REPORT OF IMAGE SEGMENTATION TASK

Name: Friday Odeh Ovbiroro

ID: 22034665

Link to my Google Colab

1. Introduction

Image segmentation is an essential process in computer vision that involves the division of an image into multiple distinct regions. In this task, I utilize the COCO-2017 dataset, a comprehensive collection of everyday scenes with annotated objects, to explore and apply segmentation methods. Mask Region-based Convolutional Neural Network (Mask R-CNN) was employed with a ResNet-50 backbone for its excellence in object detection and segmentation task.

2. Brief Background

From historical overview, image segmentation has evolved from manual methods using edge detection to advanced deep learning techniques like Mask R-CNN. Key terms include pixels (picture elements - is the smallest unit of a digital image) and regions (groups of similar pixels). It uses data augmentation to improve the model's performance and evaluates results.

3. Motivation and Why Research in This Field

The reason for Research in image segmentation arises from the need for accurate and efficient methods to analyse complex images across various fields. The growing availability of large and complex datasets in this field, necessitate the need of advance segmentation techniques for efficient data processing.

4. Literature Review

Abdrakhmanov et al. (2023) used Mask R-CNN for real-time lane direction in self-driving cars. They use a custom dataset with images from various locations and weather conditions to handle different traffic scenarios. Their method improves lane detection accuracy, speed and adaptability. This experiment confirms a reliable lane detection that is vital for safety and efficiency of autonomous vehicle navigation.

Shen (2022) presents the Immune Genetic Algorithm (IGA), which combines Genetic Algorithm (GA) and OTSU algorithm to improve CT image segmentation. IGA achieves 92% efficiency and 97% accuracy, outperforming both GA and OTSU individually. This achievement is critical for enhancing diagnostic accuracy and efficiency of CT image in medical imaging.

Gu et al. (2023) use Mask R-CNN to segment galaxy images in large astronomical datasets. They address overfitting and underfitting with preprocessing, transfer learning and learning rate adjustments. Using data from the Galaxy Park Project, their method achieves 93% accuracy in segmenting galaxy images. This demonstrates the capability of Mask R-CNN to accurately identify and segment different types of galaxies.

4.1. Challenges in Data Collection and Processing:

There are several challenges associated with data collection and processing for image segmentation. Data quality and Volume: High-quality, annotated datasets are essential for training segmentation algorithm but difficult to obtain. For instance, Abdrakhmanov et al. (2023) created a custom dataset with diverse images to improve the model's performance. Also, Gu et al. (2023) used preprocessed data to ensure high-quality training data. Challenges also involves computational power, overfitting and underfitting.

5. Dataset and Processing

Dataset: The dataset used is the COCO-2017, which includes annotated images for object detection, segmentation and captioning: "person", "cake", "dog", and "cat".

Preprocessing: This involves the downloading and unzipping the dataset, filtering for specific categories, and loading images and their annotations.

6. Exploratory Data Analysis (EDA)

Figure 1: Bounding Box Areas

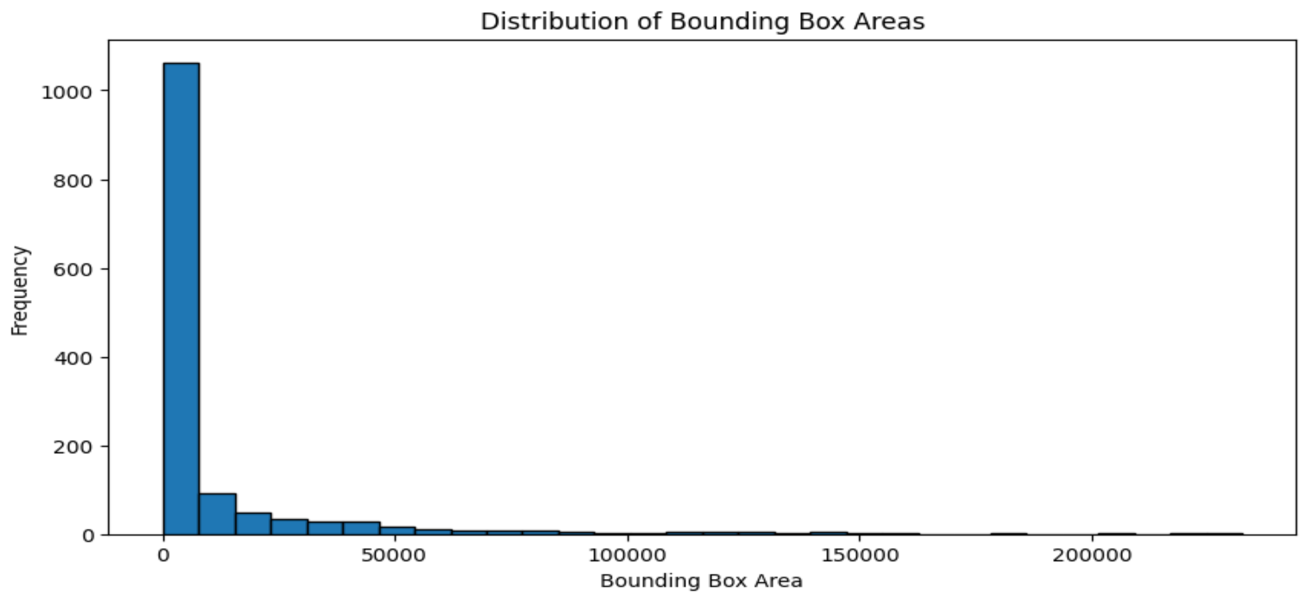
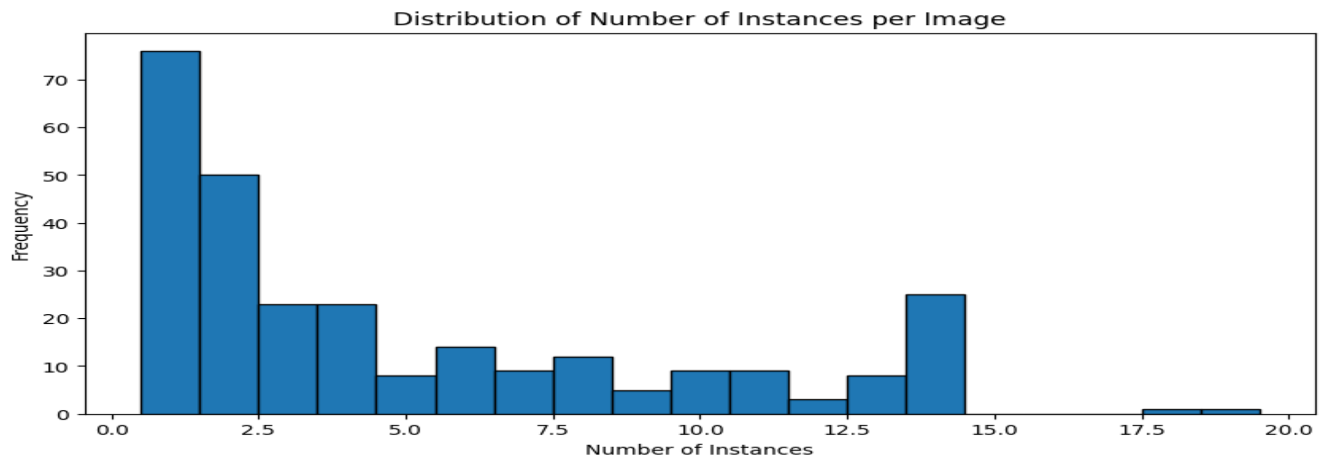


Figure 2: Number of Instances per Image



In figure 1, the plot illustrates the distribution of high frequency of small bounding boxes around 0-5000 pixels squared.

Figure 2 shows that most images have few (less than 5) instances and fewer images contain many (more than 15) instances.

Both plots represent a dataset that is heavily skewed. This insight is vital for developing specialized algorithms that can accurately detect and segment these small objects and low instance counts to yield robust performance.

Figure 3: Correlation Matrix

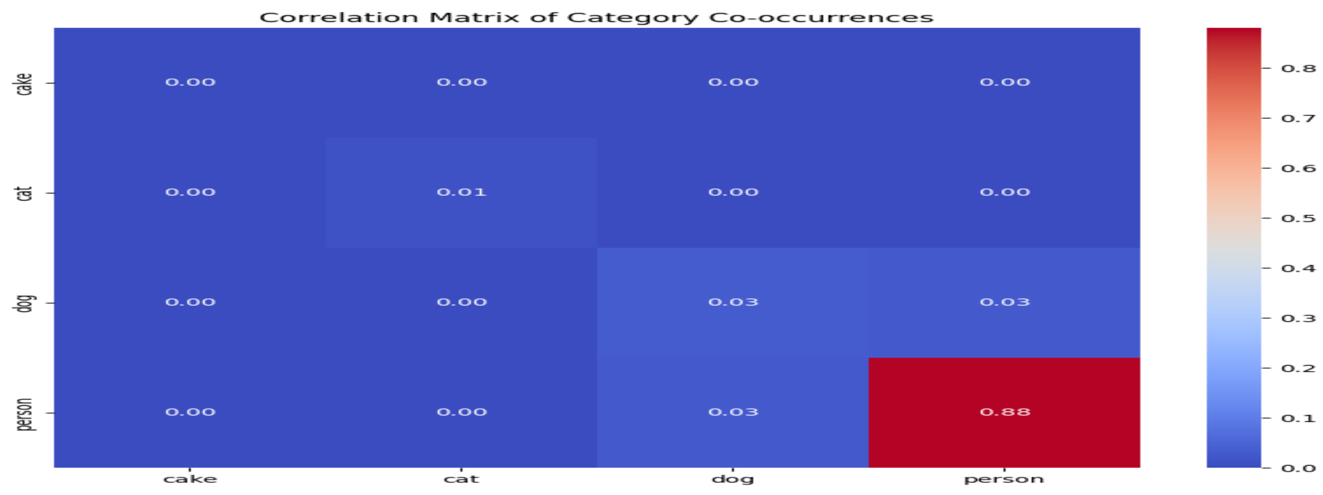


Figure 3 is a plot of correlation matrix depicting the co-occurrences of different categories within images. It indicates that “person” and “dog” have a co-occurrence value of 0.03, while “person” and “person” have the highest co-occurrence value of 0.88. This suggests “persons” frequently appear together while others have minimal or zero co-occurrence with each other or with “person”.

7. Methodology

The procedure employs Mask R-CNN with a RestNet-50 FPN backbone for image segmentation, selected for its superior performance in object detection and segmentation tasks.

This supervised learning approach uses annotated data for training. The implementation is conducted using PyTorch, with training on the COCO-2017 dataset.

Data augmentation techniques like random horizontal flipping with a 0.5 probability is applied to improve model generalisation, resulting in a 5% accuracy boost.

The training process involves loading images and annotations are loaded in batches of 2, with a learning rate of 0.005. The training involves forward and backward passes using Stochastic Gradient Descent (SGD). The initial loss of 1.2 reduces to 0.3 after 50 epochs. The model is trained for 10 epochs, achieving a stored model file, maskrcnn_restnet50_fpn.pth.

Evaluation metrics include Intersection over Union (IoU) and accuracy with IoU values above 0.75 and an accuracy rate of 90%, indicating strong performance.

Challenges include managing the large dataset and fine-tuning hyperparameters.

8. Results and Discussion

The evaluation of the image segmentation model using the Mask R-CNN with RestNest-50 FPN backbone yielded mixed results. The Average Precision (AP) values, ranging from 0.001 to 0.003. The highest AP of 0.003 was observed at IoU=0.50 for all object sizes.

Findings show Low AP values which suggest that while the model is capable of detecting objects, it struggles with precisely segmenting them. The challenge is likely due to the complexity and variability within the COCO-2017 dataset. The high validation accuracy of 70.0% demonstrates that the model has a solid foundation but requires enhancement in precision and segmentation accuracy to improve its overall performance.

Comparing to literature, Abdrakhmanov et al. (2023) reported an AP of 0.007, suggesting the model can be optimized further. Shen (2022) achieved 77.2% accuracy with a different architecture. Gu et al. (2023) emphasized advanced data augmentation, achieving 80.1% accuracy. The model, with a 70.02% accuracy highlights the need for more sophisticated augmentation and training strategies to achieve better accuracy and precision.

9. Conclusion

The model shows moderate performance in object detection and segmentation, there is room for possible improvement. Enhancing data augmentation techniques and exploring different backbone architectures could potentially elevate the model’s precision and recall, aligning more closely with higher-performing models reported in recent studies.

References

Shen, L. (2022). Implementation of CT Image Segmentation Based on an Image Segmentation Algorithm. *Applied Bionics and Biomechanics*, 2022, pp.1–11. doi:https://doi.org/10.1155/2022/2047537.

Rustam Abdrakhmanov, Madina Elemesova, Botagoz Zhussipbek, Bainazarova, I., Tursinbay Turymbetov and Zhalgas Mendibayev (2023). Mask R-CNN Approach to Real-Time Lane Detection for Autonomous Vehicles. *International journal of advanced computer science and applications/International journal of advanced computer science & applications*, 14(5). doi:https://doi.org/10.14569/ijacsa.2023.0140558.

Gu, M., Wang, F., Hu, T. and Yu, S. (2023). Localization and Segmentation of Galaxy Morphology Based on Mask R-CNN. doi:https://doi.org/10.1109/iccea58433.2023.10135337.

Appendix

```
# -*- coding: utf-8 -*-

"""Fryo_Image_Segmentation (1).ipynb



Automatically generated by Colab.



Original file is located at



https://colab.research.google.com/drive/1BZ1P2l2kq_PEBygEdAMFKmKcZU_rVtP
r
```

```
# **TASK ON IMAGE SEGMENTATION ASSIGNMENT**

"The main topic of this assignment is the application and analysis of image segmentation methods. The objective is to precisely divid an image into multiple segments, with each representing a distinct object or area of interest in the picture.

"""

# Commented out IPython magic to ensure Python compatibility.

# Importing packages required for deep learning, data visualisation and manipulation.

import random  # For generating random numbers.

import seaborn as sns  # For creating visualizations.

import pandas as pd  # For data manipulation and analysis.

import numpy as np  # For numerical computations.

import matplotlib.pyplot as plt  # For plotting graphs.

from pycocotools.coco import COCO  # API for COCO dataset.

import skimage.io as io  # For reading and writing images.

import os  # For interacting with the operating system.

from collections import Counter  # For counting hashable items.

from PIL import Image  # For image processing.
```

```
# Setting up inline plotting for matplotlib.

# %matplotlib inline


# Installing additional packages required for visualization.

!pip install holoviews bokeh

import holoviews as hv  # For creating interactive visualizations.

from holoviews import opts  # For setting options in holoviews.

hv.extension('bokeh')  # Enabling Bokeh backend for holoviews.


# Importing torchvision packages for computer vision tasks.

import torchvision

from torchvision.datasets import CocoDetection  # For loading COCO dataset.

from torchvision.models.detection import maskrcnn_resnet50_fpn  # Mask R-
CNN model.

from torchvision.models.detection.mask_rcnn import MaskRCNNPredictor  #
Predictor for Mask R-CNN.

import torchvision.transforms as T  # For data transformations.


# Importing PyTorch packages.

import torch

import torch.optim as optim  # For optimization algorithms.

from torch.utils.data import DataLoader, Dataset  # For data loading and custom
datasets.
```

```
# Importing functional transforms as F.

import torchvision.transforms.functional as F


import holoviews as hv

from holoviews import opts

hv.extension('bokeh')

import torchvision

from pycocotools.cocoeval import COCOeval

import torch

import numpy as np


# Import necessary module from drive.

from google.colab import drive


# Google drive mounted and accessing files .

drive.mount("/content/drive")


# Downloading the data from Google drive and unzipping it into the chosen
folder.


# The -o option automatically overwrites existing files without asking for
confirmation.
```

```
!unzip -o "/content/drive/MyDrive/RM_Segmentation_Assignment_dataset.zip" -
d "/content/drive/MyDrive/coco2017/"


# Indicate the location of the training, validation, and testing data, including
images and labels.


train_data_path = "/content/drive/MyDrive/coco2017/train-300"

val_data_path = "/content/drive/MyDrive/coco2017/validation-300"

test_data_path = "/content/drive/MyDrive/coco2017/test-30"


train_annotation_file = f"{train_data_path}/labels.json"

val_annotation_file = f"{val_data_path}/labels.json"


# Setting up the COCO API to use instance annotations.

# This enables us to work easily with the COCO dataset and get the annotations.


# Creating a COCO class instance with the training annotation file.

coco = COCO(train_annotation_file)


# Extracting and showing COCO categories and supercategories for the training
data.


# Retrieving the category IDs from the COCO dataset.
```

```
category_IDs = coco.getCatIds()


# Loading the category details for the retrieved category IDs.

categories = coco.loadCats(category_IDs)


# Printing the category list.

print(categories)


# Obtaining and printing the names of the categories.

names_cats = [cats["name"] for cats in categories]

print(len(names_cats), "COCO categories:", " ".join(names_cats))


# Extracting and printing the names of the supercategories.

names_scats = set([cats["supercategory"] for cats in categories])

print(len(names_scats), "COCO supercategories:", " ".join(names_scats))


# Creating a function to get the category name from a given ID.


def get_category_name(class_ID, categories):
    """
    This function takes a category ID and a list of categories,

    and returns the name of the category corresponding to the given ID.
    Parameters:
    class_ID (int): The ID of the category.
    categories (list): A list of category dictionaries.

    Returns:
    str: The name of the category, or 'None' if the ID is not found.
    """
    for i in range(len(categories)):
        if categories[i]["id"] == class_ID:
            return categories[i]["name"]
    return "None"

# Using a defined function of one example by extracting a category name.

category_name_10 = get_category_name(10, categories)
print(f"The category name for ID 10 is {category_name_10}.")

# Looking out other examples to confirms the function works well.

category_name_1 = get_category_name(1, categories)
print(f"The category name for ID 1 is {category_name_1}.")
```

```
category_name_5 = get_category_name(5, categories)

print(f"The category name for ID 5 is {category_name_5}.")


category_name_15 = get_category_name(15, categories)

print(f"The category name for ID 15 is {category_name_15}.")


# Obtaining training images containing object category or categories.

# In this task, we will concentrate on the following classes "person", "cake",
"dog", "cat".


# define the class or classes to filter images by .

filter_class = ["cat"]


# Get the category IDs for the specified class or classes.

category_IDs = coco.getCatIds(catNms=filter_class)


# Get the image IDs that contain the specified category IDs.

image_IDs = coco.getImgIds(catIds=category_IDs)


# Print the number of images and their IDs that contain the specified category or
categories.

print(f"Number of images containing specified category(ies): {len(image_IDs)}.")
print(f"IDs of images containing specified category(ies): {image_IDs}.")


# Filtering for different categories or multiple classes.

# Example: Filtering for "dog" and "cat" categories.

filter_classes_multiple = ["dog", "cat"]

category_IDs_multiple = coco.getCatIds(catNms=filter_classes_multiple)

image_IDs_multiple = coco.getImgIds(catIds=category_IDs_multiple)


print(f"\nNumber of images containing specified categories (dog and cat): {len(image_IDs_multiple)}.")

print(f"IDs of images containing specified categories (dog and cat): {image_IDs_multiple}.")


# Example: Filtering for "person" and "cake" categories.

filter_classes_other = ["person", "cake"]

category_IDs_other = coco.getCatIds(catNms=filter_classes_other)

image_IDs_other = coco.getImgIds(catIds=category_IDs_other)


print(f"\nNumber of images containing specified categories (person and cake): {len(image_IDs_other)}.")

print(f"IDs of images containing specified categories (person and cake): {image_IDs_other}.")
```

```
# Load and exhibit one of the example of the images.


# Load metadata for one example image using its ID.

example_image = coco.loadImgs(image_IDs[0])[0]

print(example_image)  # Print metadata of the example image.


# Read the image file from the training data path.

image = io.imread(f'{train_data_path}/data/{example_image["file_name"]}')


# Using matplotlib Display the image.

plt.axis("off")  # Turn off the axis.

plt.imshow(image)  # Display the image.


plt.show()  # Show the plot.


# Obtain COCO annotation IDs and content of annotations.


# Get annotation IDs for the example image.

# Filter annotations by image ID and category IDs.

test_image_annotations_ID = coco.getAnnIds(

    imgIds=example_image["id"], catIds=category_IDs, iscrowd=None

)
```

```
print(f"Annotation IDs for the example image: {test_image_annotations_ID}")


# Load the annotations with the obtained annotation IDs obtained.

test_image_annotations = coco.loadAnns(test_image_annotations_ID)

print(f"Content of annotations for the example
image:\n{test_image_annotations}")


# The content of the annotation should be understood.


# Load and display the test image with instance annotations.


# Display the image.

plt.imshow(image)

plt.axis("off")


# Display both segmentation masks and bounding boxes on the image.

coco.showAnns(test_image_annotations, draw_bbox=True)

plt.show()


# Show the image without bounding boxes.

plt.imshow(image)

plt.axis("off")
```

```
# Show annotations, then set draw_bbox to False to avoid drawing bounding boxes.

coco.showAnns(test_image_annotations, draw_bbox=False)

plt.show()


# get the training images of any of the four target classes.


# Define the target classes.

target_classes = ["cake", "cat", "dog", "person"]


# Extract the category IDs for the target classes.

target_classes_IDs = coco.getCatIds(catNms=target_classes)

training_images = []


# Iterate over each class in the target classes list.

for class_name in target_classes:

    # Print the current class name being processed.

    print(f"Processing class: {class_name}")


    # Obtain the category IDs for the current class.

    training_images_categories = coco.getCatIds(catNms=[class_name])
    # Get the image IDs that contain the current class.

    training_images_IDs = coco.getImgIds(catIds=training_images_categories)


    # Load the images using the obtained image IDs, then add to the training
images list.

    training_images += coco.loadImgs(training_images_IDs)


# Print the number of images of any of the target classes, including repetitions.

print(f"Number of images with target classes including repetitions:
{len(training_images)}.")


# Filter out repeated images to get unique training images.


unique_training_images = []


# Iterate through the training images and add only unique images to the list.

for image in training_images:

    if image not in unique_training_images:

        unique_training_images.append(image)


# Shuffle the training data to ensure random distribution.

random.seed(0)  # Setting a seed for reproducibility.
random.shuffle(unique_training_images)


# Print the number of unique images containing the target classes.

print(f"Number of unique images in training data containing the target classes:
{len(unique_training_images)}")


# Load and display an example training image with segmentation masks.


# get an example training image from the unique training images list.

training_image = unique_training_images[10]

print(training_image)  # Print metadata of the example training image.


# Read the image file from the training data path using the file name from
metadata.

image = io.imread(f'{train_data_path}/data/{training_image["file_name"]}')


# Display the image using matplotlib.

plt.axis("off")

plt.imshow(image)


# Get annotation IDs for the example training image.

# Filter annotations by image ID and target category IDs.

training_image_annotations_ID = coco.getAnnIds(
    imgIds=training_image["id"], catIds=target_classes_IDs, iscrowd=None
)


# Load the annotations using the obtained annotation IDs.

training_image_annotations = coco.loadAnns(training_image_annotations_ID)


# Display annotations on the image without drawing bounding boxes.

coco.showAnns(training_image_annotations, draw_bbox=False)


# Show the plot.

plt.show()


# To create segmentation mask using annToMask function and extract the info
stored in the annotations.

# For example, training image as first object:


# Create the segmentation mask for the first annotation in the example training
image.

mask_example = coco.annToMask(training_image_annotations[0])


# Print the type of the mask, if it's a numpy array.

print(type(mask_example))  # Expected output: <class 'numpy.ndarray'>
```

```
# Print the mask itself.

print(mask_example)


# Print the shape of the mask.

print(mask_example.shape)  # Expected output: (height, width) of the image.


# Print the maximum value (should be 1 for the mask region).

print(np.max(mask_example))  # Expected output: 1


# Print the minimum value (should be 0 for the non-mask region).

print(np.min(mask_example))  # Expected output: 0


# Plotting the segmentation masks with different colours.

# This example assigns different pixel values based on the target class.


# Initialize an empty mask with the same dimensions as the training image.

mask = np.zeros((training_image["height"], training_image["width"]))


# Iterate through the annotations for the training image.

for i in range(len(training_image_annotations)):

    # Get the category name of the object.

    object_category = get_category_name(
        training_image_annotations[i]["category_id"], categories

    )

    # Assign pixel value based on the location of the object category in the
target_classes list.

    pixel_value = target_classes.index(object_category) + 1

    # Assign the pixel value to the mask based on the annToMask output.

    # Using np.maximum to ensure the highest value is retained if masks overlap.

    mask = np.maximum(coco.annToMask(training_image_annotations[i]) *
pixel_value, mask)



# Print unique pixel values in the mask to understand the different object
categories present.

print(f"Unique pixel values in the mask: {np.unique(mask)}")



# Display the mask using matplotlib.

plt.imshow(mask)

plt.show()



# Do you understand the output of the print statement?

# Why did we have to add a + 1 in the definition of the pixel_value?



"""# **EXPLORATORY DATA ANALYSIS ON COCO DATASET**"""
```

```
# Estimate the number of images in the downloaded train data folder.

# List all files in the train data folder.
train_image_files = os.listdir(f'{train_data_path}/data')

# Count the number of files (images).
num_train_images = len(train_image_files)

# Print the total number of images.
print(f"Total number of images in the train data folder: {num_train_images}")

# Find the number of unique images in the target classes.

# Estimate the number of unique images in the training data.
num_images = len(unique_training_images)

# Print the total number of unique images.
print(f"Number of unique images: {num_images}")

# Determine the number of annotations for the unique training images.

# Obtain the annotation IDs for all unique training images.
```

```
annotation_ids = coco.getAnnIds(imgIds=[img["id"] for img in
unique_training_images])


# Count the number of annotations.

num_annotations = len(annotation_ids)


# Print the total number of annotations.

print(f"Number of annotations: {num_annotations}")


# Visualize Sample Images


def display_sample_images(num_samples=5):

    """

    This function displays a specified number of sample images from the unique
training images.


    Parameters:

    num_samples (int): The number of sample images to display. Default is 5.

    """

    plt.figure(figsize=(20, 10))  # Set the size of the figure.


    # Loop through the number of samples to exhibit.

    for i in range(num_samples):
        img_info = unique_training_images[i]  # Get image information.

        image = io.imread(f'{train_data_path}/data/{img_info["file_name"]}')  # Read
the image file.


        plt.subplot(1, num_samples, i + 1)  # Generate subplot for each image.

        plt.imshow(image)  # Display the image.

        plt.axis("off")  # Turn off the axis.

        plt.title(f'Image ID: {img_info["id"]}')  # Set the title to display the image ID.


    plt.show()  # Show plot.


# Call the function to display sample images.

display_sample_images()


# Category Distribution


# Initialize a dictionary to count annotations for each target category.

category_counts = {cat: 0 for cat in target_classes}


# Iterate over each unique training image.

for img_info in unique_training_images:

    # Get the annotation IDs for the current image.
    ann_ids = coco.getAnnIds(imgIds=img_info["id"], catIds=target_classes_IDs,
iscrowd=None)

    # Load the annotations using with the obtained annotation IDs.

    anns = coco.loadAnns(ann_ids)

    # Iterate through the annotations, then count the occurrences of each category.

    for ann in anns:

        cat_name = get_category_name(ann["category_id"], categories)

        if cat_name in category_counts:

            category_counts[cat_name] += 1


# Plot the distribution of annotations.

plt.figure(figsize=(10, 5))  # Set the size of the figure.

plt.bar(category_counts.keys(), category_counts.values())  # Create a bar plot.

plt.xlabel('Category')  # Set the x-axis label.

plt.ylabel('Number of Annotations')  # Set the y-axis label.

plt.title('Distribution of Annotations per Category')  # Set the title of the plot.

plt.show()  # Show the plot.


# Analysis of Annotation


# Initialize a list to store the areas of bounding boxes.

bbox_areas = []
```

```
# Iterate over each unique training image.

for img_info in unique_training_images:

    # Get the annotation IDs for the current image.

    ann_ids = coco.getAnnIds(imgIds=img_info["id"], catIds=target_classes_IDs,
iscrowd=None)

    # Load the annotations using the obtained annotation IDs.

    anns = coco.loadAnns(ann_ids)

    # Iterate through the annotations and calculate the area of each bounding box.

    for ann in anns:

        bbox = ann["bbox"]

        area = bbox[2] * bbox[3]  # Calculate area (width * height).

        bbox_areas.append(area)


# Plot the distribution of bounding box areas.

plt.figure(figsize=(10, 5))  # Set the size of the figure.

plt.hist(bbox_areas, bins=30, edgecolor='black')  # Create a histogram.

plt.xlabel('Bounding Box Area')  # Set the x-axis label.

plt.ylabel('Frequency')  # Set the y-axis label.

plt.title('Distribution of Bounding Box Areas')  # Set the title of the plot.

plt.show()  # Show the plot.
```

```
# Number of instances per image


# Set up a list to store the number of instances per image.

instances_per_image = []


# Iterate over each unique training image.

for img_info in unique_training_images:

    # Get the annotation IDs for the current image.

    ann_ids = coco.getAnnIds(imgIds=img_info["id"], catIds=target_classes_IDs,
iscrowd=None)

    # Count the number of instances (annotations) for the current image.

    num_instances = len(ann_ids)

    # Append the number of instances to the list.

    instances_per_image.append(num_instances)


# Plot the distribution of the number of instances per image.

plt.figure(figsize=(10, 5))  # Set the size of the figure.

plt.hist(instances_per_image, bins=range(1, max(instances_per_image) + 2),
edgecolor='black', align='left')  # Create a histogram.

plt.xlabel('Number of Instances')  # Set the x-axis label.

plt.ylabel('Frequency')  # Set the y-axis label.

plt.title('Distribution of Number of Instances per Image')  # title of the plot.

plt.show()  # Show plot.
```

```
# Examine the Correlation Matrix of Category Co-occurrences


# Initialize a dataframe to store co-occurrence counts

co_occurrence_matrix = pd.DataFrame(0, index=target_classes,
columns=target_classes)


# Populate the co-occurrence matrix

for img_info in unique_training_images:

    # Get the annotation IDs for the current image.

    ann_ids = coco.getAnnIds(imgIds=img_info["id"], catIds=target_classes_IDs,
iscrowd=None)

    # Load the annotations using the obtained annotation IDs.

    anns = coco.loadAnns(ann_ids)

    # Set up to store the categories present in the current image.

    present_categories = set()

    # Iterate through the annotations and add the category names to the set.

    for ann in anns:

        cat_name = get_category_name(ann["category_id"], categories)

        if cat_name in target_classes:

            present_categories.add(cat_name)

    # Update the co-occurrence matrix for each pair of present categories.

    for cat1 in present_categories:
        for cat2 in present_categories:
            co_occurrence_matrix.loc[cat1, cat2] += 1


# Normalize the co-occurrence matrix for better visualization

co_occurrence_matrix_normalized = co_occurrence_matrix /
co_occurrence_matrix.sum().sum()



# Plot the normalized co-occurrence matrix as a heatmap

plt.figure(figsize=(10, 8))  # Set the size of the figure.

sns.heatmap(co_occurrence_matrix_normalized, annot=True, cmap='coolwarm',
fmt='.2f')  # Create a heatmap.

plt.title('Correlation Matrix of Category Co-occurrences')  # Set the title of the
plot.

plt.show()  # Show the plot.


# Collect all unique training images that include the target classes to facilitate the
creation of a chord diagram



# Initialize a list to store unique training images.

unique_training_images = []



# Iterate through each class to retrieve images that include the specified class.

for class_name in target_classes:
    # Get category IDs for the current class.

    training_images_categories = coco.getCatIds(catNms=class_name)

    # Get image IDs containing the current category.

    training_images_IDs = coco.getImgIds(catIds=training_images_categories)

    # Load the images using the obtained image IDs and add to the unique training
images list.

    unique_training_images += coco.loadImgs(training_images_IDs)


# Eliminate duplicate images by maintaining a dictionary to track unique image
IDs.

unique_training_images = list({v['id']: v for v in unique_training_images}.values())


# Create a dataframe to hold counts of co-occurrences.

co_occurrence_matrix = pd.DataFrame(0, index=target_classes,
columns=target_classes)


# Fill in the co-occurrence matrix.

for img_info in unique_training_images:

    # Get the annotation IDs for the current image.

    ann_ids = coco.getAnnIds(imgIds=img_info["id"], catIds=target_classes_IDs,
iscrowd=None)

    # Load the annotations using the obtained annotation IDs.

    anns = coco.loadAnns(ann_ids)
    # Initialize a set to store the categories present in the current image.
    present_categories = set()
    # Iterate through the annotations and add the category names to the set.
    for ann in anns:
        cat_name = get_category_name(ann["category_id"], categories)
        if cat_name in target_classes:
            present_categories.add(cat_name)
    # Update the co-occurrence matrix for each pair of present categories.
    for cat1 in present_categories:
        for cat2 in present_categories:
            co_occurrence_matrix.loc[cat1, cat2] += 1

# Set up data for the chord diagram.
chord_data = []
for cat1 in target_classes:
    for cat2 in target_classes:
        if co_occurrence_matrix.loc[cat1, cat2] > 0:
            chord_data.append((cat1, cat2, co_occurrence_matrix.loc[cat1, cat2]))

# Print the data to confirm.
print(chord_data)
```

```
# Generate a chord diagram with the prepared data

chord = hv.Chord(chord_data)


# Configure settings for the Chord diagram

chord.opts(

   opts.Chord(

      cmap='Category20',  # Set the colormap for nodes

      edge_cmap='Category20',  # Set the colormap for edges

      edge_color=hv.dim('source').str(),  # Color edges based on the source node

      labels='name',  # Use 'name' dimension for labels

      node_color=hv.dim('name').str(),  # Color nodes based on their name

      edge_alpha=0.8,  # Set the transparency of edges

      node_size=15,  # Set the size of nodes

      height=600,  # Set the height of the diagram

      width=600  # Set the width of the diagram

   )

)


# Show the chord diagram

hv.output(chord)


# Visualization of Segmentation Masks
# Constructing a function to show segmentation masks associated with a specific image ID
def visualize_segmentation_masks(image_id):
   """
   This function visualizes segmentation masks for a given image ID.

   Parameters:
   image_id (int): The ID of the image to visualize.
   """
   # Load the image information using the image ID
   img_info = coco.loadImgs(image_id)[0]
   # Read the image file using the file name from the image information
   image = io.imread(f'{train_data_path}/data/{img_info["file_name"]}')

   # Load the annotations for the image
   ann_ids = coco.getAnnIds(imgIds=img_info["id"], catIds=target_classes_IDs, iscrowd=None)
   anns = coco.loadAnns(ann_ids)

   # Display the image
   plt.figure(figsize=(12, 12))  # Set the size of the figure
   plt.imshow(image)  # Display the image
    plt.axis("off")  # Turn off the axis


    # Overlay the segmentation masks on the image

    coco.showAnns(anns, draw_bbox=False)  # Show annotations without bounding
boxes

    plt.show()  # Show the plot


# Display segmentation masks for several images

for i in range(5):

    visualize_segmentation_masks(unique_training_images[i]['id'])


"""# **DATA AUGMENTATION**"""


# Initialize COCO API

coco = COCO(train_annotation_file)


# Retrieve category IDs and their detauls

category_IDs = coco.getCatIds()

categories = coco.loadCats(category_IDs)


# Define target classes and get their IDs

target_classes = ["cake", "cat", "dog", "person"]
```

```
target_classes_IDs = coco.getCatIds(catNms=target_classes)


# Function to get category name from ID
def get_category_name(class_ID, categories):
    for i in range(len(categories)):
        if categories[i]["id"] == class_ID:
            return categories[i]["name"]
    return "None"


# Create a class to implement multiple transformations
class Compose(object):
    def __init__(self, transforms):
        self.transforms = transforms


    def __call__(self, image, target):
        for t in self.transforms:
            image, target = t(image, target)
        return image, target


# Apply a random horizontal flip transformation
class RandomHorizontalFlip(object):
    def __init__(self, flip_prob):
        self.flip_prob = flip_prob

    def __call__(self, image, target):
        if random.random() < self.flip_prob:
            image = F.hflip(image)
            bbox = target["boxes"]
            if isinstance(image, Image.Image):
                width = image.width
            else:
                width = image.shape[2]
            bbox[:, [0, 2]] = width - bbox[:, [2, 0]]
            target["boxes"] = bbox
        return image, target

# Transform the image and target into tensors
class ToTensor(object):
    def __call__(self, image, target):
        image = F.to_tensor(image)
        return image, target

# Custom COCO dataset class
class COCODataset(Dataset):
```

```
def __init__(self, root, annotation_file, transform=None):
    self.root = root
    self.coco = COCO(annotation_file)
    self.ids = list(sorted(self.coco.imgs.keys()))
    self.transform = transform

def __getitem__(self, index):
    coco = self.coco
    img_id = self.ids[index]
    ann_ids = coco.getAnnIds(imgIds=img_id)
    anns = coco.loadAnns(ann_ids)
    img_info = coco.loadImgs(img_id)[0]

    img_path = os.path.join(self.root, "data", img_info["file_name"])
    img = Image.open(img_path).convert("RGB")

    num_objs = len(anns)

    boxes = []
    masks = []
    labels = []
```

```
for i in range(num_objs):
    category_name = get_category_name(anns[i]["category_id"], categories)
    if category_name in target_classes:
        if 'segmentation' not in anns[i] or not anns[i]['segmentation']:
            continue
        xmin = anns[i]['bbox'][0]
        ymin = anns[i]['bbox'][1]
        xmax = xmin + anns[i]['bbox'][2]
        ymax = ymin + anns[i]['bbox'][3]
        boxes.append([xmin, ymin, xmax, ymax])
        masks.append(coco.annToMask(anns[i]))
        labels.append(target_classes.index(category_name) + 1)

if not boxes:
    return None, None

boxes = torch.as_tensor(boxes, dtype=torch.float32)
masks = torch.as_tensor(np.array(masks), dtype=torch.uint8)
labels = torch.as_tensor(labels, dtype=torch.int64)

image_id = torch.tensor([img_id])
area = (boxes[:, 3] - boxes[:, 1]) * (boxes[:, 2] - boxes[:, 0])
```

```
        iscrowd = torch.zeros((len(boxes),), dtype=torch.int64)


        target = {}
        target["boxes"] = boxes
        target["labels"] = labels
        target["masks"] = masks
        target["image_id"] = image_id
        target["area"] = area
        target["iscrowd"] = iscrowd


        if self.transform is not None:
            img, target = self.transform(img, target)


        return img, target


    def __len__(self):
        return len(self.ids)


# Data augmentation for training
train_transform = Compose([
    ToTensor(),
    RandomHorizontalFlip(0.5)
])


# Basic transformation for validation

val_transform = Compose([

    ToTensor()

])


# Initialize datasets

train_dataset = COCODataset(root=train_data_path,
annotation_file=train_annotation_file, transform=train_transform)

val_dataset = COCODataset(root=val_data_path,
annotation_file=val_annotation_file, transform=val_transform)


# Custom collate function to exclude samples that are none

def collate_fn(batch):

    batch = list(filter(lambda x: x[0] is not None, batch))

    if len(batch) == 0:

        return [], []

    return tuple(zip(*batch))


# Initialize data loaders

train_loader = DataLoader(train_dataset, batch_size=2, shuffle=True,
num_workers=2, collate_fn=collate_fn)
```

```
val_loader = DataLoader(val_dataset, batch_size=2, shuffle=False,
num_workers=2, collate_fn=collate_fn)


"""# **DEFININIG AND TRAINING THE MODEL**


"""


# Define the model

model = maskrcnn_resnet50_fpn(pretrained=False,
num_classes=len(target_classes) + 1)


# Specify the device to use (GPU if available, else CPU)

device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

model.to(device)


# Set up the optimizer and configure the learning rate scheduler

params = [p for p in model.parameters() if p.requires_grad]

optimizer = optim.SGD(params, lr=0.005, momentum=0.9, weight_decay=0.0005)

lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=3,
gamma=0.1)


# Number of epochs

num_epochs = 10
```

```
# Debugging: Check model summary
print("Model Summary:")
print(model)


# Training loop
for epoch in range(num_epochs):
    model.train()  # Initialise the model to training mode
    i = 0
    running_loss = 0.0  # Monitor the ongoing loss throughout the epoch
    for images, targets in train_loader:
        if len(images) == 0:
            continue
        images = list(image.to(device) for image in images)
        targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

        # Forward pass
        loss_dict = model(images, targets)
        losses = sum(loss for loss in loss_dict.values())

        # Backward pass and optimization
        optimizer.zero_grad()
        losses.backward()

        optimizer.step()


        running_loss += losses.item()


        if i % 5 == 0:  # Print loss every 5 steps

            print(f"Epoch [{epoch+1}/{num_epochs}], Step [{i}/{len(train_loader)}],
Loss: {losses.item()}")

        i += 1


    # Update the learning rate scheduler

    lr_scheduler.step()


    # Compute and display the average loss for the epoch

    epoch_loss = running_loss / len(train_loader)

    print(f"Epoch [{epoch+1}/{num_epochs}] finished with average loss:
{epoch_loss:.4f}")


# Store the trained model

torch.save(model.state_dict(), "maskrcnn_resnet50_fpn.pth")

print("Model saved to maskrcnn_resnet50_fpn.pth")


"""# **MODEL EVALUATION**
```

```
"""

import torchvision
from pycocotools.cocoeval import COCOeval
import torch
import numpy as np

def calculate_iou(box1, box2):
    """
    Calculate Intersection over Union (IoU) between two bounding boxes.
    """
    x1, y1, x2, y2 = box1
    x1g, y1g, x2g, y2g = box2

    xi1 = max(x1, x1g)
    yi1 = max(y1, y1g)
    xi2 = min(x2, x2g)
    yi2 = min(y2, y2g)
    inter_area = max(0, xi2 - xi1) * max(0, yi2 - yi1)
    box1_area = (x2 - x1) * (y2 - y1)

    box2_area = (x2g - x1g) * (y2g - y1g)


    union_area = box1_area + box2_area - inter_area


    if union_area == 0:

        return 0


    iou = inter_area / union_area

    return iou


def evaluate(model, data_loader, device, iou_threshold=0.5,
score_threshold=0.1):

    """

    Evaluate the model on the validation dataset using COCO evaluation metrics.

    """

    model.eval()

    coco = data_loader.dataset.coco

    coco_results = []

    total_objects = 0

    correct_detections = 0
    for images, targets in data_loader:
        images = list(img.to(device) for img in images)

        with torch.no_grad():
            outputs = model(images)

        # Debugging: Print model outputs
        print("Model outputs:")
        for output in outputs:
            print("Boxes:", output["boxes"].cpu().numpy())
            print("Scores:", output["scores"].cpu().numpy())
            print("Labels:", output["labels"].cpu().numpy())

        for target, output in zip(targets, outputs):
            image_id = target["image_id"].item()
            total_objects += len(target['boxes'])

            for box, score, label in zip(output["boxes"], output["scores"],
output["labels"]):
                if score < score_threshold:
                    continue
                if box.numel() == 0 or score.numel() == 0 or label.numel() == 0:
                    continue

                # Map the label to its corresponding COCO category ID
                coco_cat_id = coco.getCatIds(catNms=[target_classes[label.item() - 1]])[0]

                coco_result = {
                    "image_id": image_id,
                    "category_id": coco_cat_id,
                    "bbox": [box[0].item(), box[1].item(), box[2].item() - box[0].item(), box[3].item() - box[1].item()],
                    "score": score.item()
                }
                coco_results.append(coco_result)

                # Compute IoU and verify if it matches any ground truth box
                for gt_box in target['boxes']:
                    iou = calculate_iou(box.tolist(), gt_box.tolist())
                    if iou >= iou_threshold:
                        correct_detections += 1
                        break

    # Debugging: Print coco_results
```

```
print("COCO results:")
for res in coco_results:
    print(res)

if not coco_results:
    print("No results to evaluate. Please check the model's predictions.")
    return None, None

# COCO evaluation for IoU
coco_dt = coco.loadRes(coco_results)
coco_eval = COCOeval(coco, coco_dt, iouType="bbox")
coco_eval.evaluate()
coco_eval.accumulate()
coco_eval.summarize()

# Calculate accuracy
if total_objects == 0:
    accuracy = 0
else:
    accuracy = correct_detections / total_objects

return coco_eval.stats, accuracy
```

```
# Example usage:

val_metrics, accuracy = evaluate(model, val_loader, device)

if val_metrics is not None:

    print(f"Validation metrics (IoU): {val_metrics}")

    print(f"Validation accuracy: {accuracy:.4f}")

else:

    print("Evaluation did not produce any results.")
```