PROJECT REPORT

Neural Style Transfer

Problem Statement

The challenge was to create an application capable of applying the artistic style of one image to the content of another image. This involves complex machine learning and deep learning techniques, particularly in the field of convolutional neural networks (CNNs). The aim was to develop a solution that could be easily used through a web interface, making advanced neural style transfer techniques accessible to a broader audience.

Project Description

Our project on Neural Style Transfer focused on leveraging machine learning and deep learning to merge the content of one image with the style of another. This was achieved using a dataset sourced from Kaggle, which provided a rich variety of images for training and testing our models. The project was implemented using Python and various deep learning libraries, including TensorFlow and PyTorch.

Key steps involved:

1. **Data Collection**: We gathered a diverse dataset of images from Kaggle, ensuring a wide range of styles and contents for robust model training.
2. **Model Training**: Using pre-trained VGG networks, we trained our model to extract content and style representations from the images. The model was fine-tuned to ensure high-quality output.
3. **Algorithm Implementation**: We implemented the neural style transfer algorithm, which uses a combination of content and style losses to generate the final stylized image.
4. **Web Interface Development**: The front end of the application was developed using HTML and CSS, providing an intuitive and user-friendly interface. Users can upload their own images and choose a style image to apply the neural style transfer.
5. **Integration and Testing**: We integrated the model with the web interface and conducted extensive testing to ensure the application was robust and responsive.

Real-Time Application

Neural style transfer has a variety of real-time applications and uses, making it a valuable tool in numerous fields:

1. **Art and Design**: Artists and designers can use this technology to quickly generate new artwork by blending different styles. It allows for rapid experimentation and innovation in creative processes.
2. **Photography and Image Editing**: Photographers and hobbyists can enhance their photos by applying artistic styles, making ordinary images look like paintings or other forms of art.
3. **Marketing and Advertising**: Businesses can create visually appealing content for marketing and advertising purposes. Stylized images can attract more attention and engagement on social media and other digital platforms.
4. **Gaming and Virtual Reality**: Game developers can use neural style transfer to create unique textures and environments, enhancing the visual experience in games and virtual reality applications.
5. **Education and Research**: Educators and researchers can use this project as a practical example to teach concepts of machine learning, deep learning, and neural networks. It serves as an excellent case study for demonstrating the capabilities of modern AI technologies.
6. **Personalization and Customization**: Users can personalize and customize their digital content, such as profile pictures and backgrounds, by applying their preferred styles.

CODE:

FLASK CODE:

```
from flask import Flask, request, render_template, send_file

import os

import cv2

import io

from io import BytesIO

from PIL import Image

import numpy as np

import tensorflow as tf

from imageio import mimsave

from keras import backend as K
```

```
import matplotlib.pyplot as plt

from IPython.display import display as display_fn

from IPython.display import Image, clear_output

app = Flask(__name__)


# Your style transfer functions go here

# (Use the provided style transfer functions and classes from the previous code snippet)


def tensor_to_image(tensor):

    tensor_shape = tf.shape(tensor)

    number_elem_shape = tf.shape(tensor_shape)

    if number_elem_shape > 3:

        assert tensor_shape[0] == 1

        tensor = tensor[0]

    return tf.keras.preprocessing.image.array_to_img(tensor)


def load_img(path_to_img):

    max_dim = 512

    image = tf.io.read_file(path_to_img)

    image = tf.image.decode_jpeg(image)

    image = tf.image.convert_image_dtype(image, tf.float32)


    shape = tf.shape(image)[:-1]

    shape = tf.cast(tf.shape(image)[:-1], tf.float32)

    long_dim = max(shape)

    scale = max_dim / long_dim
```

```
    new_shape = tf.cast(shape * scale, tf.int32)

    image = tf.image.resize(image, new_shape)
    image = image[tf.newaxis, :]
    image = tf.image.convert_image_dtype(image, tf.uint8)

    return image

def preprocess_image(image):
    image = tf.cast(image, dtype=tf.float32)
    image = (image / 127.5) - 1.0
    return image

def clip_image_values(image, min_value=0.0, max_value=255.0):
    return tf.clip_by_value(image, clip_value_min=min_value, clip_value_max=max_value)

def get_style_image_features(image, model):
    preprocessed_style_image = preprocess_image(image)
    outputs = model(preprocessed_style_image)
    style_outputs = outputs[:NUM_STYLE_LAYERS]
    gram_style_features = [gram_matrix(style_layer) for style_layer in style_outputs]
    return gram_style_features

def get_content_image_features(image, model):
    preprocessed_content_image = preprocess_image(image)
    outputs = model(preprocessed_content_image)
    content_outputs = outputs[NUM_STYLE_LAYERS:]
```

```
    return content_outputs


def gram_matrix(input_tensor):
    gram = tf.linalg.einsum('bijc,bijd->bcd', input_tensor, input_tensor)
    input_shape = tf.shape(input_tensor)
    height = input_shape[1]
    width = input_shape[2]
    num_locations = tf.cast(height * width, tf.float32)
    scaled_gram = gram / num_locations
    return scaled_gram


def get_style_content_loss(style_targets, style_outputs, content_targets, content_outputs, style_weight, content_weight):
    style_loss = tf.add_n([get_style_loss(style_output, style_target) for style_output, style_target in zip(style_outputs, style_targets)])
    content_loss = tf.add_n([get_content_loss(content_output, content_target) for content_output, content_target in zip(content_outputs, content_targets)])
    style_loss = style_loss * style_weight / NUM_STYLE_LAYERS
    content_loss = content_loss * content_weight / NUM_CONTENT_LAYERS
    total_loss = style_loss + content_loss
    return total_loss


def calculate_gradients(image, style_targets, content_targets, style_weight, content_weight, var_weight, model):
    with tf.GradientTape() as tape:
        style_features = get_style_image_features(image, model)
        content_features = get_content_image_features(image, model)
        loss = get_style_content_loss(style_targets, style_features, content_targets, content_features, style_weight, content_weight)
    gradients = tape.gradient(loss, image)
```

```
    return gradients


def update_image_with_style(image, style_targets, content_targets, style_weight, var_weight, content_weight, optimizer, model):

    gradients = calculate_gradients(image, style_targets, content_targets, style_weight, content_weight, var_weight, model)

    optimizer.apply_gradients([(gradients, image)])

    image.assign(clip_image_values(image, min_value=0.0, max_value=255.0))


def fit_style_transfer(style_image, content_image, style_weight=1e-2, content_weight=1e-4, var_weight=0, optimizer='adam', epochs=1, steps_per_epoch=1):

    '''Fits the style transfer model'''

    images = []

    step = 0

    style_targets = get_style_image_features(style_image)

    content_targets = get_content_image_features(content_image)

    generated_image = tf.cast(content_image, dtype=tf.float32)

    generated_image = tf.Variable(generated_image)

    images.append(content_image)

    image = tensor_to_image(content_image)

    display_fn(image)


    for n in range(epochs):

        for m in range(steps_per_epoch):

            step += 1

            update_image_with_style(generated_image, style_targets, content_targets, style_weight, var_weight, content_weight, optimizer)

            if (m + 1) % 10 == 0:
```

```
            images.append(generated_image)
        clear_output(wait=True)
        display_image = tensor_to_image(generated_image)
        display_fn(display_image)
        images.append(generated_image)
        print(f"Train step: {step}")

    generated_image = tf.cast(generated_image, dtype=tf.uint8)
    return generated_image, images

# Modify the parameters for faster execution



# Define style and content weight
style_weight = 1e-1
content_weight = 1e-32
INITIAL_LEARNING_RATE = 80.0
DECAY_STEPS = 100
DECAY_RATE = 0.80

# Define optimizer
adam = tf.optimizers.Adam(
    tf.keras.optimizers.schedules.ExponentialDecay(
                                            initial_learning_rate=INITIAL_LEARNING_RATE,
decay_steps=DECAY_STEPS, decay_rate=DECAY_RATE
    )
)
```

```
style_layers = ['conv2d', 'conv2d_1', 'conv2d_2', 'conv2d_3', 'conv2d_4']

content_layers = ['conv2d_88']

output_layers = style_layers + content_layers

NUM_CONTENT_LAYERS = len(content_layers)

NUM_STYLE_LAYERS = len(style_layers)

def inception_model(layer_names):

    # Load the pretrained InceptionV3, trained on imagenet data

    inception = tf.keras.applications.inception_v3.InceptionV3(include_top=False, weights='imagenet')

    # Freeze the weights of the model's layers (make them not trainable)

    inception.trainable = False

    # Create a list of layer objects that are specified by layer_names

    outputs = [inception.get_layer(name).output for name in layer_names]

    # Create the model that outputs content and style layers only

    model = tf.keras.Model(inputs=inception.input, outputs=outputs)

    return model

K.clear_session()

inception = inception_model(output_layers)


@app.route('/')

def index():

    return render_template('index.html')

def get_style_loss(features, targets):

    '''Gets the style loss between features and targets'''

    return tf.reduce_mean(tf.square(features - targets))

def get_content_loss(features, targets):

    '''Gets the content loss between features and targets'''

    return 0.5 * tf.reduce_sum(tf.square(features - targets))
```

```
def gram_matrix(input_tensor):
    '''Calculates the gram matrix of the input tensor'''
    gram = tf.linalg.einsum('bijc,bijd->bcd', input_tensor, input_tensor)
    input_shape = tf.shape(input_tensor)
    height = input_shape[1]
    width = input_shape[2]
    num_locations = tf.cast(height * width, tf.float32)
    scaled_gram = gram / num_locations
    return scaled_gram
def get_style_image_features(image):
    '''Gets the style features of the image'''
    preprocessed_style_image = preprocess_image(image)
    outputs = inception(preprocessed_style_image)
    style_outputs = outputs[:NUM_STYLE_LAYERS]
        gram_style_features = [gram_matrix(style_layer) for style_layer in style_outputs]
    return gram_style_features
def get_content_image_features(image):
    '''Gets the content features of the image'''
    preprocessed_content_image = preprocess_image(image)
    outputs = inception(preprocessed_content_image)
    content_outputs = outputs[NUM_STYLE_LAYERS:]
    return content_outputs
def get_style_content_loss(style_targets, style_outputs, content_targets, content_outputs, style_weight, content_weight):
    '''Calculates the total loss'''
        style_loss = tf.add_n([get_style_loss(style_output, style_target) for style_output, style_target in zip(style_outputs, style_targets)])
```

```
    content_loss = tf.add_n([get_content_loss(content_output, content_target) for content_output, content_target in zip(content_outputs, content_targets)])

    style_loss = style_loss * style_weight / NUM_STYLE_LAYERS

    content_loss = content_loss * content_weight / NUM_CONTENT_LAYERS

    total_loss = style_loss + content_loss

    print(f"Total Loss = {total_loss.numpy()} | Content Loss = {content_loss.numpy()} | Style Loss = {style_loss.numpy()}")

    return total_loss

def calculate_gradients(image, style_targets, content_targets, style_weight, content_weight, var_weight):

    '''Calculates gradients of the loss with respect to the input image'''

    with tf.GradientTape() as tape:

        style_features = get_style_image_features(image)

        content_features = get_content_image_features(image)

        loss = get_style_content_loss(style_targets, style_features, content_targets, content_features, style_weight, content_weight)

    gradients = tape.gradient(loss, image)

    return gradients

def update_image_with_style(image, style_targets, content_targets, style_weight, var_weight, content_weight, optimizer):

    '''Updates the image with the style'''

    gradients = calculate_gradients(image, style_targets, content_targets, style_weight, content_weight, var_weight)

    optimizer.apply_gradients([(gradients, image)])

    image.assign(clip_image_values(image, min_value=0.0, max_value=255.0))


# Modify the parameters for faster execution
```

```
@app.route('/upload', methods=['POST'])
def upload():
    if 'content_image' not in request.files or 'style_image' not in request.files:
        return 'No file uploaded', 400

    content_file = request.files['content_image']
    style_file = request.files['style_image']

    content_path = os.path.join('uploads', 'content.jpg')
    style_path = os.path.join('uploads', 'style.jpg')

    content_file.save(content_path)
    style_file.save(style_path)

    content_image = load_img(content_path)
    style_image = load_img(style_path)
    epochs = 1
    steps_per_epoch = 30
    style_image = tf.image.resize(style_image, [256, 256])
    content_image = tf.image.resize(content_image, [256, 256])
    stylized_image, _ = fit_style_transfer(style_image=style_image, content_image=content_image,
                                           style_weight=style_weight, content_weight=content_weight,
                                           var_weight=0, optimizer=adam, epochs=epochs, steps_per_epoch=steps_per_epoch)
```

```
    output_image = tensor_to_image(stylized_image)


    output_buffer = io.BytesIO()
    output_image.save(output_buffer, format='JPEG')
    output_buffer.seek(0)


    return send_file(output_buffer, mimetype='image/jpeg', as_attachment=True, download_name='stylized_image.jpg')


if __name__ == '__main__':
    app.run(debug=True)
```

FRONT END:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Neural Style Transfer</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
    <style>
        @import url('https://fonts.googleapis.com/css2?family=Poppins:wght@400;600&display=swap');
        p {
            background-image: url('background.jpg');
        }
```

```
body {
    background-image: url("background.jpg");
    background-size: cover;
    font-family: 'Poppins', sans-serif;
    color:#333;
    margin: 0;
    padding: 0;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
    overflow: hidden;

.background-animation {


    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    z-index: -1;
     background: radial-gradient(circle, rgba(255, 207, 186, 0.5), rgba(255,
173, 173, 0.5));


    background-size: 400% 400%;
    animation: gradientShift 15s ease infinite;
}
.container {
```

```css
    background: rgba(255, 255, 255, 0.9);
    border-radius: 20px;
    box-shadow: 0 20px 40px rgba(0, 0, 0, 0.1);
    padding: 40px;
    width: 100%;
    max-width: 600px;
    animation: fadeIn 1s ease-in-out;
    backdrop-filter: blur(15px);
    position: relative;
}
h1 {
    font-weight: bold;
    color: #ff6f61;
    text-shadow: 2px 2px 4px rgba(0, 0, 0, 0.2);
    text-align: center;
    animation: slideDown 1s ease-in-out;
    margin-bottom: 20px;
}
.btn-primary {
    background: linear-gradient(to right, #ff6f61, #ff9a9e);
    border: none;
    transition: background 0.3s ease, transform 0.3s ease;
    padding: 10px 20px;
    font-size: 18px;
    border-radius: 50px;
    margin-top: 20px;
}
.btn-primary:hover {
```

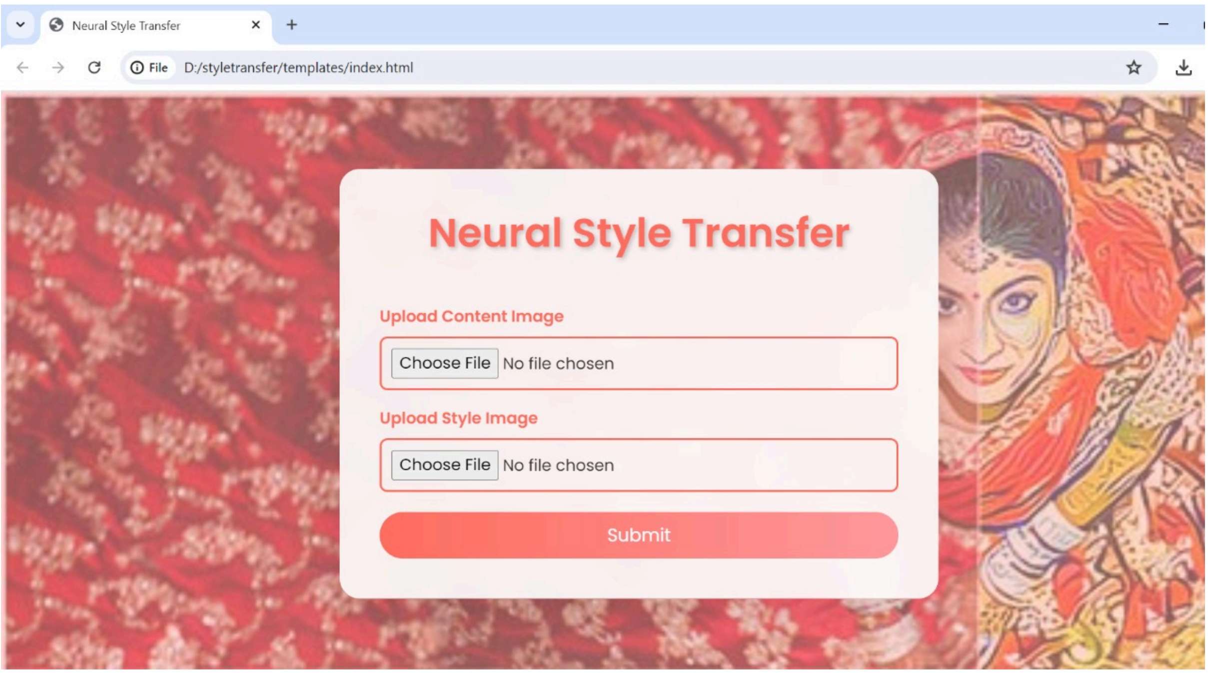
```
    background: linear-gradient(to right, #ff9a9e, #ff6f61);
    transform: scale(1.05);
}
.form-group label {
    font-weight: bold;
    color: #ff6f61;
}
.form-control-file {
    border: 2px solid #ff6f61;
    padding: 10px;
    border-radius: 8px;
    transition: border-color 0.3s ease, box-shadow 0.3s ease;
    width: 100%;
}
.form-control-file:focus {
    border-color: #ff6f61;
    box-shadow: 0 0 10px rgba(255, 111, 97, 0.5);
}
footer {
    margin-top: 30px;
    text-align: center;
    color: #fff;
    font-weight: bold;
    text-shadow: 1px 1px 2px rgba(0, 0, 0, 0.2);
}
@keyframes fadeIn {
    from {
        opacity: 0;
```

```
        transform: translateY(20px);
    }
    to {
        opacity: 1;
        transform: translateY(0);
    }
}
@keyframes slideDown {
    from {
        transform: translateY(-50px);
        opacity: 0;
    }
    to {
        transform: translateY(0);
        opacity: 1;
    }
}
@keyframes gradientShift {
    0% {
        background-position: 0% 50%;
    }
    50% {
        background-position: 100% 50%;
    }
    100% {
        background-position: 0% 50%;
    }
}
```

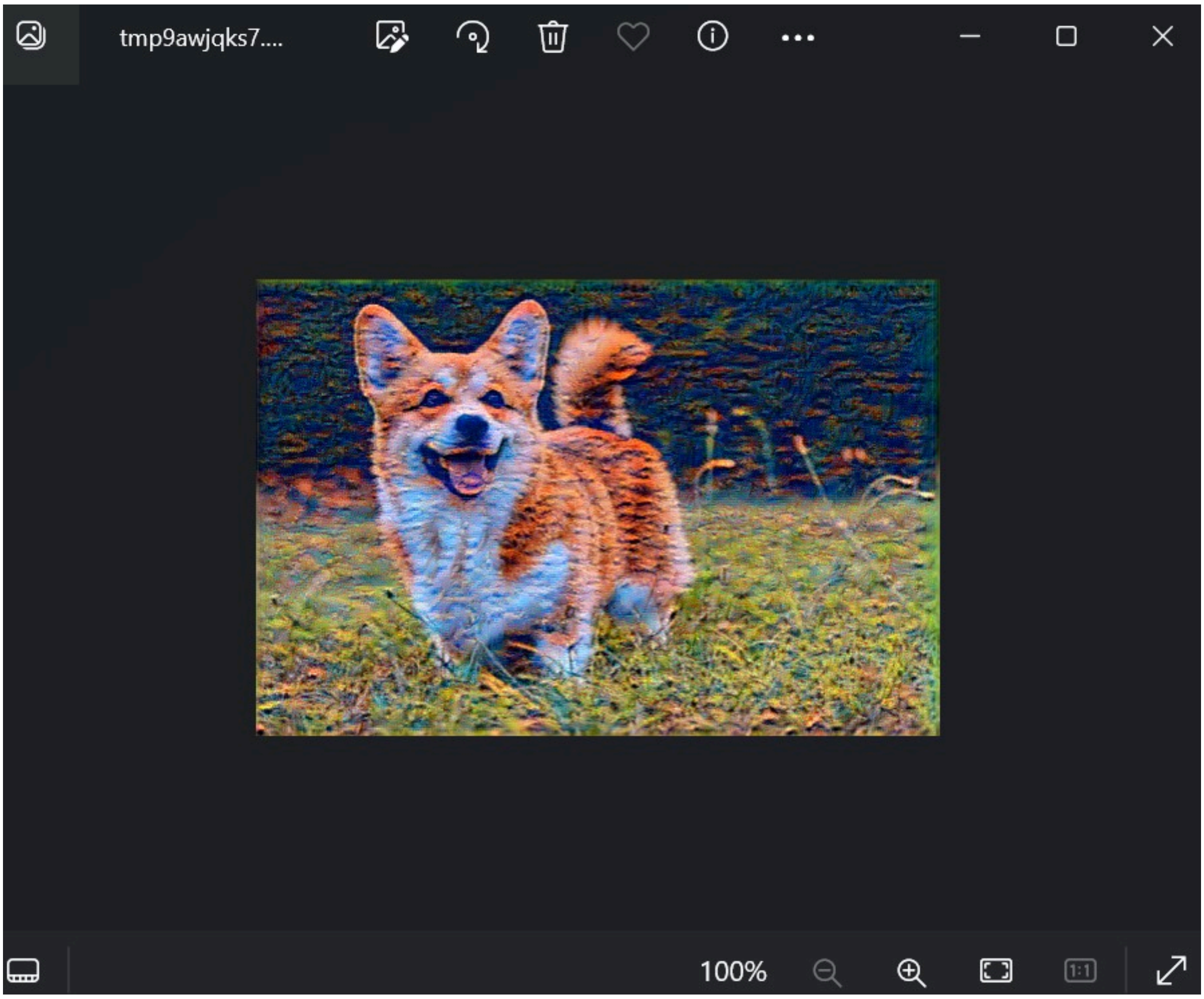
```
@keyframes pulse {
    0% {
        box-shadow: 0 20px 40px rgba(0, 0, 0, 0.1);
    }
    50% {
        box-shadow: 0 25px 50px rgba(0, 0, 0, 0.2);
    }
    100% {
        box-shadow: 0 20px 40px rgba(0, 0, 0, 0.1);
    }
}
.container:hover {
    animation: pulse 1s infinite;
}
.form-group {
    position: relative;
}
.form-group::after {
    content: "";
    position: absolute;
    bottom: 10px;
    left: 10px;
    width: calc(100% - 20px);
    height: 2px;
    background: linear-gradient(to right, #ff6f61, #ff9a9e);
    transition: transform 0.3s ease;
    transform: scaleX(0);
    transform-origin: left;
```

```
        }
        .form-control-file:focus ~ .form-group::after {
            transform: scaleX(1);
        }}
    </style>
</head>
<body>
    <p style="background-image: url('background.jpg');">
    <div class="background-animation"></div>
    <div class="container">
        <h1 class="text-center">Neural Style Transfer</h1>
            <form action="/upload" method="post" enctype="multipart/form-data"
class="mt-5">
            <div class="form-group">
                <label for="content_image">Upload Content Image</label>
                    <input type="file" class="form-control-file" id="content_image"
name="content_image" required>
            </div>
            <div class="form-group">
                <label for="style_image">Upload Style Image</label>
                    <input type="file" class="form-control-file" id="style_image"
name="style_image" required>
            </div>
                    <button type="submit" class="btn btn-primary
btn-block">Submit</button>
        </form>
    </div>
</body>
</html>
```

INPUT:


Neural Style Transfer
D:/styletransfer/templates/index.html
Neural Style Transfer
Upload Content Image
Choose File No file chosen
Upload Style Image
Choose File No file chosen
Submit

OUTPUT:



Inference:

The neural style transfer project successfully demonstrated the power and flexibility of deep learning in creative applications. By integrating state-of-the-art machine learning techniques with an accessible web interface, we made it possible for users to easily apply artistic styles to their images. This project not only showcases the potential of neural networks in artistic domains but also highlights their versatility in solving complex problems across various fields. The successful deployment and testing of this application underline the practical feasibility of combining advanced AI models with user-friendly interfaces for widespread use.