[REPORT]
CURVETOPIA: A Journey into the World of Curves

1. Introduction

Curves are a fundamental element in digital art and design, representing fluidity and elegance. CURVETOPIA seeks to explore and enhance the representation of curves by employing advanced algorithms to regularize, identify symmetry, and complete curves. This project offers a unique approach to transforming raster line art into a set of connected cubic Bezier curves, bringing precision and aesthetic appeal to digital artworks.

2. Objectives

The primary objective of CURVETOPIA is to develop a comprehensive pipeline that processes raster images of line art and outputs them as a series of cubic Bezier curves. The project addresses three main challenges:

[Regularizing Curves]
[Exploring Symmetry in Curves]
[Completing Incomplete Curves]

3. Methodology

Regularizing Curves

This phase involves identifying specific shapes within a set of curves and regularizing them based on predefined geometric properties. The challenge includes detecting and refining straight lines, circles, ellipses, rectangles, and other regular polygons.

Exploring Symmetry in Curves

Symmetry is a critical aspect of aesthetics in art. This challenge involves detecting reflective symmetries in closed shapes and fitting them with identical Bezier curves, thereby enhancing their visual harmony.

Completing Incomplete Curves

In digital art, curves often get interrupted or occluded. This phase focuses on completing such curves by analyzing their geometric properties and filling in the gaps to create smooth, continuous shapes.

4. Problem Statement

Transforming raster line art images into a set of connected cubic Bezier curves presents several computational challenges:

- How can we effectively regularize a diverse set of curves into standard geometric shapes?
- What methods can be used to detect and analyze symmetry in curves?
- How can we reconstruct missing parts of incomplete curves to restore their original form?

5. Implementation Details

Data Loading and Preprocessing :

The data is loaded from CSV files, where each file contains sets of curves represented by their x and y coordinates. The data is then preprocessed to prepare it for regularization, symmetry analysis, and curve completion.

Curve Regularization using K-Means Clustering :

K-Means clustering is employed to group similar curves based on their features, such as distances and angles, and to regularize them into specific geometric shapes.

Symmetry Detection using Convolutional Neural Networks (CNNs) :

A CNN model is trained to identify symmetrical patterns in curves. The model processes 2D representations of curves and outputs a binary classification indicating whether a curve is symmetrical.

Curve Completion using Generative Adversarial Networks (GANs) :
GANs are utilized to complete incomplete curves. The generator model attempts to reconstruct the missing parts of a curve, while the discriminator evaluates the realism of the generated curve.

Autoencoder for Curve Reconstruction :
An autoencoder model is used to reconstruct curves from their incomplete versions. The model is trained to minimize the difference between the input incomplete curve and the output reconstructed curve.

6. Experiments and Results

Data Visualization

Curves are visualized using matplotlib to inspect their structure and characteristics before and after regularization, symmetry detection, and completion.

Feature Extraction and Regularization

The effectiveness of K-Means clustering in regularizing curves is evaluated through feature extraction techniques, where each curve's geometric properties are analyzed.

Symmetry Analysis

The CNN model's performance in detecting symmetry is assessed by testing it on various closed shapes. The results are discussed in terms of accuracy and symmetry transformation.

Curve Completion Outcomes

The GAN model's ability to complete curves is demonstrated through examples of incomplete curves before and after reconstruction. The quality and smoothness of the completed curves are analyzed.

7. Conclusion

CURVETOPIA successfully demonstrates a novel approach to transforming line art into structured, regularized curves. The project addresses significant challenges in curve regularization, symmetry detection, and curve completion, providing valuable insights and tools for digital artists and designers.

[MAIN]

BASICALLY WE DO THESE THINGS HERE

1. Symmetry Detection Model:
Convolutional Layers (Conv1D):
The model uses a series of 1D convolutional layers to extract features from input curves. The input shape (374, 2) suggests that each curve is represented by 374 points in 2D space.

Conv1D(32, 3, activation='relu', input_shape=(374, 2)): First convolutional layer with 32 filters, a kernel size of 3, and ReLU activation.

MaxPooling1D(2): Reduces the spatial dimensions by taking the maximum value in every window of size 2.

The same pattern is repeated with more filters (64 and 128) in subsequent layers.

Flatten: Converts the 3D tensor output from the convolutions into a 1D vector.

Dense Layers: Fully connected layers to interpret the features extracted by the convolutional layers.

Output Layer (Dense): Uses a sigmoid activation function for binary classification (symmetrical or not).

Model Compilation:
The model is compiled with the Adam optimizer, binary cross-entropy loss (since it's a binary classification), and accuracy as a metric.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten, Dense

def build_symmetry_detection_model():
   model = Sequential([
       Conv1D(32, 3, activation='relu', input_shape=(374, 2)),
       MaxPooling1D(2),
       Conv1D(64, 3, activation='relu'),
       MaxPooling1D(2),
       Conv1D(128, 3, activation='relu'),
       Flatten(),
       Dense(64, activation='relu'),
       Dense(1, activation='sigmoid')  # Symmetrical (1) or not (0)
   ])
   model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
   return model
```

2. Training Function:

train_symmetry_detection_model: This function trains the symmetry detection model using the provided curves and labels. It also allows specifying the number of epochs and batch size.

```
def train_symmetry_detection_model(model, curves, labels, epochs=10,
batch_size=32):
   model.fit(curves, labels, epochs=epochs, batch_size=batch_size,
validation_split=0.2)
```

Explanation:

train_symmetry_detection_model: This function trains the symmetry detection model.

model.fit: Fits the model to the provided data (curves and labels). It uses 20% of the data for validation (validation_split=0.2) and runs for a specified number of epochs and batch size.

3. Curve Completion with Autoencoder:

Autoencoder Structure:

- Encoder: The encoder uses convolutional layers and max pooling to compress the input curve into a latent space representation.
- Latent Space: The compressed representation of the input data.
- Decoder: The decoder upsamples the latent space representation back to the original dimensions using convolutional layers and upsampling operations.

Autoencoder Compilation:

The autoencoder is compiled with the Adam optimizer and mean squared error (MSE) loss, which is typical for autoencoders.

```
from tensorflow.keras import layers, models

def build_autoencoder(input_shape):
   input_img = layers.Input(shape=input_shape)

   # Encoder
   x = layers.Conv1D(32, 3, activation='relu', padding='same')(input_img)
   x = layers.MaxPooling1D(2, padding='same')(x)
   x = layers.Conv1D(64, 3, activation='relu', padding='same')(x)
   x = layers.MaxPooling1D(2, padding='same')(x)
   x = layers.Conv1D(128, 3, activation='relu', padding='same')(x)
   x = layers.MaxPooling1D(2, padding='same')(x)
   x = layers.Conv1D(256, 3, activation='relu', padding='same')(x)
   x = layers.MaxPooling1D(2, padding='same')(x)

   # Flatten and Dense layer
   x = layers.Flatten()(x)
   encoded = layers.Dense(256, activation='relu')(x)

   # Decoder
   x = layers.Dense(256 * (input_shape[0] // 16),
activation='relu')(encoded)
   x = layers.Reshape((input_shape[0] // 16, 256))(x)

   x = layers.UpSampling1D(2)(x)  # Upsample to (x // 8, 256)
   x = layers.Conv1D(256, 3, activation='relu', padding='same')(x)
   x = layers.UpSampling1D(2)(x)  # Upsample to (x // 4, 128)
   x = layers.Conv1D(128, 3, activation='relu', padding='same')(x)
   x = layers.UpSampling1D(2)(x)  # Upsample to (x // 2, 64)
   x = layers.Conv1D(64, 3, activation='relu', padding='same')(x)
   x = layers.UpSampling1D(2)(x)  # Upsample to (x, 32)
   decoded = layers.Conv1D(input_shape[1], 3, activation='sigmoid',
padding='same')(x)

   autoencoder = models.Model(input_img, decoded)
   autoencoder.compile(optimizer='adam', loss='mse')
   return autoencoder
```

Explanation:

Autoencoder Architecture:

Encoder:

Uses convolutional layers (Conv1D) and pooling layers (MaxPooling1D) to compress the input data into a smaller representation (latent space).

Latent Space:

The Flatten and Dense layers compress the features into a dense vector representation (encoded).

Decoder:

Upsamples the latent vector back to the original size using UpSampling1D and Conv1D layers, reconstructing the original input.

Model Compilation:

The autoencoder is compiled with the Adam optimizer and mean squared error (mse) loss, which measures the difference between the input and the reconstructed output.

4. Shape Classification:

classify_shape: This function classifies a polyline (a series of connected points) as a line, circle, ellipse, or other shapes based on geometric fitting functions.

is_line, is_circle, fit_line, fit_circle:

These functions determine if a polyline is best approximated by a line or circle. They use linear regression (np.polyfit) and least squares (np.linalg.lstsq) to fit the shapes to the points.

Ellipse Fitting: The code has a method to check if the polyline is an ellipse by fitting it to an elliptical shape.

```
import numpy as np

def classify_shape(polyline):
    if is_line(polyline):
        return 'line'
    elif is_circle(polyline):
        return 'circle'
    elif is_ellipse(polyline):
        return 'ellipse'
    else:
        return 'other'

def is_line(polyline):
    return np.allclose(np.polyfit(polyline[:, 0], polyline[:, 1], 1), [0])

def is_circle(polyline):
    center, radius = fit_circle(polyline)
```

```
    return np.allclose(np.linalg.norm(polyline - center, axis=1), radius)

def is_ellipse(polyline):
    # Implement ellipse fitting and checking
    pass

def fit_line(polyline):
    return np.polyfit(polyline[:, 0], polyline[:, 1], 1)

def fit_circle(polyline):
    x = polyline[:, 0]
    y = polyline[:, 1]
    A = np.array([x, y, np.ones(len(x))]).T
    b = x**2 + y**2
    a = np.linalg.lstsq(A, b, rcond=None)[0]
    center = a[:2] / 2
    radius = np.sqrt(a[2] + center.dot(center))
    return center, radius
```

Explanation:

classify_shape: Determines if the input polyline is a line, circle, ellipse, or other shape.

is_line: Checks if the polyline can be fit to a straight line using linear regression (np.polyfit).

is_circle: Checks if the polyline can be fit to a circle by computing the center and radius using least squares.

fit_circle: Fits the polyline to a circle and returns the center and radius.

5. **Curve Regularization:**

   regularize_curves: This function applies the shape classification to a list of polylines and regularizes (fits) them to the nearest geometric shape (line, circle, ellipse). If the shape doesn't match any predefined type, it keeps the original polyline.

```
def regularize_curves(polylines):
    regularized_shapes = []
    for polyline in polylines:
        shape_type = classify_shape(polyline)
        if shape_type == 'line':
            regularized_shape = fit_line(polyline)
        elif shape_type == 'circle':
            center, radius = fit_circle(polyline)
            regularized_shape = (center, radius)
        elif shape_type == 'ellipse':
            params = fit_ellipse(polyline)
            regularized_shape = params
        else:
            regularized_shape = polyline  # For other shapes, keep original
        regularized_shapes.append(regularized_shape)
    return regularized_shapes
```

6. Example Usage:
CSV File Reading: The code reads a CSV file containing polylines.

```
csv_path = "/content/drive/MyDrive/isolated.csv"
paths_XYs = read_csv(csv_path)
regularized_shapes = regularize_curves(paths_XYs[0])
print(regularized_shapes)
plot(paths_XYs)
```

Explanation:

classify_shape: Determines if the input polyline is a line, circle, ellipse, or other shape.

is_line: Checks if the polyline can be fit to a straight line using linear regression (np.polyfit).

is_circle: Checks if the polyline can be fit to a circle by computing the center and radius using least squares.

fit_circle: Fits the polyline to a circle and returns the center and radius.

Regularization and Plotting: It regularizes the curves and likely plots them for visualization (though the plotting function isn't fully visible).

This code is likely used for tasks like recognizing and processing shapes within a set of curves, such as in image processing, computer vision, or geometric data analysis.