Lab 3

Image Processing with OpenCV

OpenCV (Open Source Computer Vision Library) is an open-source computer vision and image processing library. It provides a comprehensive set of tools and functions for a wide range of tasks related to image and video analysis. OpenCV is widely used in both academia and industry for various computer vision applications.

Capabilities of OpenCV:

- Image Loading and Display: OpenCV allows you to load, display, and save images and videos in various formats. It can handle a wide range of image and video file types.

- Image Processing: OpenCV provides a plethora of image processing functions, including operations like resizing, cropping, filtering, thresholding, and color space conversions. These operations enable you to enhance, clean, and manipulate images.

- Feature Detection and Matching: OpenCV supports various feature detection and matching techniques, including corner detection, edge detection, and keypoint detection, which are crucial for tasks like image stitching and object recognition.

- Object Tracking: OpenCV provides tools for tracking objects in videos. It can be used for applications such as surveillance and motion analysis.

- Object Detection: OpenCV offers pre-trained models and algorithms for object detection tasks, allowing you to identify and locate objects in images or video frames.

- Image Segmentation: You can segment images into distinct regions based on pixel properties, which is useful for applications like medical image analysis and image understanding.

- Machine Learning: OpenCV integrates machine learning algorithms for tasks like image classification, clustering, and regression. It supports the training and deployment of machine learning models.

- Deep Learning Integration: OpenCV seamlessly integrates with popular deep learning frameworks like TensorFlow, PyTorch, and Caffe. This allows you to use deep neural networks for tasks such as image classification, object detection, and image generation.

Synergy with Deep Learning:

- OpenCV and deep learning are often used together to address complex computer vision challenges. Here's how they work in synergy:

- Object Detection: OpenCV can load and use pre-trained deep learning models, such as YOLO (You Only Look Once), to detect and locate objects in images or videos. This combination is widely used in security systems, autonomous vehicles, and more.

- Image Classification: Deep learning models trained for image classification can be seamlessly integrated into OpenCV, enabling you to classify images into various categories or labels.

- Image Generation: Deep learning models, such as Generative Adversarial Networks (GANs), can be employed to generate images. OpenCV can help process and display the generated images.

- Image Preprocessing: Deep learning often requires pre-processing images before feeding them into neural networks. OpenCV's image processing capabilities are invaluable for these tasks.

- Data Augmentation: OpenCV can be used to augment image datasets for deep learning, enhancing model training and generalization.

Step 1: Basic Image Loading and Display

Begin with a simple example of loading an image using OpenCV and displaying it.

```
import cv2

# Load an image

image = cv2.imread('your_image.jpg')

# Display the image

cv2.imshow('Image', image)

cv2.waitKey(0)

cv2.destroyAllWindows()
```

In details, this is an explanation of each part of the code:

- **import cv2:** This line imports the OpenCV library, allowing you to use its **functions** and **classes** for image processing and computer vision tasks.
- **image = cv2.imread('your_image.jpg'):** This line loads an image from a file named 'your_image.jpg' and stores it in the variable 'image'. You should replace 'your_image.jpg' with the actual filename of the image you want to load. The 'cv2.imread' function reads the image file and **creates a NumPy array**, representing the image data. The 'image' variable will hold this NumPy array.
- **cv2.imshow('Image', image):** This line displays the loaded image in a window with the title 'Image'. The 'cv2.imshow' function takes two arguments:
    - The first argument is the **title of the window** where the image will be displayed ('Image' in this case).
    - The second argument is the **image data stored in the 'image' variable**. This function creates a graphical window and shows the image in it.

- **cv2.waitKey(0)**: This line is used to wait for a key event indefinitely or **until a key is pressed.** The 'cv2.waitKey' function takes an **argument that represents the delay in milliseconds.** In this case, '0' means it will wait indefinitely until a key is pressed. **When a key is pressed, the program will continue execution.**
- **cv2.destroyAllWindows():** This line closes all OpenCV windows. It's a good practice to use this line to clean up and close any open windows after you're done with the image display. It's especially important when you have multiple windows open.

Step 2: How OpenCV represents images as NumPy arrays, which allows for various image manipulations.

OpenCV represents images as NumPy arrays, which is a fundamental and efficient way to work with image data. NumPy is a popular Python library for numerical and array operations, and OpenCV leverages it to represent and manipulate images. Here's how OpenCV represents images as NumPy arrays and why it's advantageous:

- **NumPy Array as Image Container:**
  In OpenCV, images are represented as multi-dimensional arrays where each element in the array corresponds to a pixel in the image. The most common format for color images is a 3D NumPy array with dimensions (height, width, channels), where "height" is the image's height in pixels, "width" is the image's width in pixels, and "channels" represents the color channels (usually 3 for Red, Green, and Blue in color images).
- **Data Types:**
  NumPy arrays can use different data types, which are essential for efficient memory storage and image manipulation. In OpenCV, common data types include unsigned 8-bit integers (uint8) for grayscale images and unsigned 8-bit integers with three channels (uint8) for color images.
- **Efficient Image Manipulation:**
  NumPy provides a wide range of functions and methods for array manipulation, allowing you to perform various image operations efficiently. You can use NumPy operations for tasks like cropping, resizing, rotating, filtering, and many other image processing tasks. This makes it convenient to work with image data directly using the same tools as other numerical data.
- **Compatibility with Other Libraries:**
  Using NumPy arrays to represent images ensures compatibility with other Python libraries and tools for scientific computing and data analysis. You can seamlessly integrate OpenCV with libraries like scikit-learn, TensorFlow, and more.
- **Access to Individual Pixels:**
  NumPy arrays allow you to access and manipulate individual pixels in an image easily. You can read, modify, or process pixel values using standard array indexing and slicing.
- **Interoperability:**
  OpenCV is often used in conjunction with other computer vision and machine learning libraries. NumPy's array format facilitates the exchange of image data between OpenCV and other libraries.

Step 3: Image Processing and Transformation

Introduce basic image processing operations, such as resizing, cropping, and color conversions.

```
# Resize the image

resized_image = cv2.resize(image, (width, height))

# Crop a region of interest (ROI)

roi = image[y:y+h, x:x+w]

# Convert the image to grayscale

gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

Part 1: Resizing the Image:

This part of the code resizes the input image, which is stored in the image variable, to a new size specified by width and height.

- **cv2.resize** is a function provided by OpenCV for image resizing. It takes two main arguments:
  - The first argument is the image you want to resize (image in this case).
  - The second argument is a tuple representing the new dimensions (width, height) you want the image to be resized to.

After executing this line of code, resized_image will contain the resized image.

Part 2: Cropping a Region of Interest (ROI):

This code extracts a region of interest (ROI) from the original image. The ROI is defined by specifying the coordinates of a rectangle within the image. In addition, x and y are the starting coordinates of the top-left corner of the rectangle, and w and h are the width and height of the rectangle.

- image[y:y+h, x:x+w] is a NumPy array slicing operation. It selects a sub-region of the original image by specifying the range of rows and columns.

After this line is executed, the variable roi contains the extracted region of interest.

Part 3: Converting the Image to Grayscale:

This part of the code converts the original image to grayscale.

- The cv2.cvtColor function is used for color space conversion in OpenCV. It takes two main arguments:
    - The first argument is the source image (image in this case).
    - The second argument is the conversion code, which specifies the type of conversion. In this case, cv2.COLOR_BGR2GRAY is used to convert the image to grayscale.

After executing this line of code, gray_image will contain the grayscale version of the original image**.**

Step 4: Example that combines OpenCV and NumPy to perform image manipulation.

In this example, we'll take an image, apply a simple color filter to it, and display the processed image.

```
import cv2

import numpy as np

# Load an image

image = cv2.imread('your_image.jpg')

# Create a color filter (e.g., blue filter)

blue_filter = np.zeros_like(image)

blue_filter[:,:,0] = image[:,:,0]  # Blue channel

# Display the original and filtered images

cv2.imshow('Original Image', image)

cv2.imshow('Blue Filtered Image', blue_filter)

# Wait for a key press and then close the windows

cv2.waitKey(0)

cv2.destroyAllWindows()
```

Hereby we are explaining the code in Details:

Part 1: Creating a Color Filter:

- The code creates a color filter, in this case, a "blue filter," by creating a NumPy array with the same shape as the image but initially filled with zeros. This new array is named **blue_filter.**

- The filter is applied to the image by copying only the blue channel of the original image into the blue filter array. This is done by assigning the blue channel (image[:,:,0]) to the blue filter (blue_filter[:,:,0]).

Example: Try your own photo!

OpenCV example that involves loading an image, performing image processing, and saving the result.

In this example, we'll apply a simple filter to the image, converting it to grayscale and then saving the processed image.

```
import cv2

# Load an image

image = cv2.imread('your_image.jpg')  # Load the specified image
file

# Convert the image to grayscale

gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)  # Convert the
color image to grayscale

# Save the grayscale image

cv2.imwrite('grayscale_image.jpg', gray_image)  # Save the grayscale
image to a file

# Display the original and grayscale images

cv2.imshow('Original Image', image)  # Display the original color
image

cv2.imshow('Grayscale Image', gray_image)  # Display the grayscale
image

# Wait for a key press and then close the windows

cv2.waitKey(0)  # Wait indefinitely until a key is pressed

cv2.destroyAllWindows()  # Close all OpenCV windows
```

Fall 2023
Intelligent Systems
Deep Learning

Applications in Deep Learning:

Example: Image Classification with Convolutional Neural Networks (CNN) and OpenCV Preprocessing:

- **Image Classification:** Image classification is a fundamental computer vision task where the goal is to assign a label or category to an input image. In this task, we aim to teach a machine learning model, in this case, a Convolutional Neural Network (CNN), to recognize and classify objects or patterns within images. Each image in the dataset is associated with a particular class or category, and the model's job is to predict the correct class for a given input image.
- **Convolutional Neural Networks (CNNs):** CNNs are a class of deep learning models that are particularly well-suited for image-related tasks. They consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers. Convolutional layers are designed to automatically and adaptively learn spatial hierarchies of features from input images.
- **OpenCV Preprocessing:** OpenCV (Open Source Computer Vision Library) is an open-source computer vision and image processing library that provides a wide range of functions for tasks such as image loading, manipulation, and analysis. In the context of image classification, OpenCV can be used to preprocess the image data before feeding it into a CNN.

Steps in the Process:

Now, let's break down the process of "Image Classification with Convolutional Neural Networks and OpenCV Preprocessing":

1. **Data Preparation:**

The first step involves preparing the dataset for training and evaluation. In this example, we use the CIFAR-10 dataset, which contains 60,000 32x32 color images in 10 different classes (e.g., airplanes, automobiles, birds, cats, etc.).

2. **Preprocessing with OpenCV:**

Before feeding the images into the CNN model, it's essential to preprocess them. OpenCV is used for various preprocessing steps:

3. **Image Loading:**

OpenCV can be used to load images from the dataset. It's common to load images as NumPy arrays, making them suitable for further processing.

4. **Image Resizing:**

Images may need to be resized to a specific dimension to match the input size expected by the CNN model. This can help standardize the input data.

5. **Normalization:**

Normalizing the pixel values to a specific range (e.g., [0, 1]) is essential to make training more stable and efficient.

6. **CNN Model Building:**

A Convolutional Neural Network is defined using a deep learning framework (e.g., TensorFlow or PyTorch). The architecture of the CNN consists of layers for feature extraction and classification. Convolutional layers learn spatial features, while fully connected layers make class predictions.

7. **Model Compilation:**

The CNN model is compiled by specifying the optimizer, loss function, and evaluation metrics. In this example, the Adam optimizer and sparse categorical cross-entropy loss are used.

8. **Training the Model:**

The model is trained on the preprocessed training data for a specific number of epochs. During training, the model learns to recognize patterns and features in the images and make predictions. The weights of the model are updated to minimize the defined loss.

9. **Model Evaluation:**

The trained model is evaluated on a separate test dataset to assess its accuracy and generalization performance. Accuracy is a common metric used to measure the model's success in correctly classifying test images.

10. **Fine-tuning and Experimentation:**

To improve performance, fine-tuning the model and experimenting with different hyperparameters, network architectures, and preprocessing techniques is often necessary.

Example:

You are asked to choose any ready dataset and use a proper CNN API along with OpenCV preprocessing in order to train, compile and test an image classification task.

The task flow is as the following:

- Choose a dataset (e.g., CIFAR-10, MNIST, or a custom dataset) for image classification.
- Implement a Convolutional Neural Network (CNN) using a deep learning framework of your choice (e.g., TensorFlow or PyTorch).
- Preprocess the dataset using OpenCV to augment and prepare the images.
- Train the CNN model on the preprocessed data.
- Evaluate the model's performance using various metrics and visualize the results.

Hint:

Make the code does the following:

- Loads the CIFAR-10 dataset using TensorFlow.
- Uses OpenCV to preprocess the images by resizing them and normalizing the pixel values.
- Defines a simple CNN model using TensorFlow's Keras API.
- Compiles the model with the Adam optimizer and cross-entropy loss.
- Trains the model on the preprocessed data for 10 epochs.
- Evaluates the model's accuracy on the test data.
- This is a basic example to get you started with image classification using a CNN and OpenCV preprocessing. You can further fine-tune the model, experiment with different architectures, and explore advanced techniques to improve performance.

Code: Try it Now!

```
import tensorflow as tf

from tensorflow.keras import datasets, layers, models

import cv2

import numpy as np


# Load the CIFAR-10 dataset

(train_images, train_labels), (test_images, test_labels) =
datasets.cifar10.load_data()


# Preprocess the data using OpenCV

def preprocess_images(images):

    processed_images = []

    for img in images:

        # Resize the image to a specific size (e.g., 64x64)

        img = cv2.resize(img, (64, 64))

        # Normalize pixel values to be between 0 and 1
```

```
        img = img / 255.0

        processed_images.append(img)

    return np.array(processed_images)


train_images = preprocess_images(train_images)

test_images = preprocess_images(test_images)


# Define the CNN model

model = models.Sequential()

# Convolutional layers for feature extraction

model.add(layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(64, 64, 3))

model.add(layers.MaxPooling2D((2, 2))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2))

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

# Flatten the output for fully connected layers

model.add(layers.Flatten())

# Fully connected layers for classification

model.add(layers.Dense(64, activation='relu'))

model.add(layers.Dense(10))  # 10 output classes for CIFAR-10


# Compile the model

model.compile(optimizer='adam',
```

```
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
,
              metrics=['accuracy'])


# Train the model

model.fit(train_images, train_labels, epochs=10,
validation_data=(test_images, test_labels))



# Evaluate the model

test_loss, test_acc = model.evaluate(test_images, test_labels,
verbose=2)

print("\nTest accuracy:", test_acc)
```

Assignment:

Re-implement with different dataset and compare the two models evaluations.