SOMAIYA
VIDYAVIHAR

K J Somaiya Institute of Technology
(Formerly known as K J Somaiya Institute of Engineering and Information Technology)
An Autonomous Institute Permanently Affiliated to University of Mumbai.

Somaiya
TRUST

Department of COMP Engineering

AI IA 2 : Neural Style Transfer

Peeth Chowdhary - 16010122034
Rohit Deshpande - 16010122041
Eeshanya Joshi - 16010122074
Shubh Jalui - 16010122072
Ninad Marathe - 16010122106

Abstract

Neural Style Transfer (NST) merges the content of one image with the artistic style of another.

This project implements NST using two approaches: PyTorch-based optimization and TensorFlow Hub-based model inference.

Stylized outputs are generated using user-supplied images.

Problem Statement

Implement Neural Style Transfer for combining a content image with a style image
Achieve aesthetically pleasing style transfers using accessible tools and frameworks.

Objectives

- Implement NST using both optimization-based and pre-trained model-based techniques.
- Provide a user-friendly interface for uploading and processing images.
- Compare effects of different loss functions and hyperparameters on stylization quality.

Literature Survey

PUBLICATION YEAR	TITLE OF THE PAPER	OBJECTIVE
2017	Neural Style Transfer: A Review	To provide an overview of NSTs and the research conducted around them
2025	Dynamic Neural Style Transfer for Artistic Image Generation using VGG19	To test a different methodology of using VGG19 in ways that allow for flexible adjustments to style weight ratios and reduce processing times

Implementation plan

PyTorch Version:

- Use a pre-trained VGG19 to extract content and style features.
- Compute style using Gram matrices; optimize a generated image to minimize style/content loss.

TensorFlow Version:

- Use TensorFlow Hub’s pre-trained NST model for fast, high-quality transfer.
- Deploy with a simple Streamlit web interface for user interaction.

Tech Stack:

Languages: Python

Frameworks: PyTorch, TensorFlow, Streamlit

Libraries: OpenCV, PIL, NumPy, torchvision, TensorFlow Hub

Implementation 1

```
import streamlit as st        Rohit Deshpande, last week • ai ia2 done …
import tensorflow as tf
import tensorflow_hub as hub
import numpy as np
from PIL import Image
from io import BytesIO

# Set page config first
st.set_page_config(page_title="Neural Style Transfer", layout="wide")

# Cache the model
@st.cache_resource
def load_model():
    return hub.load('https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-256/2')

stylize_model = load_model()

def load_image(image_file, image_size=(512, 512)):
    """Load and preprocess an image."""
    try:
        img = Image.open(image_file).convert('RGB')
        img = np.array(img, dtype=np.float32) / 255.0
        img = tf.image.resize(img, image_size, preserve_aspect_ratio=True)
        return img[tf.newaxis, :]
    except Exception as e:
        st.error(f"Error loading image: {e}")
        return None

def export_image(tf_img):
    """Convert tensor to downloadable PNG."""
    img = np.squeeze(tf_img, axis=0) * 255
```

Deployment using Streamlit for real-time stylization using arbitrary-image-stylization-v1

Implementation 1

```
def export_image(tf_img):
    """Convert tensor to downloadable PNG."""
    img = np.squeeze(tf_img, axis=0) * 255
    pil_img = Image.fromarray(img.astype(np.uint8))
    buffer = BytesIO()
    pil_img.save(buffer, format="PNG")
    return buffer.getvalue()

def main():
    """Streamlit UI for style transfer."""
    st.sidebar.title("Style Transfer")
    st.sidebar.write("Blend content with style!")

    # File uploaders
    content_file = st.sidebar.file_uploader("Content Image", ["jpg", "jpeg", "png"], help="Image to stylize")
    style_file = st.sidebar.file_uploader("Style Image", ["jpg", "jpeg", "png"], help="Style source")

    # Layout columns
    col1, col2, col3 = st.columns(3)

    # Process and display images
    content_img = load_image(content_file) if content_file else None
    if content_file:
        col1.header("Content")
        col1.image(content_file, use_container_width=True)

    style_img = load_image(style_file, (256, 256)) if style_file else None
    if style_file:
        col2.header("Style")
        col2.image(style_file, use_container_width=True)
```

Implementation 1

```
    style_img = load_image(style_file, (256, 256)) if style_file else None
    if style_file:
        col2.header("Style")
        col2.image(style_file, use_container_width=True)

    # Styling button
    if st.sidebar.button("Stylize"):
        if content_img is not None and style_img is not None:
            with st.spinner("Generating..."):
                stylized_img = stylize_model(content_img, style_img)[0].numpy()
                col3.header("Result")
                col3.image(stylized_img, use_container_width=True)
                col3.download_button(
                    "Download",
                    export_image(stylized_img),
                    "stylized_image.png",
                    "image/png"
                )
        else:
            st.sidebar.error("Upload both images.")

if __name__ == "__main__":
    main()
```

Implementation 2

```
import os
import time
import torch
import torch.nn as nn
import torch.optim as optim
from PIL import Image
import torchvision.transforms as transforms
import torchvision.models as models
from torchvision.utils import save_image


class VGG(nn.Module):
    def __init__(self):
        super(VGG, self).__init__()

        self.chosen_features = ['0', '5', '10', '19', '28']
        self.model = models.vgg19(pretrained=True).features[:29]

    def forward(self, x):
        features = []
        for layer_num, layer in enumerate(self.model):
            x = layer(x)
            if str(layer_num) in self.chosen_features:
                features.append(x)
        return features


def load_image(image_name):
    image = Image.open(image_name)
    image = loader(image).unsqueeze(0)
    return image.to(device)


def data_collection(lr, m, fname):
    model = VGG().to(device).eval()  # .eval() freezes the weights
    original_image = load_image('times_square.jpg')
    style_image = load_image('style2.jpg')
```

Loss Function = Content loss (MSE or MAE) + Style loss using Gram matrices

Training Parameter = Varying learning rates (0.1, 1, 10)

Implementation 2

```
def data_collection(lr, m, fname):
    model = VGG().to(device).eval()  # .eval() freezes the weights
    original_image = load_image('times_square.jpg')
    style_image = load_image('style2.jpg')
    # generated_image = torch.randn(original_image.shape, device=device, requires_grad=True) # --> This is just noise
    generated_image = original_image.clone().requires_grad_(True)  # Seems to work better than starting off as noise

    # hyper parameters
    total_steps = 2000  # try 3000 and lower if taking too long
    learning_rate = lr  # [0.001]
    alpha = 1  # content multiplier
    beta = 200  # style multiplier
    optimizer = optim.Adam([generated_image], lr=learning_rate)

    file = open(fname+".csv", "w")
    line = 'step_number,total_loss,content_loss,style_loss,time'
    file.writelines(str(line) + '\n')
```

Implementation 2

```
for step in range(total_steps + 1):
    start = time.time()
    generated_image_features = model(generated_image)
    original_image_features = model(original_image)
    style_image_features = model(style_image)

    style_loss = content_loss = 0
    for gen_feature, orig_feature, style_feature in zip(
            generated_image_features, original_image_features, style_image_features):
        batch_size, channel, height, width = gen_feature.shape

        # Compute Gram Matrix for generated_image
        gen_matrix = gen_feature.view(channel, height * width)
        gen_gram_matrix = gen_matrix.mm(gen_matrix.t())

        # Compute Gram Matrix for style_image
        style_matrix = style_feature.view(channel, height * width)
        style_gram_matrix = style_matrix.mm(style_matrix.t())

        if m == 'rms':
            content_loss = content_loss + torch.mean((gen_feature - orig_feature) ** 2)
            style_loss = style_loss + torch.mean((gen_gram_matrix - style_gram_matrix) ** 2)
        else:
            mae_loss = torch.nn.L1Loss()
            content_loss = content_loss + mae_loss(gen_feature, orig_feature)
            style_loss = style_loss + mae_loss(gen_gram_matrix, style_gram_matrix)

    total_loss = alpha * content_loss + beta * style_loss
    optimizer.zero_grad()
    total_loss.backward()
    optimizer.step()
    t = f'Time: {time.time() - start}'
    if step % 50 == 0:
        print(total_loss)
        save_image(generated_image, fname + '/' + fname + ',' + str(step) + '.png')

    line = str(step) + ',' + str(total_loss.item()) + ',' + str(content_loss.item()) + ',' + str(style_loss.item()) + ',' + str(t)
    file.writelines(str(line) + '\n')
file.close()
```

Implementation 2

```python
if __name__ == '__main__':
    device = torch.device('cpu' if not (torch.cuda.is_available()) else 'cuda')
    image_size = 356  # use 178 = 356/2 or 89 if computation taking too long

    loader = transforms.Compose(
        [
            transforms.Resize((image_size, image_size)),
            transforms.ToTensor(),
        ]
    )
    learning_rates = [0.001, 0.01, 0.1]
    metrics = ['mae', 'rms']
    for i in range(1, 4):
        for lr in learning_rates:
            for metric in metrics:
                file_name = '{}_{}_t{}'.format(lr, metric, i)
                os.mkdir(file_name)
                data_collection(lr, metric, file_name)
```

Implementation 2

```
if __name__ == '__main__':
    device = torch.device('cpu' if not (torch.cuda.is_available()) else 'cuda')
    image_size = 356  # use 178 = 356/2 or 89 if computation taking too long

    loader = transforms.Compose(
        [
            transforms.Resize((image_size, image_size)),
            transforms.ToTensor(),
        ]
    )
    content_multipliers = [0.001, 0.01, 0.1, 1, 10]
    style_multipliers = [x for x in range(200, 601, 100)]
    for content_multiplier in content_multipliers:
        for style_multiplier in style_multipliers:
            file_name = '{},{}'.format(content_multiplier,style_multiplier)
            os.mkdir(file_name)
            data_collection(content_multiplier, style_multiplier, file_name)
```

Slightly altered code that changes the content and style multipliers (alpha + beta) instead of learning rates and ways to calculate loss.

Implementation 2

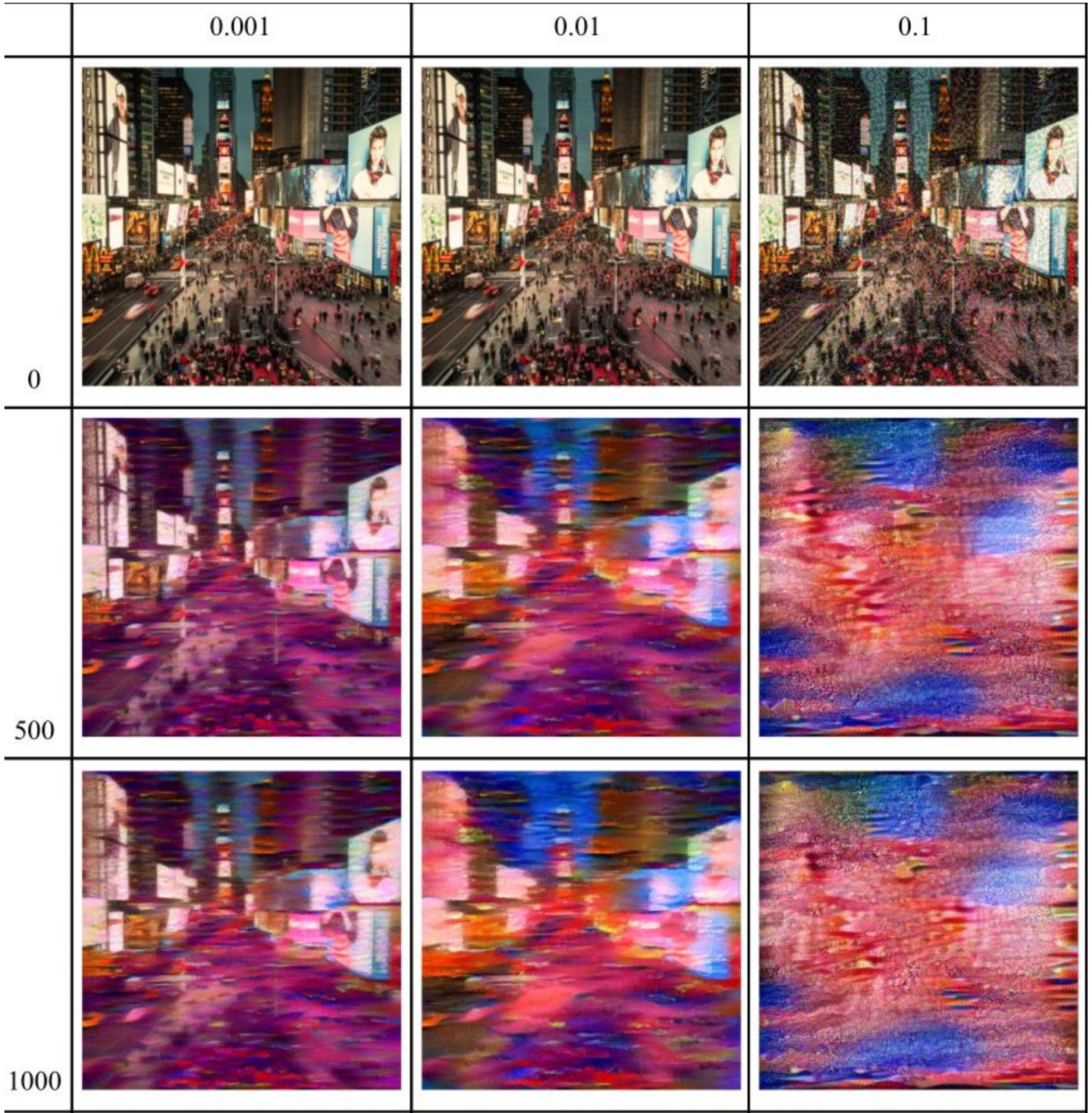
```
def data_collection(a, b, fname):
    model = VGG().to(device).eval()  # .eval() freezes the weights
    original_image = load_image('dog.jpeg')
    style_image = load_image('style.jpeg')
    # generated_image = torch.randn(original_image.shape, device=de
    generated_image = original_image.clone().requires_grad_(True)

    # hyper parameters
    total_steps = 6000  # try 3000 and lower if taking too long
    learning_rate = 0.01  # [0.001]
    alpha = a  # content multiplier
    beta = b  # style multiplier
    optimizer = optim.Adam([generated_image], lr=learning_rate)

    file = open(fname+".csv", "w")
    line = 'step_number,total_loss,content_loss,style_loss'
    file.writelines(str(line) + '\n')
```

Associated changes made to the beginning of the data_collection method

Implementation 2 Results



Changes in learning rate and how that affected the generated image over 100s of iterations

Conclusion

- Both optimization-based and model-inference approaches to NST are effective.
- The project showcases the flexibility and creativity enabled by neural networks in digital art generation.

References

- Studies mentioned in literature survey on slide 5