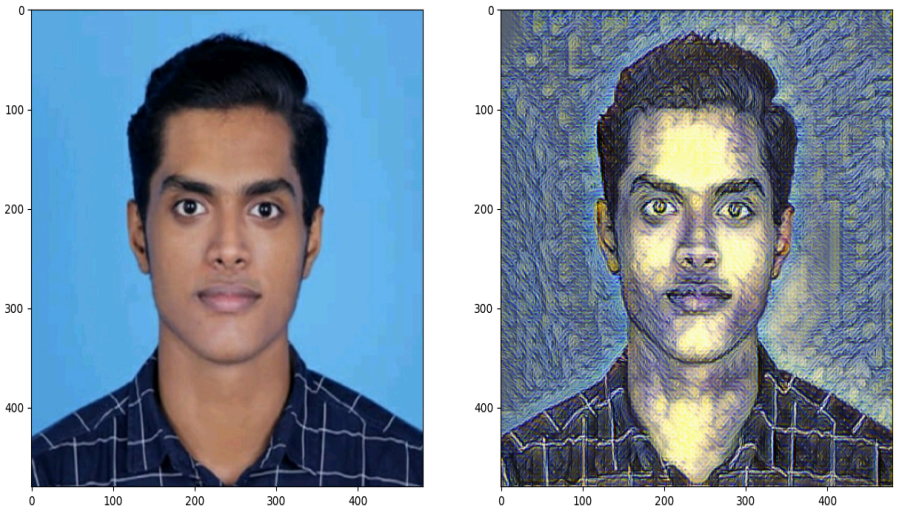


Fig : Aswin Sreekumar’s Photo styled as starry night

The observations are as follows:

1. For training the model using 100 images, 400 images and 5000 images. We see more training images, better the styled output image. This is because more the training image, the model sees more images to optimise the style model and tune it better.

2. The time taken to train the style model is significantly high when using limited computing resources.

3. However once the training is over, styled images for various content images can be generated in one pass unlike Gatys style transfer. Hence this method is preferred for real time style transfer.

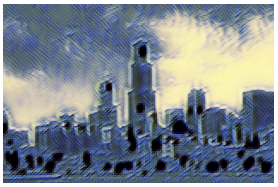
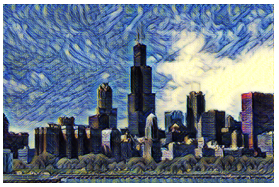
4. The output image depends on the weights given to the style image and content image. Hence it takes trial and error to arrive at the weights.

5. Fast style transfer is much faster than the traditional Gatys approach as it requires us to train the model only once per style.

6. The quality of the output largely depends on the number of images used for training. Hence in case of limited resources for training output, a styled image in case of fast style Transfer won't be as good as Gatys style transfer.

7. To style content image in other styles, a different model has to be trained and used.

5.2 Comparison of Results :

Comparison of Results	Gatys Style Transfer	Fast Style Transfer using 100 training image	Fast Style Transfer using 400 training image	Fast Style Transfer using 5000 training image
Output				
Time taken to train the model	NA	45 mins	150 mins	15 hours
Time taken to generate output styled image	3 hours	1 seconds	3 seconds	3 seconds

1. Comparison of algorithm used in both the approaches :

Gatys style transfer is based on optimising the content and style representations of two images using deep neural networks.

Fast style transfer employs feedforward neural networks, often based on convolutional neural networks (CNNs), to directly generate stylized images. It does not involve iterative optimization like Gatys style transfer.

2. **Comparison of speed of both the approaches :**

   Gatys style transfer can be computationally intensive, especially during the optimization process, which involves multiple iterations to generate the stylized image. This iterative process can be slow, particularly for high-resolution images.

   Fast Style transfer utilises a trained style model for generating styled images of various content images without iterative optimization and hence is faster and real-time or near-real-time stylization of images. However, the process of training the model itself can be computationally intensive and time consuming.

3. **Comparison of the quality of the output styled image :**

   Gatys style transfer is known for producing high-quality stylized images with fine-grained style details while preserving content of the content image.. However, this quality often comes at the expense of longer processing times.

   Fast style transfer approach sacrifices some level of detail compared to Gatys style transfer. The quality of the styled image depends on the number of images the style model was trained on. So the quality of stylized image in case of fast style transfer is less compared to that achieved while using Gatys style transfer approach. However, it offers practical solutions for real-time or near-real-time stylization applications.

5.3 Conclusion

Gatys style transfer and Fast style transfer are both techniques used in neural style transfer, a process of applying the artistic style of one image (style image) to another image (content image) while preserving the content of the latter. In conclusion, Gatys style transfer offers high-quality stylization but can be computationally expensive and slow, while fast style transfer methods provide real-time style transfer at the cost of some loss in quality. The choice between the two approaches depend on the specific requirements of the application, including speed, quality, and computational resources available.

6 References

[1] A neural algorithm of artistic style, 2015 Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge
[2] Perceptual Losses For Real-Time Style Transfer And Super-Resolution Justin Johnson, Alexandre Alahi, Li Fei-Fei
[3] Image Neural Style Transfer With Global And Local Optimization Fusion Hui-Huang Zhao, Paul L Rosin, Yu Kun Lai, Mu-Gang Lin, Qin-Lin Liu
[4] Instance Normalisation: The Missing Ingredient for Fast Stylization Dmitry Ulyanov, Andrea Vedaldi, Victor Lempitsky
[5] Towards Data Science : Neural Style Transfer.
[6] Tarush Bisht : Neural Style Transfer.