|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| T-Pyramid of an Image  from PIL import Image  def compute\_t\_pyramid(image\_path, levels):  original\_image = Image.open(image\_path)  pyramid = [original\_image]  current\_image = original\_image  for \_ in range(1, levels):  width, height = current\_image.size  new\_size = (width // 2, height // 2)  reduced\_image = current\_image.resize(new\_size, Image.ANTIALIAS)  pyramid.append(reduced\_image)  current\_image = reduced\_image  return pyramid  def save\_pyramid(pyramid, base\_name):  for i, image in enumerate(pyramid):  image.save(f"{base\_name}\_level\_{i}.png")  print(f"Level {i} saved as {base\_name}\_level\_{i}.png")  image\_path = "/content/drive/MyDrive/PSGiTech/Subjects Taken/CCS349 - Image and Video Analytics - Integrated/Lab/Itechlogo.png"  levels = 4  t\_pyramid = compute\_t\_pyramid(image\_path, levels)  save\_pyramid(t\_pyramid, "output\_image")  Motion Analysis Using Moving Edges  import cv2 np plt low\_threshold = 50 high\_threshold = 150  contour\_color = (0, 255, 0) contour\_thickness = 2  def detect\_edges(frame):  gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)  blurred = cv2.GaussianBlur(gray, (5, 5), 0)  edges = cv2.Canny(blurred, low\_threshold, high\_threshold) return edges  def display\_frame(frame, title="Frame"):  frame\_rgb = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB) plt.figure(figsize=(10, 6))  plt.imshow(frame\_rgb) plt.title(title)  plt.axis('off') plt.show()  def motion\_analysis(video\_path):  cap = cv2.VideoCapture(video\_path)  ret, prev\_frame = cap.read() if not ret:  print("Error: Could not read video.") return  prev\_edges = detect\_edges(prev\_frame)  while cap.isOpened():  ret, frame = cap.read() if not ret: break  edges = detect\_edges(frame)  motion\_edges = cv2.absdiff(edges, prev\_edges)  contours, \_ = cv2.findContours(motion\_edges, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)  motion\_frame = frame.copy()  cv2.drawContours(motion\_frame, contours, -1, contour\_color, contour\_thickness)  display\_frame(motion\_frame, title="Motion Analysis using Moving Edges")  prev\_edges = edges cap.release() video\_path = motion\_analysis(video\_path) | Rotate  import cv2 np plt  def load\_image(image\_path):  image = cv2.imread(image\_path) return image  def display\_image(image, title="Image"):  rgb\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)  plt.imshow(rgb\_image) plt.title(title)  plt.axis('off') plt.show()  def rotate\_image(image, angle):  (h, w) = image.shape[:2] center = (w // 2, h // 2)  M = cv2.getRotationMatrix2D(center, angle, 1.0)  rotated = cv2.warpAffine(image, M, (w, h))  display\_image(rotated, "Rotated Image")  return rotated  def scale\_image(image, scale\_x, scale\_y):  (h, w) = image.shape[:2]  scaled = cv2.resize(image, (int(w \* scale\_x), int(h \* scale\_y)))  display\_image(scaled, "Scaled Image") return scaled  def skew\_image(image, skew\_x, skew\_y):  (h, w) = image.shape[:2]  M = np.float32([[1, skew\_x, 0], [skew\_y, 1, 0]])  skewed = cv2.warpAffine(image, M, (w, h))  display\_image(skewed, "Skewed Image")  return skewed  def affine\_transform(image, src\_points, dst\_points):  M = cv2.getAffineTransform(np.float32(src\_points), np.float32(dst\_points))  (h, w) = image.shape[:2]  transformed = cv2.warpAffine(image, M, (w, h))  display\_image(transformed, "Affine Transformed Image") return transformed  def bilinear\_transform(image, src\_points, dst\_points):  M = cv2.getPerspectiveTransform(np.float32(src\_points), np.float32(dst\_points)) (h, w) = image.shape[:2]  transformed = cv2.warpPerspective(image, M, (w, h))  display\_image(transformed, "Bilinear Transformed Image") return transformed  def main():  image\_path =  image = load\_image(image\_path)  rotated\_image = rotate\_image(image, 45)  scaled\_image = scale\_image(image, 1.5, 0.5)  skewed\_image = skew\_image(image, 0.2, 0.0)  src\_points\_affine = [(50, 50), (200, 50), (50, 200)]  dst\_points\_affine = [(10, 100), (200, 50), (100, 250)]  affine\_image = affine\_transform(image, src\_points\_affine, dst\_points\_affine)  src\_points\_bilinear = [(0, 0), (300, 0), (0, 300), (300, 300)]  dst\_points\_bilinear = [(0, 0), (320, 50), (0, 320), (320, 320)]  bilinear\_image = bilinear\_transform(image, src\_points\_bilinear, dst\_points\_bilinear)  if \_\_name\_\_ == "\_\_main\_\_": main() | |  | |  | |
|  |  | |  | |  | |
|  | |  | | Quad tree  from PIL import Image, ImageDraw  import numpy as np  class QuadTreeNode:  def \_\_init\_\_(self, x, y, size, intensity=None):  self.x = x self.y = y self.size = size  self.intensity = intensity self.children = None  def is\_leaf(self):  return self.children is None  def is\_homogeneous(region, threshold=0):  unique\_values = np.unique(region)  return len(unique\_values) == 1 or (len(unique\_values) == 2 and abs(unique\_values[0] - unique\_values[1]) <= threshold)  def build\_quadtree(image, x, y, size, threshold=0):  region = image[y:y+size, x:x+size]  if is\_homogeneous(region, threshold) or size == 1:  intensity = np.mean(region)  return QuadTreeNode(x, y, size, intensity=intensity)  half\_size = size // 2  node = QuadTreeNode(x, y, size)  node.children = [  build\_quadtree(image, x, y, half\_size, threshold),  build\_quadtree(image, x + half\_size, y, half\_size, threshold),  build\_quadtree(image, x, y + half\_size, half\_size, threshold),  build\_quadtree(image, x + half\_size, y + half\_size, half\_size, threshold)  ] return node  def draw\_quadtree(draw, node):  if node.is\_leaf():  x, y, size = node.x, node.y, node.size  draw.rectangle([x, y, x + size, y + size], outline="red")  else:  for child in node.children:  draw\_quadtree(draw, child)  def load\_image\_grayscale(image\_path):  image = Image.open(image\_path).convert('L')  return np.array(image)  def main():  image\_path =  image = load\_image\_grayscale(image\_path)  height, width = image.shape  size = 1 << max(height.bit\_length(), width.bit\_length() - 1)  image\_padded = np.zeros((size, size), dtype=image.dtype)  image\_padded[:height, :width] = image  quadtree = build\_quadtree(image\_padded, 0, 0, size)  result\_image = Image.fromarray(image\_padded)  draw = ImageDraw.Draw(result\_image)  draw\_quadtree(draw, quadtree)  result\_image.show()  result\_image.save("quadtree\_output.png")  if \_\_name\_\_ == "\_\_main\_\_": main() | | Face Recognition  from tensorflow.keras.preprocessing.image import ImageDataGenerator import os zipfile plt  home\_dir = os.path.expanduser('~')  data\_dir = os.path.join(home\_dir, 'Downloads', 'lfw\_facedata', 'lfw-deepfunneled', 'lfw-deepfunneled')  datagen = ImageDataGenerator(rescale=1./255, validation\_split=0.2)  train\_data = datagen.flow\_from\_directory(  directory=data\_dir, target\_size=(128, 128),  batch\_size=32, class\_mode='categorical',  subset='training' )  def extract\_zip(zip\_file\_path, extract\_to\_folder):  os.makedirs(extract\_to\_folder, exist\_ok=True)  with zipfile.ZipFile(zip\_file\_path, 'r') as zip\_ref:  zip\_ref.extractall(extract\_to\_folder)  print(f"Extracted all files to {extract\_to\_folder}")  zip\_file\_path = 'Dataset/lfw\_facedata.zip'  extract\_to\_folder = 'Dataset'  extract\_zip(zip\_file\_path, extract\_to\_folder)  val\_data = datagen.flow\_from\_directory(  directory=data\_dir, target\_size=(128, 128),  batch\_size=32, class\_mode='categorical',  subset='validation' )  print(train\_data.class\_indices) images, labels = next(train\_data) plt.imshow(images[3])  y\_train = train\_data.labels y\_val = val\_data.labels  print("Training labels shape:", y\_train.shape)  print("Validation labels shape:", y\_val.shape)  print("First few training labels:", y\_train[:5])  class\_indices = train\_data.class\_indices  index\_to\_class = {v: k for k, v in class\_indices.items()}  predicted\_label = 3  person\_name = index\_to\_class[predicted\_label]  print("Predicted person:", person\_name)  from tensorflow.keras import layers, models  model = models.Sequential()  model.add(layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(128, 128, 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(128, (3, 3), activation='relu'))  model.add(layers.MaxPooling2D((2, 2)))  model.add(layers.Flatten())  model.add(layers.Dense(444, activation='relu'))  model.add(layers.Dropout(0.2))  model.add(layers.Dense(5749, activation='softmax'))  model.summary()  model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])  model.fit(train\_data, validation\_data=val\_data, epochs=10)  import cv2 img = cv2.imread("Aaron\_Guiel\_0001.jpg", cv2.IMREAD\_COLOR) print(f"Image shape: {img.shape}")  img = cv2.resize(img, (128, 128)) plt.imshow(img)  img = np.expand\_dims(img, axis=0)  result = model.predict(img)  predicted\_label = np.argmax(result)  person\_name = index\_to\_class[predicted\_label]  print("Predicted person:", person\_name) |
|  | |  | |  | |  |
|  | |  | |  | |  |
|  | |  | |  | |  |
|  | |  | |  | |  |
|  | |  | |  | |  |
|  | |  | |  | |  |