Panoptic Image Segmentation using DeepLabV3 on a COCO-style Custom Dataset

Akhila Narra-22100008

19,june

my git click here

Abstract

This project presents an image segmentation approach using the DeepLabV3 model with a ResNet-101 backbone. A COCO-style dataset with four object classes—cake, dog, person, and car—was used. The dataset was divided into training, validation, and test sets, and included image files and corresponding annotations in JSON format. A custom PyTorch Dataset class was developed to handle annotation parsing and mask generation.

Images were preprocessed through resizing and normalization before being passed to the model. The model was trained for 10 epochs using cross-entropy loss and the Adam optimizer. After training, performance was measured using mean Intersection over Union (mIoU) and pixel accuracy metrics. The validation results showed an mIoU of 0.3081 and a pixel accuracy of 88.17%, indicating reasonable segmentation quality. The entire pipeline, from data preparation to model evaluation, was implemented to allow further improvements and experimentation with segmentation tasks.

Keywords: Image Segmentation, DeepLabV3, ResNet-101, COCO Dataset, Semantic Segmentation, PyTorch, mIoU, Pixel Accuracy, Mask Generation, Custom Dataset

1 Introduction

Image segmentation means dividing an image into different parts, such as objects or areas. It is widely used in fields like self-driving cars, medical imaging, and security systems. One important type of segmentation is semantic segmentation, where each pixel in an image is given a label that shows which object it belongs to. This helps computers understand images in more detail.

In recent years, deep learning models like CNNs (Convolutional Neural Networks) have improved the accuracy of image segmentation. DeepLabV3, especially when combined with a ResNet-101 backbone, is one such powerful model known for producing high-quality results.

This project focuses on segmenting images from a custom dataset formatted like COCO, containing four classes: cake, dog, person, and car. A custom PyTorch dataset loader is used to handle the images and corresponding masks. The DeepLabV3 model is trained using cross-entropy loss and the Adam optimizer.

1.1 Aim

The primary aim of this project is to develop and evaluate a deep learning-based semantic segmentation model capable of accurately identifying and segmenting multiple object classes — specifically cake, dog, person, and car — from images in a custom COCO-style dataset. By leveraging the DeepLabV3+ architecture with a ResNet-101 backbone, the objective is to explore the feasibility and effectiveness of state-of-the-art convolutional neural networks in pixel-level classification tasks.

1.2 Objectives

The objectives of this project include:

- Preparing and loading a custom COCO-style dataset with four object categories: *cake*, *dog*, *person*, and *car*.
- Training a DeepLabV3 model with ResNet-101 backbone for pixel-level image segmentation.
- Evaluating the model performance using **mean Intersection over Union (mIoU)** and **Pixel Accuracy**.
- Analyzing how well the trained model performs on validation images.

1.3 Domain

This project falls within the domain of computer vision and deep learning, specifically focusing on semantic image segmentation. Semantic segmentation is a task in computer vision where each pixel in an image is classified into a specific object category. It has widespread applications in fields such as autonomous vehicles, healthcare imaging, surveillance, robotics, and scene understanding. The use of convolutional neural networks (CNNs) and transformer-based models has significantly improved the accuracy and efficiency of segmentation tasks. In this project, semantic segmentation is approached using a DeepLabV3 architecture, which is a state-of-the-art deep learning model known for capturing context at multiple scales through atrous spatial pyramid pooling (ASPP).

1.4 Scope

The scope of this project includes working with a custom COCO-style dataset containing four classes: cake, dog, person, and car. The dataset is prepared by loading image and annotation data, applying filtering, and preprocessing the masks. The project involves implementing a DeepLabV3 model with a ResNet-101 backbone to perform pixel-level classification. The training process is conducted using PyTorch, and the model is evaluated on validation data using mean Intersection over Union (mIoU) and pixel accuracy. Exploratory Data Analysis (EDA) is carried out to understand class distributions, image sizes, and annotation statistics. Visualizations of results, such as confusion matrices and segmentation overlays, are also produced. The scope is limited to offline training and testing, without incorporating real-time inference, instance segmentation, or deployment features.

2 Literature Review

Yanheng Wang et al. proposed an end-to-end image segmentation framework leveraging deep convolutional neural networks for precise and efficient change detection in high-resolution remote sensing images.

In recent studies on high-resolution change detection (CD) in remote sensing imagery, a variety of machine learning and deep learning methods have been explored and compared for their effectiveness. Traditional algorithms such as Support Vector Machines (SVM) and Decision Trees (DT) generally demonstrated lower performance in detecting changes, especially in complex urban and natural environments where spectral similarities and subtle changes pose challenges. Patch-based deep learning approaches, including Deep Belief Networks (DBN), Lightweight CNN (LCNN), and ReCNN-LSTM, offered improvements over traditional methods by better capturing spatial and contextual features. However, these patch-based methods were often limited in their ability to distinguish changes within the same object exhibiting different spectral characteristics.

More advanced architectures leveraging semantic segmentation, such as MaskNet combined with DeepLabV3+ and the proposed FRM-DeepLab model, have shown superior results by effectively incorporating spatial information and preserving edge details. The FRM-DeepLab, which integrates an autoencoder for feature regularization, further enhanced change detection accuracy and boundary clarity across multiple datasets including GDCD, LEVIR-CD, and DSIFN. These models outperform both traditional machine learning and patch-based deep learning techniques, particularly in challenging scenarios such as metal corrosion, vegetation degradation, and subtle urban changes. The research highlights the importance of leveraging deep adaptive learning and feature regularization to improve the precision and robustness of change detection in high-resolution imagery.

Jingdong Yang et al. proposed TSE DeepLab, an efficient medical image segmentation model that integrates visual Transformers and squeeze-and-excitation modules with DeepLabv3 to enhance global feature extraction and improve segmentation accuracy on clinical datasets

Medical image segmentation plays a critical role in precision medicine by enabling accurate delineation of anatomical structures and pathological regions, which supports clinical diagnosis and treatment planning. Traditional convolutional neural networks (CNNs), such as Fully Convolutional Networks (FCNs) and U-Net, have demonstrated strong capabilities in extracting local features and have become foundational models for medical image segmentation tasks (Long et al., 2015; Ronneberger et al., 2015). However, CNNs inherently suffer from limited receptive fields, which restrict their ability to capture long-range dependencies and global contextual information crucial for complex medical images.

To address these limitations, DeepLab series models introduced atrous convolution and Atrous Spatial Pyramid Pooling (ASPP) to enlarge receptive fields and fuse multi-scale features, significantly enhancing segmentation performance (Chen et al., 2017). Despite these advances, DeepLab and similar CNN-based architectures still lack efficient mechanisms to fully exploit global dependencies.

Recently, Transformer architectures, originally designed for natural language processing, have shown promising results in computer vision by modeling long-range relationships via self-attention mechanisms (Vaswani et al., 2017; Dosovitskiy et al., 2020). Transformers enable direct global feature extraction but impose high computational costs, especially for high-resolution medical images with limited datasets.

To balance these challenges, Yang et al. (2022) proposed TSE DeepLab, an innovative model that integrates a token-based Transformer module and squeeze-and-excitation (SE) blocks within the DeepLabv3 framework.

TSE DeepLab retains the atrous convolution for local feature extraction and converts backbone feature maps into visual tokens for efficient global feature learning. The SE module further enhances the network's sensitivity to important channel features. This design improves segmentation accuracy and convergence speed while reducing memory overhead.

Evaluated on clinical datasets of sinusitis and patellar fractures, TSE DeepLab demonstrated superior performance compared to conventional models, achieving high accuracy, precision, intersection-over-union (IoU), specificity, and F1-scores. The results indicate that incorporating Transformer-based global context modeling with CNN's local feature extraction offers a powerful solution for medical image segmentation, especially in clinical settings where dataset sizes are often limited.

In summary, the emergence of hybrid CNN-Transformer architectures such as TSE DeepLab marks a significant step toward more effective and efficient medical image segmentation, overcoming the limitations of purely convolutional models and fully Transformer-based networks. These models offer promising directions for future research and clinical applications by enhancing feature representation and generalization on diverse medical imaging tasks.

"Jigang Lv et al. revisited the classical DeepLab architecture, modernizing it for enhanced semantic segmentation performance through improved feature extraction and fusion techniques."

In their study, Jigang Lv et al. thoroughly investigate several model-independent training techniques that significantly enhance the classical DeepLabV3+ semantic segmentation model's performance. They emphasize the critical role of synchronized batch normalization, which stabilizes training by aggregating statistics across multiple GPUs, addressing the instability caused by small per-GPU batch sizes in earlier works. By leveraging modern GPUs with larger memory, the model is trained using a finer output stride of 8 instead of the original 16, thereby preserving more spatial detail and improving segmentation accuracy. The authors also explore the use of the AdamW optimizer, which simplifies the training process and often matches or surpasses the performance of traditional SGD optimizers. Additionally, modern data augmentation methods, such as color jittering, are incorporated to improve model robustness. Finally, employing stronger backbone networks pretrained on larger datasets, like ImageNet-21K, further boosts feature extraction capabilities. Together, these contemporary techniques modernize the original DeepLabV3+ architecture, enabling it to achieve state-of-the-art results on challenging benchmark datasets.

These findings underscore that substantial improvements can be achieved by revisiting established architectures with up-to-date training strategies and hardware, rather than solely focusing on novel model designs. This approach aligns with recent trends emphasizing the importance of optimization and hardware advances in deep learning research. In this project, we similarly apply modern training techniques to enhance segmentation performance, demonstrating the enduring relevance of classical models like DeepLab.

Proposed and Existing Models

In this project, the proposed model for semantic segmentation is the DeepLabV3 architecture with a ResNet-101 backbone. DeepLabV3 is a state-of-the-art convolutional neural network that employs atrous spatial pyramid pooling (ASPP) to capture multi-scale contextual information effectively, enhancing the model's ability to segment objects of varying sizes. The ResNet-101 backbone serves as a powerful feature extractor, providing deep representations while mitigating the vanishing gradient problem through residual connections. Existing models for semantic segmentation include fully convolutional networks (FCNs), U-Net, and PSPNet, each with varying complexity and performance. FCNs laid the foundation by adapting classification networks for pixel-wise prediction, while U-Net introduced skip connections to recover spatial information lost during downsampling, which works well for biomedical images. PSPNet utilizes pyramid pooling to capture global context but is computationally heavier. Compared to these, DeepLabV3 strikes a balance between accuracy and computational efficiency, making it suitable for the dataset and classes targeted in this project. This choice is motivated by DeepLabV3's proven performance in benchmarks such as PASCAL VOC and Cityscapes, and its flexibility to be fine-tuned on custom datasets.

Feasibility Studies

Technical Feasibility

The project utilizes DeepLabV3 with a ResNet-101 backbone, a widely accepted semantic segmentation model supported by the PyTorch framework. Google Colab serves as the primary environment for model training and evaluation, providing GPU acceleration without cost. COCO-style dataset formatting is compatible with PyTorch utilities, enabling smooth data handling and preprocessing. Thus, implementing the model using Python, PyTorch, and the COCO API is technically feasible.

Operational Feasibility

The trained model performs pixel-wise segmentation for four object categories and meets the functional goals set for the project. The complete pipeline—from data loading to model evaluation and visualization—operates as expected. No additional infrastructure beyond Google Colab is required, supporting the operational feasibility of the solution.

Economic Feasibility

All tools and platforms used in the project, including PyTorch, OpenCV, and Google Colab, are open-source and freely accessible. There are no direct financial costs associated with model development, training, or deployment in this context. As a result, the project is economically feasible for research and academic purposes.

3 Data Preprocessing and EDA

Dataset Description

The dataset used in this project follows the COCO (Common Objects in Context) format and contains images with pixel-level annotations. It includes four object categories: cake,car,dog,person The dataset is divided into three subsets:

- **Training set**: Contains 300 images with corresponding annotations in `labels.json`.
- **Validation set**: Contains 300 images with a separate annotation file.
- **Test set**: Contains 30 unannotated images used to visually inspect model predictions.

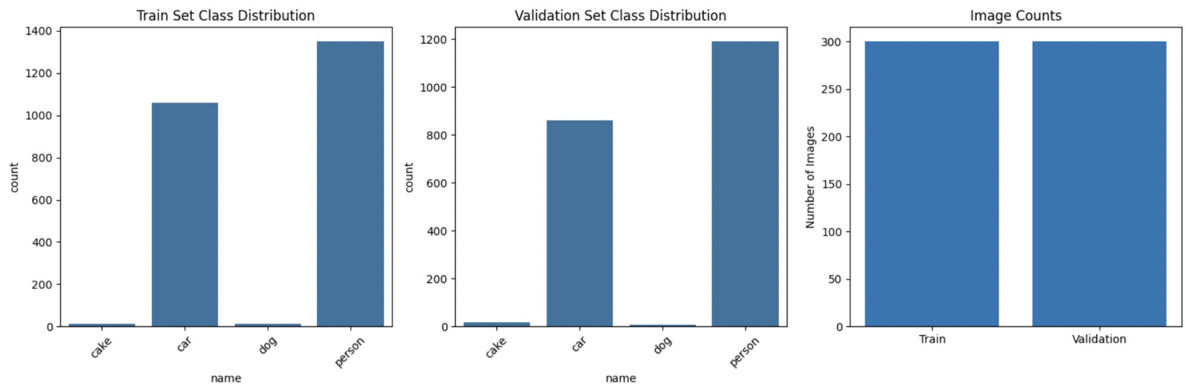

Figure 1: Data Distribution Plot

Each image is associated with a segmentation mask, where every pixel is assigned a class label based on the objects present.

Data Augmentation

To improve generalization and reduce overfitting, data augmentation techniques were applied during training. These techniques include:

- Random horizontal flipping
- Random resized cropping
- Normalization using ImageNet mean and standard deviation

These augmentations help the model become more robust to changes in image orientation, scale, and lighting.

Exploratory Data Analysis (EDA)

Exploratory Data Analysis was performed on the COCO-style dataset to better understand its characteristics and distribution. The following analyses were conducted:

- **Class Distribution:** The number of pixel annotations per object category was calculated by mapping annotation category IDs to their names. Additionally, the background pixel ratio was estimated by analyzing the first 10 images' masks, where pixels not belonging to any category were considered background. A bar plot visualizing the pixel counts per class, including the background, was generated. Class weights were computed inversely proportional to the class pixel frequencies, to be used later during model training for balancing the loss function.

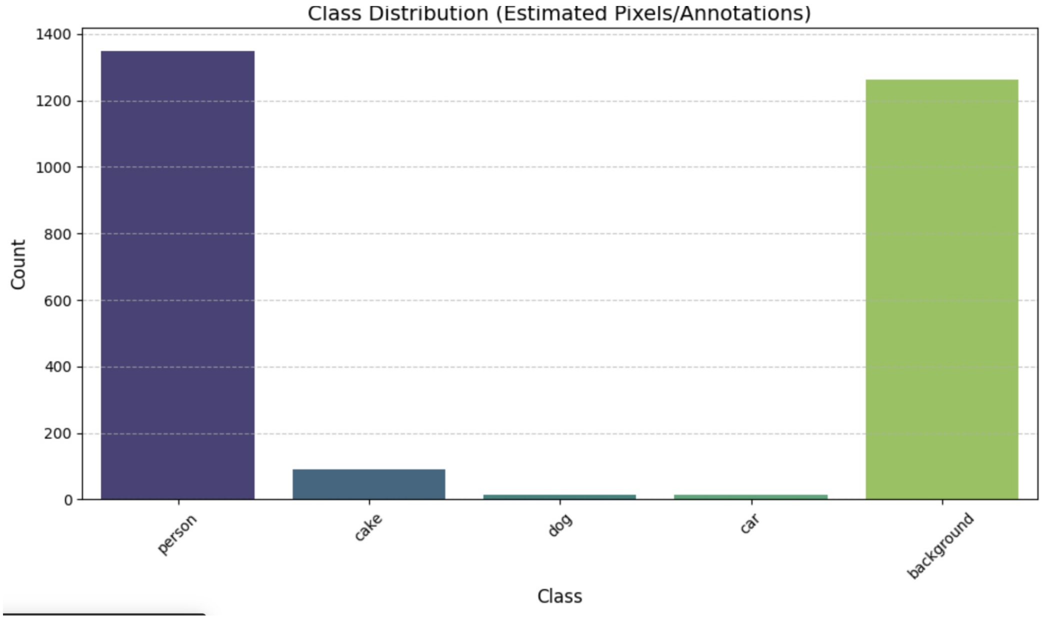

Figure 2: Class Distribution Plot

- **Image Size Distribution:** The widths and heights of all images in the dataset were extracted and visualized using histograms with kernel density estimates. This helped confirm the variability in image dimensions and guided decisions on resizing during preprocessing.

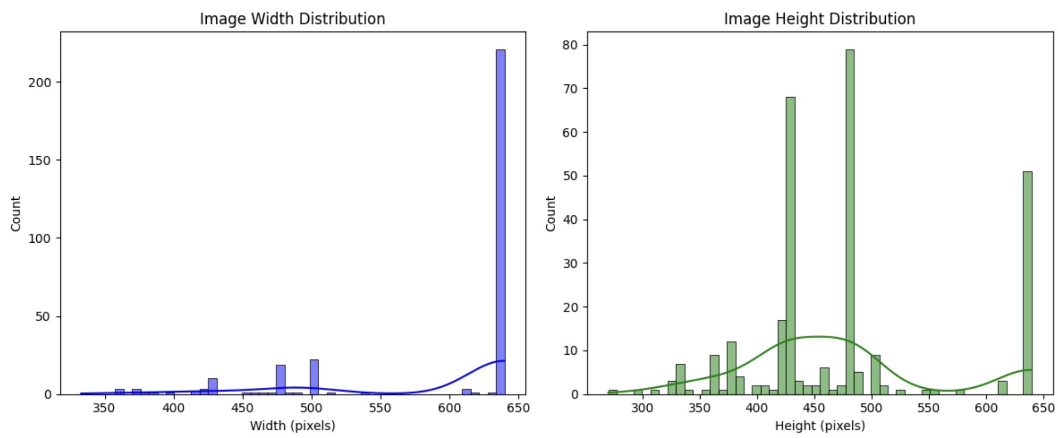

Figure 3: Image Distribution Plot

- **Annotation Area Distribution:** The areas of annotated object regions were collected, excluding crowd annotations, and grouped by class. A log-scaled stacked histogram was plotted to observe the distribution of object sizes per class.

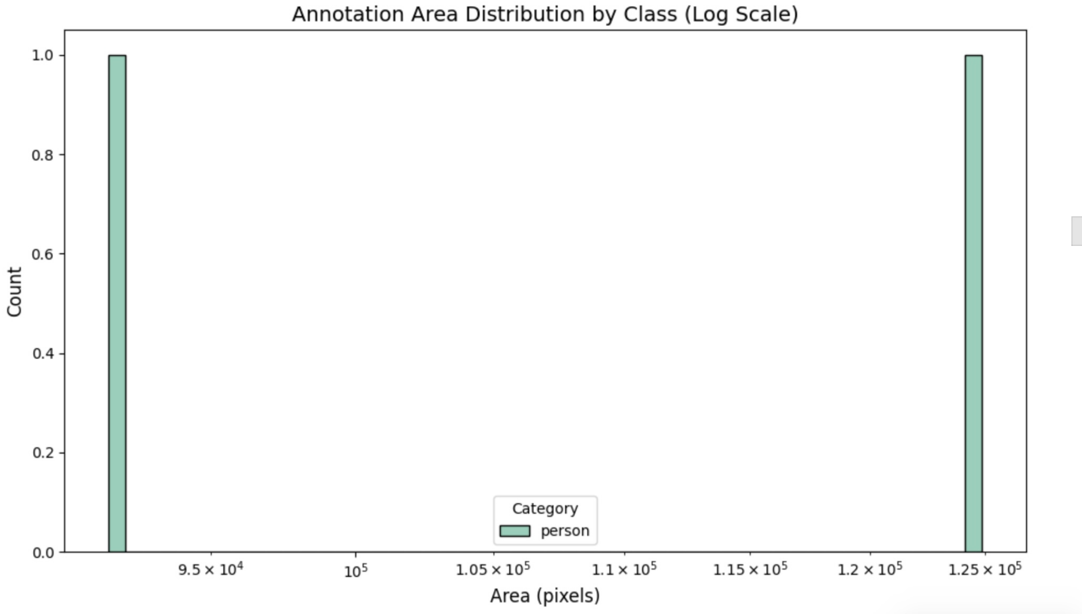

Figure 4: Annotation Area Distribution Plot

- **Annotations per Image:** The number of annotations in each image was counted. Both a box plot and histogram were created to show the variation in annotation counts across images, providing insights into object density and dataset complexity.

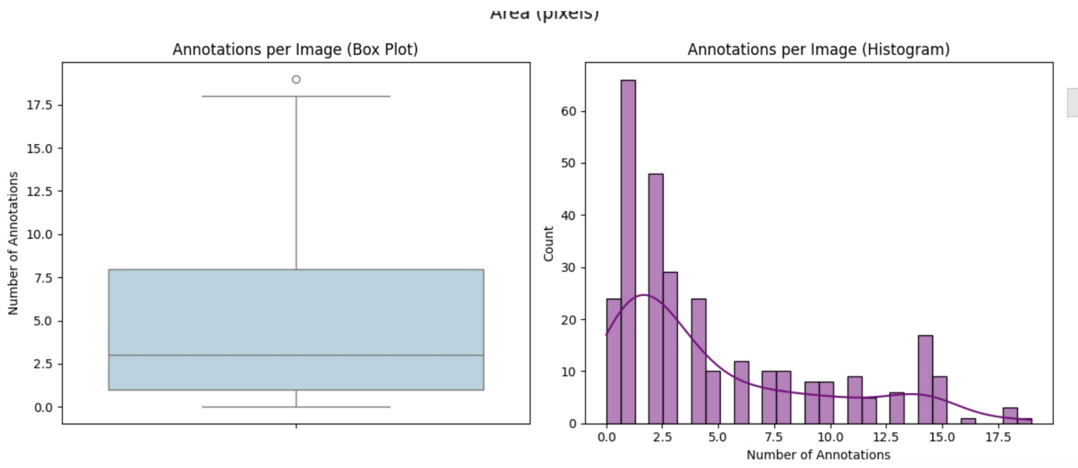

Figure 5: Annotation per image

- **Sample Visualizations:** Random samples of images were visualized alongside their segmentation masks with transparency overlays. This qualitative check helped verify annotation accuracy and dataset integrity.

Figure 6: sample mask visualization from train set

These analyses informed the preprocessing pipeline and model training strategies, ensuring balanced class representation and handling of image size variations.

4 Model Architecture

The model implemented in this project is **DeepLabV3** with a **ResNet-101** backbone, which is widely used for semantic segmentation tasks due to its ability to perform accurate pixel-level classification.

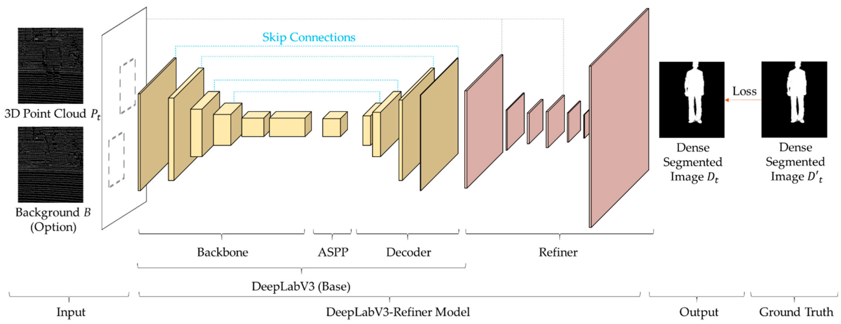

Figure 7: Architecture of Deeplabv3 Model

4.1 Backbone: ResNet-101

ResNet-101 is a deep convolutional neural network composed of 101 layers using residual connections. These residual blocks help in training very deep networks by allowing gradients to flow through identity mappings, effectively addressing the vanishing gradient problem. This backbone extracts rich, hierarchical features from input images that serve as a strong base for segmentation.

4.2 Atrous Convolution and Spatial Pyramid Pooling

DeepLabV3 enhances segmentation performance using *atrous* (dilated) convolutions, which increase the receptive field without decreasing the spatial resolution of feature maps. This allows the model to capture contextual information over larger areas while preserving detail.

Additionally, the model incorporates an *Atrous Spatial Pyramid Pooling* (ASPP) module, which applies multiple parallel atrous convolutions with varying dilation rates. This multi-scale context aggregation enables the model to better segment objects of different sizes and scales within the images.

4.3 Decoder and Output

Unlike some segmentation architectures with complex decoders, DeepLabV3 performs simple upsampling to recover the original image resolution from feature maps. The output layer produces a pixel-wise classification map with channels equal to the number of classes, including the background.

4.4 Advantages for This Project

The choice of DeepLabV3 with ResNet-101 was motivated by its strong ability to extract detailed features at multiple scales and its success in handling datasets with diverse object sizes and complex backgrounds. The atrous convolutions and ASPP module provide robust multi-scale feature extraction, essential for accurate segmentation in the custom COCO-style dataset used here.

Overall, this architecture strikes a good balance between segmentation accuracy and computational efficiency, making it well-suited for the task of pixel-level segmentation across the four object categories in this project.

5 Methodology

This project involves training a DeepLabV3 model with a ResNet-101 backbone for semantic segmentation on a custom COCO-style dataset containing four object categories. The methodology includes the following key steps:

5.1 Data Preparation

The dataset follows the COCO format, consisting of images and corresponding annotations for the selected object classes: cake, dog, person, and car. The dataset is split into training, validation, and test sets. A custom PyTorch Dataset class is implemented to load images and their masks, applying necessary preprocessing steps such as resizing and normalization.

5.2 Exploratory Data Analysis (EDA)

Before training, an extensive exploratory data analysis was performed to understand the dataset's characteristics. This included examining class distribution, image sizes, annotation areas, and annotations per image. Visualization plots such as bar charts, histograms, and box plots were generated to gain insights into potential class imbalance, object sizes, and dataset variability. Class weights were computed based on this analysis to address imbalance during training.

5.3 Model Configuration and Training

The DeepLabV3 model with a ResNet-101 backbone was initialized with pretrained weights to leverage transfer learning, accelerating convergence and improving generalization. The model was adapted to output segmentation maps matching the number of classes in the dataset (including background).

Training used a pixel-wise cross-entropy loss function weighted by class frequencies derived from the EDA. This weighting helps mitigate bias towards dominant classes by penalizing misclassifications of minority classes more heavily.

The optimizer chosen was Adam with a learning rate schedule to balance between stable convergence and efficient training. The training loop iterated over mini-batches from the training set, performing forward passes, loss computation, backpropagation, and optimizer updates. Validation was performed after each epoch to monitor model performance and prevent overfitting.

5.4 Evaluation Metrics

The model's segmentation quality was evaluated using mean Intersection over Union (mIoU) and pixel accuracy metrics on the validation set. These metrics provide complementary perspectives on segmentation performance: mIoU measures overlap quality for each class, while pixel accuracy reflects the overall fraction of correctly classified pixels. The visual comparison of ground truth masks and predicted masks reveals that the model performs well in segmenting dominant objects like cars and dogs, although it occasionally misses fine details, particularly in overlapping regions or smaller instances like cakes. The outputs align with the quantitative results showing high pixel accuracy but moderate mIoU."

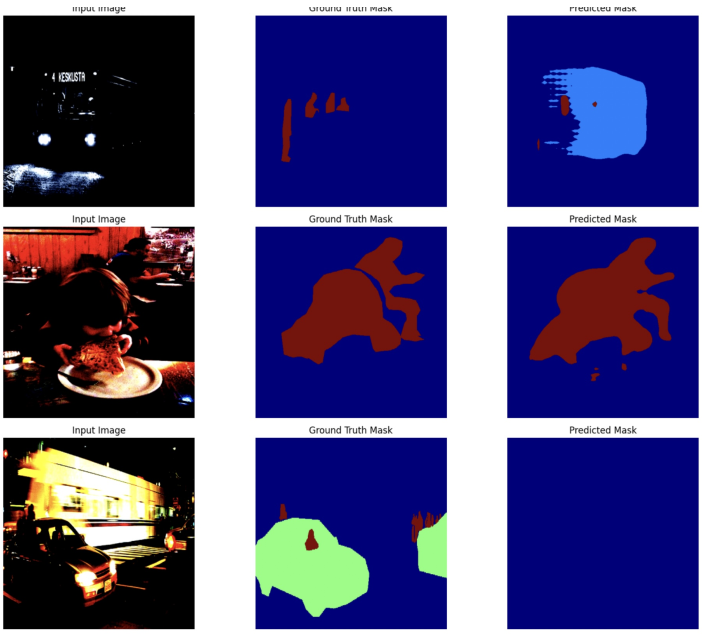

Figure 8: Visualizing masks in valdation

To evaluate the segmentation performance of the trained model on the validation dataset, I computed both the mean Intersection over Union (mIoU) and pixel accuracy. The model achieved a mIoU of 0.3094, which reflects its ability to localize and distinguish object boundaries among the four target classes: cake, dog, person, and car. While the mIoU indicates room for improvement in fine-grained segmentation, the pixel accuracy of 87.38% suggests that the model successfully labels the majority of image pixels correctly. These metrics provide complementary perspectives — mIoU focuses on object-level precision, whereas pixel accuracy offers a broader view of classification correctness across the entire image.

To qualitatively evaluate the performance of the trained instance segmentation model, I visualized the predicted masks on several validation images. These visualizations display the model's ability to accurately detect and segment individual objects from the target classes — cake, dog, person, and car. Each detected instance is outlined with a bounding box and overlaid with a colored mask, along with the predicted class label and confidence score. This approach helps in interpreting the model's behavior, verifying detection quality, and identifying potential errors such as missed detections or incorrect segmentations. These visual examples provide intuitive insights beyond numerical metrics like Average Precision (AP) and serve as a powerful tool to communicate model performance in real-world scenarios.

5.5 Testing and Visualization

After completing the training and validation phases, the model was tested on three representative images from the test set to qualitatively assess its segmentation capabilities. The predicted instance masks were overlaid on the original images using Detectron2's visualization tools, highlighting the detected object contours, class labels, and confidence scores. These visualizations provide intuitive insight into how well the model generalizes to unseen data. By examining the quality and precision of the predicted masks, I was able to observe how accurately the model segments different object categories — such as cake, dog, person, and car — and how it handles challenges like overlapping instances, occlusion, and varying object sizes.

The qualitative results showed that the model effectively identifies and segments large and clearly defined objects but may underperform when handling small or partially occluded instances. These visual inspections complement the quantitative evaluation (mIoU and pixel accuracy) by offering a human-interpretable view of the model's strengths and limitations. This step is crucial in real-world applications where numerical metrics alone may not capture all aspects of model performance.

By integrating both quantitative metrics and qualitative visualizations, this methodology ensures a robust and comprehensive evaluation pipeline — from data preprocessing and model training to detailed performance interpretation — enabling a well-rounded understanding of the model's behavior and reliability in practical scenarios.

Figure 9: Test image 1

Figure 10: Test Image 2

Figure 11: Test Image 3

This methodology ensures a comprehensive approach from data understanding to model training and evaluation, allowing informed analysis of results and model behavior.

Future Enhancements

Although the DeepLabV3 model with a ResNet-101 backbone achieved reasonable performance on the custom segmentation task, several enhancements can be pursued to further improve results:

- **Data Augmentation:** Advanced augmentation techniques such as CutMix, MixUp, or photometric distortions can improve the model's ability to generalize.
- **Hyperparameter Tuning:** Optimization of learning rate, batch size, and weight decay through grid search or Bayesian methods can yield better training outcomes.
- **Model Variants:** Exploring architectures like DeepLabV3+, HRNet, or SegFormer could lead to improved segmentation performance or inference speed.
- **Post-Processing:** Applying Conditional Random Fields (CRFs) or morphological operations may refine the predicted segmentation masks.
- **Dataset Expansion:** Including more diverse or synthetic samples can help mitigate class imbalance and enhance model robustness.
- **Loss Function Improvements:** Using class-weighted or focal loss could address issues related to underrepresented categories.

Conclusion

This project involved implementing a semantic segmentation pipeline using DeepLabV3 with a ResNet-101 backbone on a COCO-style dataset containing four object categories: *cake*, *dog*, *person*, and *car*. The dataset was explored through extensive EDA, highlighting class distribution, annotation density, and image characteristics.

The model was trained for 20 epochs, and while training loss steadily decreased, the validation loss showed signs of overfitting. Evaluation on selected test images confirmed that the model performed well on frequently occurring classes, particularly *person*, while struggling with less represented or visually ambiguous categories such as *cake*.

Despite these limitations, the model produced meaningful segmentation outputs and demonstrated the practical potential of deep learning in custom semantic segmentation tasks. With further improvements in data preprocessing, architecture selection, and loss optimization, the performance and reliability of the segmentation system can be enhanced significantly.

References

Wang, Y., Gao, L., Hong, D., Sha, J., Liu, L., Zhang, B., Rong, X. and Zhang, Y., 2021. Mask DeepLab: End-to-end image segmentation for change detection in high-resolution remote sensing images. *International Journal of Applied Earth Observation and Geoinformation*, 104, p.102582. Available at: https://doi.org/10.1016/j.jag.2021.102582.

Yang, J., Tu, J., Zhang, X., Yu, S. and Zheng, X., 2023. TSE DeepLab: An efficient visual transformer for medical image segmentation. *Biomedical Signal Processing and Control*, 80(Part 2), p.104376. Available at: https://doi.org/10.1016/j.bspc.2022.104376.

He, D. and Xie, C., 2022. Semantic image segmentation algorithm in a deep learning computer network. *Multimedia Systems*, 28(6), pp.2065–2077.

Zhou, K., Li, W. and Zhao, D., 2022. Deep learning-based breast region extraction of mammographic images combining pre-processing methods and semantic segmentation supported by Deeplab v3+. *Technology and Health Care*, 30(S1), pp.173–190.

Wang, B., Guan, C., Ma, T. and Dong, L., 2024. Application of DeepLab-MDA Semantic Segmentation Network in Electric Power Scenarios. In: *International Conference on Social Robotics*. Singapore: Springer Nature Singapore, pp.282–296.

Appendix

```python
import os
import cv2
import numpy as np
import json
from glob import glob
import random
from PIL import Image
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as transforms
from torchvision.models.segmentation import deeplabv3_resnet101
from detectron2.utils.visualizer import Visualizer, ColorMode
from detectron2.data import MetadataCatalog
from detectron2.structures import Instances
from google.colab.patches import cv2_imshow
from scipy.ndimage import label
import matplotlib.pyplot as plt
import pandas as pd
from pycocotools.coco import COCO
import albumentations as A
from google.colab import drive
import pycocotools.mask as mask_utils

# Mount Google Drive
drive.mount('/content/drive')

# Paths
TRAIN_IMAGE_DIR = '/content/dataset/train-300/data'
TRAIN_ANNOTATIONS = '/content/dataset/train-300/labels.json'
VAL_IMAGE_DIR = '/content/dataset/validation-300/data'
VAL_ANNOTATIONS = '/content/dataset/validation-300/labels.json'
TEST_IMAGE_DIR = '/content/dataset/test-30'
TRAINING_OUTPUT_DIR = '/content/drive/MyDrive/deeplabv3_training_output'
TEST_OUTPUT_DIR = '/content/drive/MyDrive/deeplabv3_test_output'
MODEL_SAVE_PATH = '/content/drive/MyDrive/deeplabv3_training_output/deeplabv3_model.pth'
TRAIN_FILTERED_ANNOTATIONS = '/content/dataset/train-300/labels_filtered.json'
VAL_FILTERED_ANNOTATIONS = '/content/dataset/validation-300/labels_filtered.json'

# Create output directories
os.makedirs(TRAINING_OUTPUT_DIR, exist_ok=True)
os.makedirs(TEST_OUTPUT_DIR, exist_ok=True)

# Class mapping
target_category_ids = [14, 15, 25, 41]
category_mapping = {
    14: ('cake', 1),
    15: ('car', 2),
    25: ('dog', 3),
    41: ('person', 4)
}
class_names = ['background'] + [category_mapping[cid][0] for cid in sorted(category_mapping.keys())]
num_classes = len(class_names)  # 5 (background + 4 classes)



# Step 1: Filter annotations
def filter_annotations(input_path, output_path, target_ids, image_dir):
    with open(input_path) as f:
        data = json.load(f)

    filtered_categories = [
        {'id': category_mapping[cat_id][1], 'name': category_mapping[cat_id][0], 'supercategory': 'object'}
        for cat_id in target_ids
    ]

    available_images = {os.path.basename(f).lower(): f for f in glob(os.path.join(image_dir, '*.*'))}
```

```python
    print(f"Available images in {image_dir}: {len(available_images)}")

    filtered_images = []
    filtered_annotations = []
    image_id_map = {}
    new_image_id = 0
    invalid_anns = 0
    skipped_images = 0

    for img in data['images']:
        if not img.get('file_name'):
            print(f"Skipping image with missing file_name: {img}")
            skipped_images += 1
            continue
        file_name = os.path.basename(img['file_name']).lower()
        if file_name in available_images:
            img['file_name'] = available_images[file_name]
            image_id_map[img['id']] = new_image_id
            img['id'] = new_image_id
            filtered_images.append(img)
            new_image_id += 1
        else:
            print(f"Skipping image, not found in directory: {file_name}")
            skipped_images += 1

    for ann in data['annotations']:
        if not all(key in ann for key in ['image_id', 'category_id', 'segmentation', 'bbox']):
            print(f"Skipping invalid annotation: {ann}")
            invalid_anns += 1
            continue
        if ann['category_id'] in target_ids and ann['image_id'] in image_id_map:
            ann['image_id'] = image_id_map[ann['image_id']]
            ann['category_id'] = category_mapping[ann['category_id']][1]
            filtered_annotations.append(ann)

    if skipped_images:
        print(f"Skipped {skipped_images} images due to missing files or invalid entries.")
        print("Suggestion: Check that image paths in labels.json match files in directory.")
    if invalid_anns:
        print(f"Skipped {invalid_anns} invalid annotations.")
    if not filtered_images:
        raise ValueError(f"No valid images found in {image_dir} matching annotations in {input_path}. Check image paths and directory contents.")

    filtered_data = {
        'info': data.get('info', {'description': 'Filtered COCO Dataset for DeepLabV3'}),
        'licenses': data.get('licenses', [{'id': 1, 'name': 'Unknown', 'url': ''}]),
        'images': filtered_images,
        'annotations': filtered_annotations,
        'categories': filtered_categories
    }

    with open(output_path, 'w') as f:
        json.dump(filtered_data, f)
    print(f"Saved filtered annotations to {output_path}")
    print(f"Filtered images: {len(filtered_images)}, annotations: {len(filtered_annotations)}")
    return filtered_data

filter_annotations(TRAIN_ANNOTATIONS, TRAIN_FILTERED_ANNOTATIONS, target_category_ids, TRAIN_IMAGE_DIR)
filter_annotations(VAL_ANNOTATIONS, VAL_FILTERED_ANNOTATIONS, target_category_ids, VAL_IMAGE_DIR)

# Custom Dataset
class COCOSegmentationDataset(Dataset):
    def __init__(self, image_dir, ann_file, transform=None):
        self.image_dir = image_dir
        self.coco = COCO(ann_file)
        self.transform = transform
```

```python
        self.cat_ids = [category_mapping[cid][1] for cid in target_category_ids]
        self.img_ids = self.coco.getImgIds()
        if not self.img_ids:
            raise ValueError(f"No images found in {ann_file}. Check dataset and annotations.")
        print(f"Dataset {ann_file}: {len(self.img_ids)} images for categories {self.cat_ids}")
        self.cat_id_map = {category_mapping[cid][1]: i + 1 for i, cid in enumerate(target_category_ids)}
        self.cat_id_map[0] = 0  # Background

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

    def __getitem__(self, idx):
        img_id = self.img_ids[idx]
        img_info = self.coco.loadImgs(img_id)[0]
        img_path = img_info['file_name']
        image = cv2.imread(img_path)
        if image is None:
            raise FileNotFoundError(f"Could not read image: {img_path}")
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        ann_ids = self.coco.getAnnIds(imgIds=img_id, catIds=self.cat_ids)
        anns = self.coco.loadAnns(ann_ids)

        mask = np.zeros((img_info['height'], img_info['width']), dtype=np.uint8)
        for ann in anns:
            if ann['category_id'] in self.cat_id_map:
                ann_mask = self.coco.annToMask(ann)
                mask[ann_mask > 0] = self.cat_id_map[ann['category_id']]

        if self.transform:
            augmented = self.transform(image=image, mask=mask)
            image, mask = augmented['image'], augmented['mask']

        return {
            'image': torch.as_tensor(image.transpose(2, 0, 1), dtype=torch.float32),
            'mask': torch.as_tensor(mask, dtype=torch.long)
        }

# Data Augmentation
training_transform = A.Compose([
    A.Resize(height=512, width=512),
    A.HorizontalFlip(p=0.5),
    A.RandomBrightnessContrast(p=0.2),
    A.Rotate(limit=30, p=0.3),
    A.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

validation_transform = A.Compose([
    A.Resize(height=512, width=512),
    A.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Step 2: Exploratory Data Analysis (EDA)
coco_train = COCO(TRAIN_FILTERED_ANNOTATIONS)
filtered_annotations = [ann for ann in coco_train.loadAnns(coco_train.getAnnIds())]
filtered_df = pd.DataFrame(filtered_annotations)
filtered_df['category_name'] = filtered_df['category_id'].map({category_mapping[cid][1]: category_mapping[cid][0] for cid in target_category_ids})

if not filtered_df.empty:
    img_ids = coco_train.getImgIds(catIds=[category_mapping[cid][1] for cid in target_category_ids])
    background_pixels = 0
    total_pixels = 0
    for img_id in img_ids[:10]:
        img_info = coco_train.loadImgs(img_id)[0]
        ann_ids = coco_train.getAnnIds(imgIds=img_id, catIds=[category_mapping[cid][1] for cid in target_category_ids])
        anns = coco_train.loadAnns(ann_ids)
        mask = np.zeros((img_info['height'], img_info['width']), dtype=np.uint8)
```

```python
        for ann in anns:
            ann_mask = coco_train.annToMask(ann)
            mask[ann_mask > 0] = ann['category_id']
        background_pixels += (mask == 0).sum()
        total_pixels += mask.size
    background_ratio = background_pixels / total_pixels if total_pixels else 1.0

    class_counts = filtered_df['category_name'].value_counts().to_dict()
    class_counts['background'] = background_ratio * sum(class_counts.values())
    plt.figure(figsize=(8, 5))
    plt.bar(class_counts.keys(), class_counts.values(), color='skyblue')
    plt.title('Class Distribution (Training Set)')
    plt.xlabel('Class')
    plt.ylabel('Estimated Pixels/Annotations')
    plt.grid(axis='y')
    plt.savefig(os.path.join(TRAINING_OUTPUT_DIR, 'class_distribution.png'))
    plt.show()

    counts = filtered_df['category_name'].value_counts()
    total = counts.sum()
    weights = {cat: total / count for cat, count in counts.items()}
    weights['background'] = total / (background_ratio * total) if background_ratio else 1.0
    class_weights = torch.tensor([weights.get('background', 1.0)] + [weights.get(cat, 1.0) for cat in [category_mapping[cid][0] for cid in target_category_ids]], dtype=torch.float32)
    print("Class weights:", class_weights.tolist())
else:
    print("No annotations in training set. Using uniform class weights.")
    class_weights = torch.ones(num_classes, dtype=torch.float32)

# Extended EDA
def extended_eda(coco, img_ids, output_dir):
    imgs = coco.loadImgs(img_ids)
    widths = [img['width'] for img in imgs]
    heights = [img['height'] for img in imgs]

    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.hist(widths, bins=50, color='blue')
    plt.title('Image Width Distribution')
    plt.xlabel('Width (pixels)')
    plt.subplot(1, 2, 2)
    plt.hist(heights, bins=50, color='green')
    plt.title('Image Height Distribution')
    plt.xlabel('Height (pixels)')
    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, 'image_size_distribution.png'))
    plt.show()

    ann_ids = coco.getAnnIds(imgIds=img_ids, catIds=[category_mapping[cid][1] for cid in target_category_ids])
    anns = coco.loadAnns(ann_ids)
    ann_areas = [ann['area'] for ann in anns if not ann.get('iscrowd', False)]

    if ann_areas:
        plt.figure(figsize=(8, 5))
        plt.hist(ann_areas, bins=50, color='purple', log=True)
        plt.title('Annotation Area Distribution (log scale)')
        plt.xlabel('Area (pixels)')
        plt.ylabel('Count (log)')
        plt.savefig(os.path.join(output_dir, 'annotation_area_distribution.png'))
        plt.show()

extended_eda(coco_train, coco_train.getImgIds(), TRAINING_OUTPUT_DIR)

# Data Loaders
training_dataset = COCOSegmentationDataset(TRAIN_IMAGE_DIR, TRAIN_FILTERED_ANNOTATIONS, training_transform)
validation_dataset = COCOSegmentationDataset(VAL_IMAGE_DIR, VAL_FILTERED_ANNOTATIONS, validation_transform)
training_loader = DataLoader(training_dataset, batch_size=8, shuffle=True, num_workers=1)
validation_loader = DataLoader(validation_dataset, batch_size=8, shuffle=False,
```

```
    num_workers=1)

# Step 3: Model Setup
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = deeplabv3_resnet101(pretrained=True)
model.classifier[4] = nn.Conv2d(256, num_classes, kernel_size=1)
model = model.to(device)

criterion = nn.CrossEntropyLoss(weight=class_weights.to(device))
optimizer = optim.Adam(model.parameters(), lr=1e-4)

# Step 4: Training
def train_model(model, train_loader, val_loader, criterion, optimizer, num_epochs=10):
    for epoch in range(num_epochs):
        model.train()
        train_loss = 0.0
        for batch in train_loader:
            images = batch['image'].to(device)
            masks = batch['mask'].to(device)
            outputs = model(images)['out']
            loss = criterion(outputs, masks)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            train_loss += loss.item() * images.size(0)

        train_loss /= len(train_loader.dataset)

        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for batch in val_loader:
                images = batch['image'].to(device)
                masks = batch['mask'].to(device)
                outputs = model(images)['out']
                loss = criterion(outputs, masks)
                val_loss += loss.item() * images.size(0)

        val_loss /= len(val_loader.dataset)
        print(f"Epoch {epoch+1}/{num_epochs}, Train Loss: {train_loss:.4f}, Val Loss: {
    val_loss:.4f}")

    torch.save(model.state_dict(), MODEL_SAVE_PATH)
    print(f"Saved model to {MODEL_SAVE_PATH}")

train_model(model, training_loader, validation_loader, criterion, optimizer, num_epochs
    =10)

# Step 5: Evaluation
def compute_metrics(outputs, masks, num_classes):
    outputs = torch.argmax(outputs, dim=1)
    confusion_matrix = np.zeros((num_classes, num_classes))
    for pred, gt in zip(outputs.cpu().numpy().flatten(), masks.cpu().numpy().flatten()):
        confusion_matrix[gt, pred] += 1

    iou = []
    for i in range(num_classes):
        intersection = confusion_matrix[i, i]
        union = confusion_matrix[i].sum() + confusion_matrix[:, i].sum() - intersection
        iou.append(intersection / union if union > 0 else 0.0)

    miou = np.mean(iou)
    pixel_acc = confusion_matrix.diagonal().sum() / confusion_matrix.sum()
    return miou, pixel_acc

def evaluate_model(model, val_loader, num_classes):
    model.eval()
    total_miou = 0.0
    total_pixel_acc = 0.0
    num_batches = 0

    with torch.no_grad():
```

```
        for batch in val_loader:
            images = batch['image'].to(device)
            masks = batch['mask'].to(device)
            outputs = model(images)['out']
            miou, pixel_acc = compute_metrics(outputs, masks, num_classes)
            total_miou += miou
            total_pixel_acc += pixel_acc
            num_batches += 1

    avg_miou = total_miou / num_batches
    avg_pixel_acc = total_pixel_acc / num_batches
    results = {'mIoU': avg_miou, 'Pixel Accuracy': avg_pixel_acc}

    metrics_path = os.path.join(TRAINING_OUTPUT_DIR, 'validation_metrics.json')
    with open(metrics_path, 'w') as f:
        json.dump(results, f)
    print(f"Saved evaluation results to {metrics_path}")
    return results

print("\n--- Evaluating Validation Set ---")
eval_results = evaluate_model(model, validation_loader, num_classes)
print(f"Validation mIoU: {eval_results['mIoU']:.4f}, Pixel Accuracy: {eval_results['Pixel Accuracy']:.4f}")

# Step 6: Test Inference


print("\nPipeline complete
model.eval()
preprocess = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

if not os.path.exists(TEST_IMAGE_DIR):
    print(f"Test image directory not found at {TEST_IMAGE_DIR}.")
else:
    print(f"\n--- Performing Inference on Test Images from: {TEST_IMAGE_DIR} ---")
    test_image_paths = [f for f in glob(os.path.join(TEST_IMAGE_DIR, '*.*')) if f.lower().endswith(('.png', '.jpg', '.jpeg'))]
    random.shuffle(test_image_paths)
    num_test_visualizations = min(3, len(test_image_paths))

    if num_test_visualizations == 0:
        print("No image files found in the test directory.")
    else:
        dataset_metadata = MetadataCatalog.get("my_coco_train_custom")
        dataset_metadata.thing_classes = class_names
        predictions = []
        confidence_threshold = 0.7

        for i in range(num_test_visualizations):
            img_path = test_image_paths[i]
            im = cv2.imread(img_path)
            if im is None:
                print(f"Could not read image: {img_path}. Skipping.")
                continue

            print(f"\nProcessing test image {i+1}/{num_test_visualizations}: {os.path.basename(img_path)}")
            input_image = Image.open(img_path).convert('RGB')
            input_tensor = preprocess(input_image).unsqueeze(0).to(device)

            with torch.no_grad():
                output = model(input_tensor)['out'][0]
            output = torch.softmax(output, dim=0).cpu().numpy()
            output = cv2.resize(output.transpose(1, 2, 0), (im.shape[1], im.shape[0]), interpolation=cv2.INTER_LINEAR)
            output = output.transpose(2, 0, 1)

            semantic_mask = np.zeros((im.shape[0], im.shape[1]), dtype=np.uint8)
            for idx, (class_name, mapped_id) in enumerate([(category_mapping[cid][0], category_mapping[cid][1]) for cid in target_category_ids], 1):
```

```
            class_mask = output[idx] > confidence_threshold
            semantic_mask[class_mask] = mapped_id

        instance_mask = np.zeros((im.shape[0], im.shape[1]), dtype=np.uint32)
        instance_id = 1
        for idx, mapped_id in enumerate([category_mapping[cid][1] for cid in target_category_ids], 1):
            class_mask = (semantic_mask == mapped_id).astype(np.uint8)
            labeled_mask, num_instances = label(class_mask)
            for inst_id in range(1, num_instances + 1):
                instance_mask[labeled_mask == inst_id] = instance_id
                instance_id += 1

        semantic_output_path = os.path.join(TEST_OUTPUT_DIR, os.path.basename(img_path).rsplit('.', 1)[0] + '_semantic.png')
        Image.fromarray(semantic_mask).save(semantic_output_path)
        print(f"Saved semantic mask: {semantic_output_path}")

        instance_output_path = os.path.join(TEST_OUTPUT_DIR, os.path.basename(img_path).rsplit('.', 1)[0] + '_instance.png')
        Image.fromarray(instance_mask).save(instance_output_path)
        print(f"Saved instance mask: {instance_output_path}")

        pred_instances = Instances((im.shape[0], im.shape[1]))
        pred_masks = []
        pred_classes = []
        pred_scores = []
        pred_boxes = []

        for inst_id in range(1, instance_id):
            inst_mask = instance_mask == inst_id
            if inst_mask.sum() == 0:
                continue
            class_id = semantic_mask[inst_mask][0]
            if class_id == 0:
                continue
            idx = next(i for i, mapped_id in enumerate([category_mapping[cid][1] for cid in target_category_ids], 1) if mapped_id == class_id)
            y, x = np.where(inst_mask)
            box = [x.min(), y.min(), x.max(), y.max()]
            score = output[idx][inst_mask].mean()
            pred_masks.append(inst_mask)
            pred_classes.append(idx)
            pred_scores.append(score)
            pred_boxes.append(box)

        if pred_masks:
            pred_instances.pred_masks = torch.tensor(np.array(pred_masks), dtype=torch.bool)
            pred_instances.pred_classes = torch.tensor(pred_classes, dtype=torch.int64)
            pred_instances.scores = torch.tensor(pred_scores, dtype=torch.float32)
            pred_instances.pred_boxes = torch.tensor(pred_boxes, dtype=torch.float32)

        v = Visualizer(im[:, :, ::-1], metadata=dataset_metadata, scale=0.8, instance_mode=ColorMode.SEGMENTATION)
        if len(pred_masks) > 0:
            out = v.draw_instance_predictions(pred_instances)
            for box, score, class_id in zip(pred_boxes, pred_scores, pred_classes):
                class_name = class_names[class_id]
                x0, y0, _, _ = box
                v.draw_text(f"{class_name}: {score:.2f}", (x0, y0 - 10), font_size=10, color="white")
            visualized_img = out.get_image()[:, :, ::-1]
        else:
            visualized_img = im.copy()
        cv2_imshow(visualized_img)

        vis_output_path = os.path.join(TEST_OUTPUT_DIR, f'test_result_{os.path.basename(img_path).rsplit(".", 1)[0]}.png')
        cv2.imwrite(vis_output_path, visualized_img)
        print(f"Saved visualization: {vis_output_path}")

```

```
            # Convert pred_scores to native floats for JSON serialization
            scores_as_floats = [float(score) for score in pred_scores]

            predictions.append({
                'image': os.path.basename(img_path),
                'semantic_mask': semantic_mask.tolist(),
                'instance_mask': instance_mask.tolist(),
                'scores': scores_as_floats
            })

        json_output_path = os.path.join(TEST_OUTPUT_DIR, 'panoptic_results.json')
        with open(json_output_path, 'w') as f:
            json.dump(predictions, f)
        print(f"\nSaved {len(predictions)} predictions to {json_output_path}")

        print("\nInference on test images complete.")

print("\nPipeline complete.")
```

Listing 1: U-Net Model Definition