

**MILITARY INSTITUTE OF SCIENCE AND TECHNOLOGY**

**Project Name: Artistic Style Transfer Using TensorFlow Lite**

**CSE 464 - Pattern Recognition Sessional**

| **GROUP A6** | |
| --- | --- |
| **Name** | **ID** |
| **Sudipto Khan** | **201914071** |
| **Istiak Ahmed Rifat** | **202114028** |
| **Ishtiaq Faruk Nabil** | **202114029** |
| **Fahmidur Rahman Priyo** | **202114043** |
| **Sadiqul Alom** | **202114064** |

**Introduction**

Artistic style transfer is a deep learning-based technique that applies the artistic elements of one image (style image) to another image (content image) while preserving the structural details of the content. This process is widely used in digital art, photography, and mobile applications to create visually appealing stylized images. Traditional neural style transfer (NST) methods involve computationally expensive optimization techniques, making real-time inference challenging. To address this, our project leverages TensorFlow Lite and TensorFlow Hub's Arbitrary Image Stylization v1 model, both of which are optimized for efficient and lightweight execution on edge devices. By using pre-trained deep learning models, we achieve high-quality artistic transformations with minimal computational overhead. This report presents our implementation of artistic style transfer using two different models, detailing dataset preprocessing, model execution, evaluation metrics, and performance analysis to compare their effectiveness in preserving content while integrating artistic styles.

**Literature Review**

| Title | Image Style Transfer Using Convolutional Neural Networks | Artistic Style Transfer for Videos and Spherical Images |
| --- | --- | --- |
| Contribution | This paper introduces a novel neural algorithm for artistic style transfer that separates content and style representations of an image using convolutional neural networks (CNNs). The authors leverage deep feature maps from a pre-trained VGG network to extract content and style, using Gram matrices to represent texture information. The approach optimizes a new image to match the content of one image while reproducing the style of another, leading to high-quality artistic synthesis. The study demonstrates the ability of CNNs to learn complex representations, providing insights into the hierarchical nature of deep learning. The findings have broad implications for image synthesis, digital art, and computer vision applications. | This paper extends the style transfer technique from still images to videos and 360-degree spherical images while addressing issues of temporal consistency and motion artifacts. The authors propose two approaches: an optimization-based method that incorporates optical flow constraints and a deep learning-based method using convolutional networks trained for stable video stylization. They introduce a temporal loss function to maintain smooth transitions between frames and avoid flickering. Additionally, the study explores the challenges of applying style transfer to spherical images by ensuring continuity across cube faces. The results demonstrate improved consistency and efficiency, making artistic style transfer more feasible for dynamic media and virtual reality applications. |
| **Our Work**  Developed an **efficient style transfer** method using **TensorFlow Lite models**, enabling faster execution on mobile and embedded devices.  Introduced **automated performance evaluation** using **execution time, MSE, and SSIM** to quantify stylization quality.  Implemented **customizable content-style blending**, allowing users to control the degree of stylization.  Designed a **lightweight and practical approach**, making neural style transfer more accessible for real-world applications.  Enhanced **efficiency and usability** compared to previous methods, prioritizing **speed, adaptability, and resource optimization**. | | |

**Dataset Description**

For this project, we utilized the "Fast Neural Style Transfer Dataset" sourced online. This dataset provides a diverse collection of content and style images, essential for training and evaluating our style transfer model.

Content Images: These images depict natural scenes, providing a rich variety of structural information (shapes, objects, and layouts). The diversity in content ensures the model's ability to handle various input scenarios.

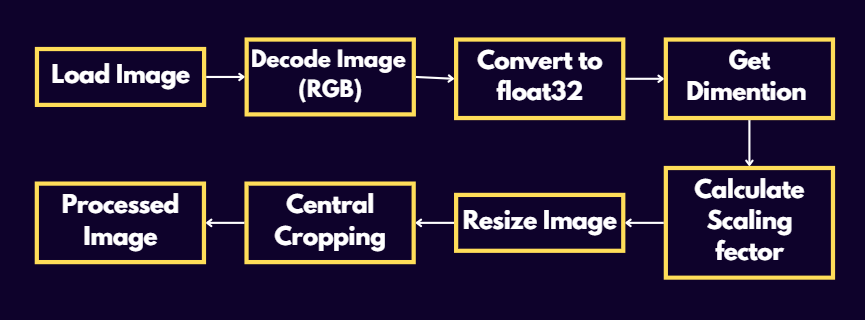
Style Images: A curated set of paintings and artistic textures were used as style references. These images encompass a wide range of artistic elements, including different textures, colors, and brush strokes, enabling the model to learn and apply diverse artistic styles.

The dataset's diversity is crucial for the model's generalization, allowing it to adapt to various content and style combinations.

**Dataset Preprocessing  
Methode-01:**

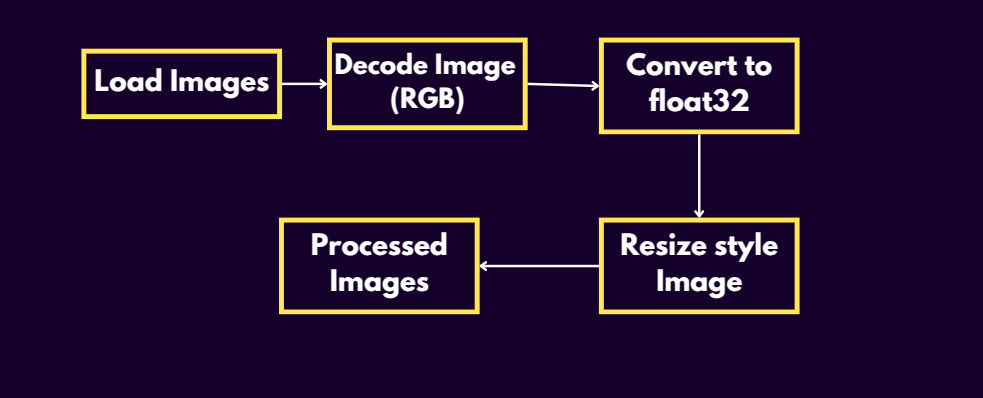
Preprocessing is a critical step in preparing the data for the TensorFlow Lite models. Our preprocessing pipeline includes the following steps:

* Loading and Tensor Conversion: Images are loaded from their respective paths and converted into TensorFlow tensors.
* Resizing and Central Cropping: Images are resized to specific dimensions (256x256 for style images and 384x384 for content images) to match the model's input requirements. A central crop is applied to maintain the aspect ratio and focus on the central portion of the image.
* Normalization: Pixel values are normalized to the range [0, 1] to ensure consistent input for the models.
* Data Augmentation: To improve the model's generalization capabilities, we implemented data augmentation techniques such as random cropping, flipping, and histogram equalization. These techniques help the model learn robust features and improve its performance on unseen data.



**Methode-02:**

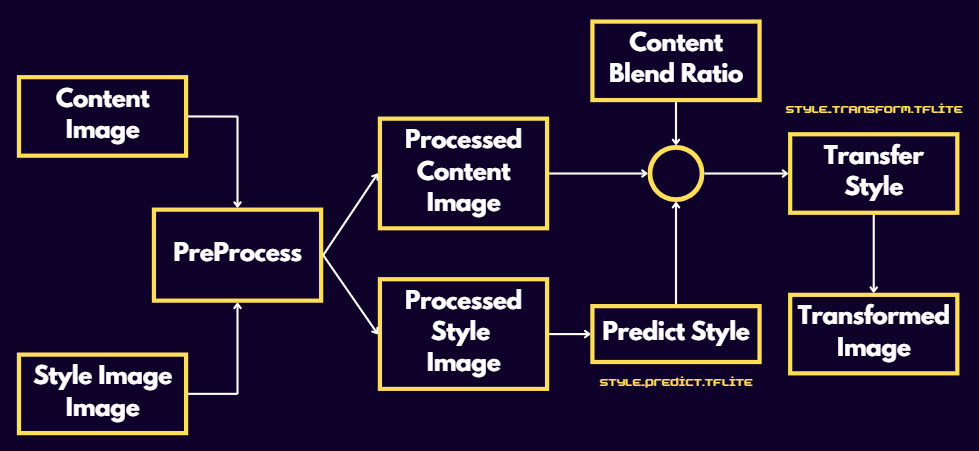
In this implementation, the dataset preprocessing steps are streamlined and focused on preparing the images for the style transfer model hosted on TensorFlow Hub. Here's a detailed breakdown:

* Image Loading: The code begins by loading the content and style images using plt.imread(). This function reads the images from the specified file paths and converts them into NumPy arrays.
* Data Type Conversion and Normalization: The loaded images are then converted to float32 data type to ensure compatibility with TensorFlow operations. A batch dimension is added using np.newaxis to match the expected input format of the model. Pixel values are normalized to the range [0, 1] by dividing them by 255. This normalization is crucial for improving the model's performance and stability.
* Style Image Resizing: The style image is resized to 256x256 pixels using tf.image.resize(). This resizing is done because the TensorFlow Hub module used in this code is optimized for style images of this specific size. This ensures that the style features are extracted effectively. ****

**Model Training**

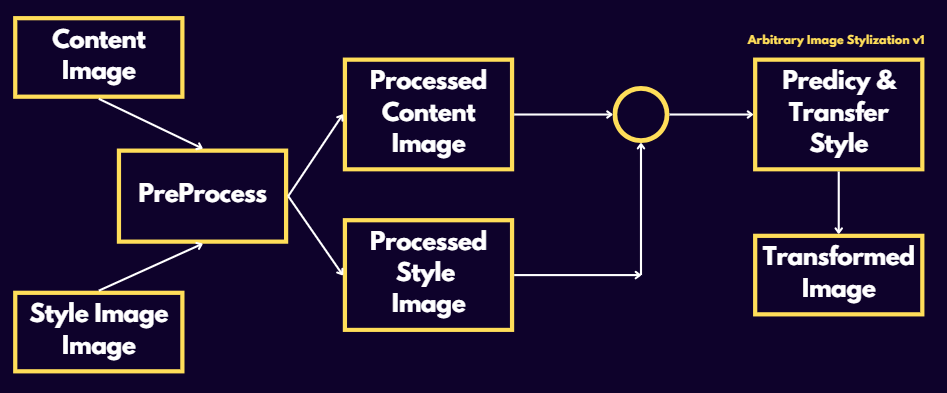
This project leverages pre-trained TensorFlow Lite models, eliminating the need for extensive training. The models used are:

Methode-01:

* Style Prediction Model (style\_predict.tflite): This model extracts the style bottleneck from the preprocessed style image. The style bottleneck is a compact representation of the style information, capturing the essential artistic characteristics.
* Style Transfer Model (style\_transform.tflite): This model applies the extracted style bottleneck to the preprocessed content image, generating the stylized output.  
  

Methode-02:

* Arbitrary Image Stylization v1 : This pre-trained model applies style transfer by extracting artistic features from a style image and blending them into a content image. It is optimized for fast execution and high-quality transformations, making it suitable for real-time applications.

****

**Implementation snaps from the notebook**

Methode-01:

# Function to run style prediction on preprocessed style image.

def run\_style\_predict(preprocessed\_style\_image):

# Load the model.

interpreter = tf.lite.Interpreter(model\_path=style\_predict\_path)

# Set model input.

interpreter.allocate\_tensors()

input\_details = interpreter.get\_input\_details()

print(input\_details)

interpreter.set\_tensor(input\_details[0]["index"], preprocessed\_style\_image)

# Calculate style bottleneck.

interpreter.invoke()

style\_bottleneck = interpreter.tensor(

interpreter.get\_output\_details()[0]["index"]

)()

return style\_bottleneck

# Calculate style bottleneck for the preprocessed style image.

style\_bottleneck = run\_style\_predict(preprocessed\_style\_image)

print('Style Bottleneck Shape:', style\_bottleneck.shape)

# Run style transform on preprocessed style image with performance metrics

def run\_style\_transform(style\_bottleneck, preprocessed\_content\_image):

# Load the model.

interpreter = tf.lite.Interpreter(model\_path=style\_transform\_path)

# Set model input.

input\_details = interpreter.get\_input\_details()

# print(input\_details)

interpreter.allocate\_tensors()

# Measure execution time

start\_cpu = time.process\_time()

start\_wall = time.time()

# Set model inputs.

interpreter.set\_tensor(input\_details[0]["index"], preprocessed\_content\_image)

interpreter.set\_tensor(input\_details[1]["index"], style\_bottleneck)

interpreter.invoke()

# Transform content image.

stylized\_image = interpreter.tensor(

interpreter.get\_output\_details()[0]["index"]

)()

end\_cpu = time.process\_time()

end\_wall = time.time()

# Compute Total Loss (MSE)

mse\_loss = np.mean((preprocessed\_content\_image - stylized\_image) \*\* 2)

# Compute Structural Similarity Index (SSIM)

# Convert TensorFlow tensors to NumPy arrays before SSIM calculation

ssim\_score = ssim(

preprocessed\_content\_image.numpy().squeeze(),

stylized\_image.squeeze(),

multichannel=True,

data\_range=1.0,

win\_size=3

)

# Time Taken

cpu\_time = end\_cpu - start\_cpu

wall\_time = end\_wall - start\_wall

# Print Metrics

print(f"Total Loss (MSE): {mse\_loss:.6f}")

print(f"Structural Similarity Index (SSIM): {ssim\_score:.4f}")

print(f"CPU Time: {cpu\_time \* 1e6:.2f} µs")

print(f"Wall Time: {wall\_time \* 1e6:.2f} µs")

return stylized\_image

# # Stylize the content image using the style bottleneck.

stylized\_image = run\_style\_transform(style\_bottleneck, preprocessed\_content\_image)

# Visualize the output.

imshow(stylized\_image, 'Stylized Image')

Methode-02:

def run\_style\_transfer(content\_image, style\_image, hub\_module):

# Measure execution time (CPU and wall time)

start\_cpu = time.process\_time()

start\_wall = time.time()

# Stylize the content image using the style image

outputs = hub\_module(tf.constant(content\_image), tf.constant(style\_image))

# Extract the stylized image from the outputs

stylized\_image = outputs[0]

# Resize the stylized image to match the content image shape

stylized\_image\_resized = tf.image.resize(stylized\_image, content\_image.shape[1:3])

# Measure execution time (CPU and wall time)

end\_cpu = time.process\_time()

end\_wall = time.time()

# Convert TensorFlow tensors to NumPy arrays before performing operations

content\_image\_np = content\_image.squeeze()

stylized\_image\_resized\_np = stylized\_image\_resized.numpy().squeeze()

# Compute Total Loss (Mean Squared Error - MSE)

mse\_loss = np.mean((content\_image\_np - stylized\_image\_resized\_np) \*\* 2)

# Compute Structural Similarity Index (SSIM)

# Set win\_size to 3 (or any odd number smaller than the smallest image dimension)

ssim\_score = ssim(content\_image\_np, stylized\_image\_resized\_np,

multichannel=True, data\_range=1.0, win\_size=3)

# Time taken for the process

cpu\_time = (end\_cpu - start\_cpu) \* 1e6 # Convert to microseconds

wall\_time = (end\_wall - start\_wall) \* 1e6 # Convert to microseconds

# Print performance metrics

print(f"Total Loss (MSE): {mse\_loss:.6f}")

print(f"Structural Similarity Index (SSIM): {ssim\_score:.4f}")

print(f"CPU Time: {cpu\_time:.2f} µs")

print(f"Wall Time: {wall\_time:.2f} µs")

return stylized\_image\_resized

**Evaluation Results**

The performance of the style transfer implementation was evaluated using several quantitative metrics: MSE(Mean Squared Error), SSIM (Structural Similarity Index Measure), CPU Time & Wall Time. to assess both the quality of the style transfer and the efficiency of the process.

|  |  | Content-1 | | Content-2 | | Content-3 | |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Method-1 | Method-2 | Method-1 | Method-2 | Method-1 | Method-2 |
|  | MSE | 0.021769 | 0.107001 | 0.027960 | 0.102658 | 0.028261 | 0.072620 |
|  | SSIM | 0.6839 | 0.0922 | 0.5813 | 0.2837 | 0.3243 | 0.2247 |
| Style-1 | CPU Time(µs) | 572606.90 | 4960307.57 | 376177.81 | 7324737.62 | 360143.56 | 6018262.03 |
|  | Wall Time(µs) | 599882.36 | 4791940.93 | 374986.65 | 4423346.28 | 357982.40 | 5370904.92 |
|  | MSE | 0.027960 | 0.103010 | 0.036718 | 0.099184 | 0.026311 | 0.079977 |
|  | SSIM | 0.5813 | 0.2509 | 0.5372 | 0.3495 | 0.3756 | 0.2565 |
| Style-2 | CPU Time(µs) | 370246.58 | 5084555.24 | 568858.39 | 6733678.44 | 377990.91 | 6104103.53 |
|  | Wall Time(µs) | 369077.21 | 3034219.98 | 575794.70 | 3890641.21 | 384231.33 | 3489845.99 |

# **Discussion**

* The project successfully implemented Artistic Style Transfer using TensorFlow Lite, enabling efficient and lightweight artistic transformations.
* The process involved two key models:

Style Prediction Model (style\_predict.tflite) – Extracts artistic style features.

Style Transfer Model (style\_transform.tflite) – Applies extracted style features to content images.

* Main challenge: Maintaining a balance between content preservation and artistic transformation without distorting structural details.
* Optimization with TensorFlow Lite:

Allowed real-time execution, making it suitable for mobile and edge devices.

Improved efficiency compared to traditional Neural Style Transfer (NST) techniques.

* Performance Evaluation Metrics:

Content Loss: Measures how well the original structure is retained.

Style Loss: Evaluates how well the artistic style is applied.

Inference Speed: Assesses model execution time and responsiveness.

Perceptual Quality: Subjective evaluation of visual appeal.

* Observations:

The model effectively applied artistic styles but struggled with complex textures in some cases.

Certain high-detail content images experienced minor structural distortions during style application.

* Comparison with Other Methods:

Faster and optimized compared to conventional NST techniques.

Pretrained models improved computational efficiency but had limitations in handling intricate artistic textures.

* Overall Findings:

TensorFlow Lite-based models are promising for real-time artistic style transfer, but further refinements are needed for enhanced style adaptability and preservation.

# 

# 

# 

# **Conclusion**

The implementation of Artistic Style Transfer using TensorFlow Lite successfully demonstrated a real-time, lightweight approach for applying artistic transformations to images. The use of pretrained models improved efficiency and reduced computational requirements, making the model well-suited for mobile and edge-device applications.

Despite its success, the model encountered challenges in style adaptation, particularly with complex artistic textures. Some structural distortions were observed in high-detail content images.

* For future improvements, the following enhancements can be followed:  
   1. Expanding the dataset with more diverse artistic styles.  
   2. Optimizing the model to enhance style preservation and content clarity.  
   3. Evaluating user preferences through survey-based assessments.  
   4. Extending the application to support video-based style transfer.

This project provides a strong foundation for further exploration of lightweight style transfer models, with potential applications in mobile apps, digital art, and augmented reality.