import numpy as np

from PIL import Image

import base64

import tensorflow as tf

import matplotlib.pyplot as plt

from tensorflow.keras.layers import Conv2D

# Function to display a matrix

def display\_matrix(matrix):

    for row in matrix:

        print(" ".join(map(str, row)))

# Function to convert a float to binary16 explicit format

def float\_to\_binary16\_explicit(value):

    if value == 0.0:

        return '0' \* 16  # Special case for zero

    # Get the sign, exponent, and mantissa

    sign = 0 if value >= 0 else 1

    value = abs(value)

    exponent = 0

    while value < 0.5:  # Normalize value to [0.5, 1) range

        value \*= 2

        exponent -= 1

    while value >= 1:  # Normalize value to [0.5, 1)

        value /= 2

        exponent += 1

    mantissa = int(value \* (2 \*\* 10))  # 10 bits for the mantissa

    # Convert to IEEE 754 binary16 format

    exponent\_bits = exponent + 15  # Add bias (15) for binary16

    binary16\_bits = (sign << 15) | (exponent\_bits << 10) | mantissa

    return bin(binary16\_bits)[2:].zfill(16)

# Function to convert binary string to decimal

def binary\_to\_decimal(binary\_str):

    decimal = 0

    power = 0

    for bit in reversed(binary\_str):

        if bit == '1':

            decimal += 2 \*\* power

        power += 1

    return decimal

# Function to find the maximum exponent in a matrix

def max\_expo(matrix):

    max\_exponent = 0

    for value in matrix.flatten():

        exponent = 0

        normalized\_value = abs(value)

        while normalized\_value < 0.5:

            normalized\_value \*= 2

            exponent -= 1

        while normalized\_value >= 1:

            normalized\_value /= 2

            exponent += 1

        max\_exponent = max(max\_exponent, exponent)

    return max\_exponent

# Function to convert a float to binary16 with exponent alignment

def float\_to\_binary16\_explicit\_updated(value, diff, max\_exponent):

    if value == 0.0:

        return '0' \* 16  # Special case for zero

    # Get the sign, exponent, and mantissa

    sign = 0 if value >= 0 else 1

    value = abs(value)

    while value < 0.5:  # Normalize value to [0.5, 1) range

        value \*= 2

    while value >= 1:  # Normalize value to [0.5, 1)

        value /= 2

    # Apply exponent difference to mantissa

    mantissa = int(value \* (2 \*\* 10))  # 10 bits for the mantissa

    if diff > 0:

        mantissa >>= abs(diff)

    elif diff < 0:

        mantissa <<= abs(diff)

    # Convert to IEEE 754 binary16 format

    exponent\_bits = max\_exponent + 15  # Add bias (15) for binary16

    binary16\_bits = (sign << 15) | (exponent\_bits << 10) | mantissa

    return bin(binary16\_bits)[2:].zfill(16)

# Get the image path from the user

image\_path = input("Enter the path to the Image: ")

image\_height = int(input("Enter the height of the Image: "))

image\_width = int(input("Enter the width of the Image: "))

# Convert the image to grayscale and resize it

image = Image.open(image\_path).convert("L")

image = image.resize((image\_width, image\_height))  # Resize the image to match the matrix dimensions

matrix = np.array(image, dtype=float)

# Display the extracted pixel values

print("Extracted pixel values from the Image:")

display\_matrix(matrix)

print()

# Scale the pixels by dividing by 255

scaled\_matrix = matrix / 255

# Display the scaled pixel values

print("Scaled pixel values (divided by 255):")

display\_matrix(scaled\_matrix)

print()

# Find the maximum exponent among all the exponents in the scaled matrix

max\_exponent = max\_expo(scaled\_matrix)

print("\nMaximum exponent among all elements in scaled Image:", max\_exponent)

print()

# Create a matrix to store the differences between max\_exponent and each element's exponent

exponent\_difference\_matrix = np.zeros((image\_height, image\_width), dtype=int)

# Calculate the exponent difference for each element

for i in range(image\_height):

    for j in range(image\_width):

        value = scaled\_matrix[i, j]

        exponent = 0

        normalized\_value = abs(value)

        while normalized\_value < 0.5:

            normalized\_value \*= 2

            exponent -= 1

        while normalized\_value >= 1:

            normalized\_value /= 2

            exponent += 1

        exponent\_difference\_matrix[i, j] = max\_exponent - exponent

# Display the exponent difference matrix

print("Exponent difference matrix:")

display\_matrix(exponent\_difference\_matrix)

print()

# Create a matrix with updated binary16 representation using exponent difference

binary16\_updated\_matrix = np.vectorize(float\_to\_binary16\_explicit\_updated)(scaled\_matrix, exponent\_difference\_matrix, max\_exponent)

# Display the updated IEEE 754 binary16 representation of the matrix elements

print("Binary16 representation of the Image matrix with aligned mantissa:")

display\_matrix(binary16\_updated\_matrix)

print()

# Convert the BFP representation back to the original decimal form

reconstructed\_matrix = np.vectorize(lambda x: int(x[6:], 2) / (2 \*\* 10))(binary16\_updated\_matrix)

# Display the reconstructed decimal values

print("Reconstructed decimal values from binary16 representation:")

display\_matrix(reconstructed\_matrix)

print()

# Define kernel options

kernel\_options = {

    1: np.array([[0.1111, 0.1111, 0.1111], [0.1111, 0.1111, 0.1111], [0.1111, 0.1111, 0.1111]]),  # 3x3 all ones

    2: np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]]),  # 3x3 edge detection

    3: np.array([[0.0625, 0.125, 0.0625], [0.125, 0.25, 0.125], [0.0625, 0.125, 0.0625]]),

    4: np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]]),

    5: np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]]),

    # Add more kernel options here

}

# Get the kernel option from the user

print("Available Kernel Options:")

for option, kernel in kernel\_options.items():

    print(f"{option}:")

    print(kernel)

    print()

kernel\_option = int(input("Choose a kernel option (1, 2, ...): "))

# Select the chosen kernel

Filter = kernel\_options.get(kernel\_option)

imageMatrix\_normal = 1 - scaled\_matrix

imageMatrix\_normal = imageMatrix\_normal.reshape(1, image\_height, image\_width, 1)

imageMatrix\_bfp = 1 - reconstructed\_matrix

imageMatrix\_bfp = imageMatrix\_bfp.reshape(1, image\_height, image\_width, 1)

# Prepare the filter weights

filterWeights = Filter.reshape(Filter.shape[0], Filter.shape[1], 1, 1)

# Define input shape for the Conv2D layer

input\_shape = (1, image\_height, image\_width, 1)

filters = 1

kernelSize = Filter.shape

# Create and apply the Conv2D layer

featureMap\_normal = Conv2D(filters=filters, kernel\_size=kernelSize, input\_shape=input\_shape, use\_bias=False,

                    activation='relu', weights=[filterWeights], padding='valid',

                    strides=(1, 1))(imageMatrix\_normal)

print('Feature Map shape(normal): ', featureMap\_normal.shape)

print()

# Convert the feature map to an image

featureMap\_normal = featureMap\_normal.numpy()

output\_image\_normal = featureMap\_normal[0, :, :, 0]

output\_image\_normal = (1 - output\_image\_normal) \* 255  # Convert back to original scale

C = output\_image\_normal

output\_image\_normal = output\_image\_normal.astype(np.uint8)

# Display the output image

output\_image\_pil\_normal = Image.fromarray(output\_image\_normal)

display(output\_image\_pil\_normal)

# Create and apply the Conv2D layer

featureMap\_bfp = Conv2D(filters=filters, kernel\_size=kernelSize, input\_shape=input\_shape, use\_bias=False,

                    activation='relu', weights=[filterWeights], padding='valid',

                    strides=(1, 1))(imageMatrix\_bfp)

print('Feature Map shape(bfp): ', featureMap\_bfp.shape)

print()

# Convert the feature map to an image

featureMap\_bfp = featureMap\_bfp.numpy()

output\_image\_bfp = featureMap\_bfp[0, :, :, 0]

output\_image\_bfp = (1 - output\_image\_bfp) \* 255  # Convert back to original scale

D = output\_image\_bfp

output\_image\_bfp = output\_image\_bfp.astype(np.uint8)

# Display the output image

output\_image\_pil\_bfp = Image.fromarray(output\_image\_bfp)

display(output\_image\_pil\_bfp)

# Display pixel values of the output image

print("Pixel values of the output image (bfp):")

print(output\_image\_bfp)

print()

# Define matrices A (true values) and B (approximated values)

A = matrix

B = reconstructed\_matrix \* 255

# Mean Absolute Error (MAE)

def mean\_absolute\_error(true, approx):

    return np.mean(np.abs(true - approx))

# Mean Squared Error (MSE)

def mean\_squared\_error(true, approx):

    return np.mean((true - approx) \*\* 2)

# Root Mean Squared Error (RMSE)

def root\_mean\_squared\_error(true, approx):

    return np.sqrt(mean\_squared\_error(true, approx))

# R-squared (Coefficient of Determination)

def r\_squared(true, approx):

    ss\_total = np.sum((true - np.mean(true)) \*\* 2)

    ss\_res = np.sum((true - approx) \*\* 2)

    return 1 - (ss\_res / ss\_total)

# Maximum Relative Error

def max\_relative\_error(true, approx):

    relative\_errors = np.abs((true - approx) / true)

    return np.max(relative\_errors)

# Calculate and display the errors

mae\_input = mean\_absolute\_error(A,B)

mse\_input = mean\_squared\_error(A, B)

rmse\_input = root\_mean\_squared\_error(A, B)

r2\_input = r\_squared(A, B)

max\_rel\_error\_input = max\_relative\_error(A, B)

print("Error Analysis of Input")

print(f"Mean Absolute Error (MAE): {mae\_input}")

print(f"Mean Squared Error (MSE): {mse\_input}")

print(f"Root Mean Squared Error (RMSE): {rmse\_input}")

print(f"R-squared (R^2): {r2\_input}")

print(f"Maximum Relative Error: {max\_rel\_error\_input}")

print()

# Calculate and display the errors

print("Error Analysis of Output")

mae\_output = mean\_absolute\_error(C, D)

mse\_output = mean\_squared\_error(C, D)

rmse\_output = root\_mean\_squared\_error(C, D)

r2\_output = r\_squared(C, D)

max\_rel\_error\_output = max\_relative\_error(C, D)

print(f"Mean Absolute Error (MAE): {mae\_output}")

print(f"Mean Squared Error (MSE): {mse\_output}")

print(f"Root Mean Squared Error (RMSE): {rmse\_output}")

print(f"R-squared (R^2): {r2\_output}")

print(f"Maximum Relative Error: {max\_rel\_error\_output}")

# Display the images side by side using matplotlib

fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].imshow(image, cmap='gray')

axes[0].set\_title('Input Image')

axes[1].imshow(output\_image\_bfp, cmap='gray')

axes[1].set\_title('BFP-Output Image')

plt.show()