COCO-Based Binary Mask Generation for Human Class

Task 1: Dataset Preparation using Python

Sanjot Sagar Totade

April 16, 2025

1 Github link

The entire project can be found at https://github.com/sanjot-sagar/vj$_t$*ech*

2 Introduction

In this report, we describe the preprocessing pipeline and rationale behind our choice of dataset and annotations for the rapid prototyping phase of Task 2 in the modeling track. The aim was to prepare binary segmentation masks for the "person" category using the COCO 2017 dataset to support early experimentation with segmentation and vision models.

3 Dataset Selection

We chose the **COCO 2017 validation set** (`val2017`) as our base dataset. This subset contains 5,000 images and provides a good trade-off between diversity and computational feasibility. Key reasons for this selection are:

- **Small size:** With only 5k images, the dataset is ideal for rapid prototyping.
- **Availability:** The dataset and its annotations are easily accessible from the official COCO dataset website.
- **Quality:** COCO provides high-quality annotations with detailed segmentation masks.

4 Annotation Filtering and Class Prioritization

To simplify the segmentation task, we focused only on the "`person`" class. This decision was motivated by our need to test downstream models quickly without the overhead of multi-class segmentation. Additionally, humans are one of the most frequently occurring and visually complex categories in COCO, making them a meaningful choice for early experimentation.

This subset was deemed sufficient for evaluating early-stage models and understanding segmentation behavior on a real-world dataset.

5 Mask Generation Pipeline

The segmentation masks were generated using the official `pycocotools` library. The core steps in the pipeline include:

1. Downloading and extracting the `val2017` images and annotation files using HTTP requests.
2. Parsing the annotation JSON using COCO API to get image IDs with the "person" category.
3. Sorting annotations by a priority order to resolve overlapping instances (if multiple categories were used).
4. Creating a binary segmentation mask per image, marking only the pixels corresponding to the "person" category.
5. Saving the mask as a PNG file with the same base name as the image.

For visualization, we added a utility that overlays the segmentation mask on the original image using color maps for qualitative inspection. This was used to validate the correctness of the generated masks.

6 Design Decisions and Justification

- **Category Selection:** We filtered the COCO annotations to retain only "person" instances to reduce complexity and support binary mask generation.
- **Mask Format:** The masks were saved in PNG format with integer pixel values (0 = background, 1 = person), which simplifies downstream model training.
- **Visualization:** To ensure quality control, we implemented a visualization function that overlays the segmentation masks on RGB images using colored overlays.
- **Reproducibility:** All steps were implemented using publicly available tools and the script was designed to be modular and reproducible.

7 Visualizations of Segmentation Masks

In this section, we show sample images from the dataset alongside their corresponding binary masks to illustrate the segmentation results.

Figure 1: Original Image 1

Figure 2: Segmentation Mask for Image 1

Figure 3: Original Image 2

Figure 4: Segmentation Mask for Image 2

8 Task 1 Conclusion

This preprocessing pipeline produces binary masks for the "person" class using COCO `val2017`, enabling quick iteration for segmentation model experiments. The decisions taken—such as dataset size, class filtering, and visualization—were made to prioritize speed, simplicity, and interpretability during the prototyping phase of Task 2.

All code used in this pipeline is modular and can be easily extended to include additional COCO classes or other datasets in future experiments.

Task 2: Fine-tuning Pre-trained Models for Downstream Task – Train an Image Segmentation Model

9 Training and Evaluation Pipeline

We implemented a training pipeline for semantic segmentation using PyTorch. The main goals were reproducibility, extensibility, and comparative evaluation of different segmentation models. The core features of the pipeline are summarized below:

9.1 Model Selection and Motivation

The models supported in our code include `UNet`, `DeepLabV3`, and `FastSCNN`. These models were selected to cover a range of architectures from lightweight real-time segmentation (FastSCNN) to more accurate and heavier networks (DeepLabV3).

9.2 Dataset and Dataloader

We used the `COCOVal2017` dataset for prototyping due to its manageable size of 5,000 images. Out of these, 2,693 images contain the human class. Binary masks were generated for this class, simplifying the segmentation task to a single foreground-background objective. This decision was guided by the need for rapid prototyping for a subsequent modeling task.

9.3 Training Procedure

Each model is trained using:

- **Loss Function:** Binary Cross-Entropy with Logits Loss (`BCEWithLogitsLoss`).
- **Optimizer:** Adam optimizer with a learning rate of 1×10^{-5}. We later tune this and this is a part of our main ablation table.
- **Epochs:** 20 epochs per model.
- **Batch Size:** 8.
- **Device:** Training is done on GPU if available. We in particular leverage Nvidia A4000 for our task.

Training logs are saved using TensorBoard for both training and validation metrics. Directory structures are automatically created with timestamps .

9.4 Metrics and Evaluation

The following evaluation metrics are computed on train, validation, and test sets:

- Dice Score
- Intersection over Union (IoU)
- Pixel Accuracy

An initial checkpoint is saved after the first batch to inspect early model behavior. The final model and evaluation metrics are saved after training.

10 Modelling Decisions

We experimented with multiple segmentation models (DeepLabV3, Fast-SCNN, and UNet) and varied learning rates to identify the best-performing architecture for our task. The following summarizes the results:

DeepLabV3

- **LR = 1e-6:** Dice = 0.6576, IoU = 0.6576, Pixel Accuracy = 0.0.9483

Fast-SCNN

- **LR = 1e-4:** Overfitting observed Dice = IoU = Pixel Accuracy = 1.0

UNet

- **LR = 1e-6:** Dice = 0.78, IoU = 0.78 suspected high overfitting due to very high pixel wise accuracy

Based on the results, we recommend **DeepLabV3 with a learning rate of 1e-6** as the final model for downstream fine-tuning, due to its strong performance and generalization.

11 Tensorboard plots for the best model

In this section, we show Tensorboard plots for the best model(deeplab). Grey line is for train, blue is for validation

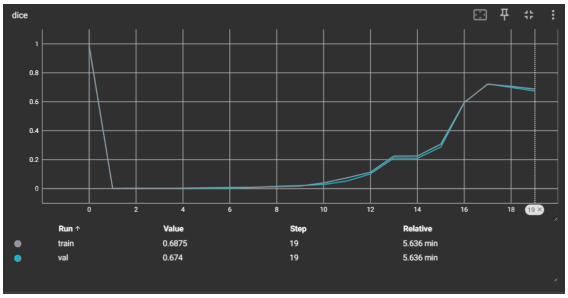

Figure 5: dice for train and val versus epochs

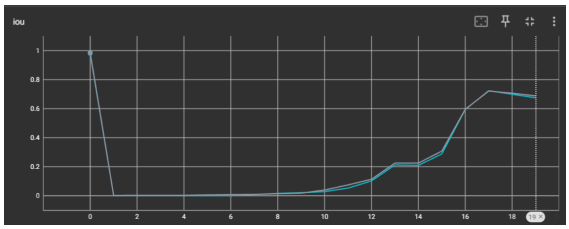

Figure 6: iou for train and val versus epochs

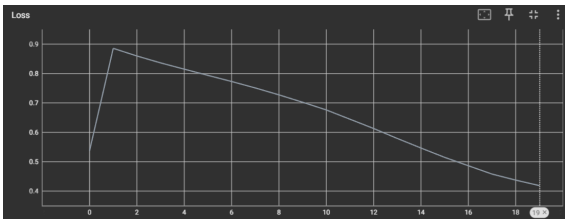

Figure 7: loss for train and val versus epochs

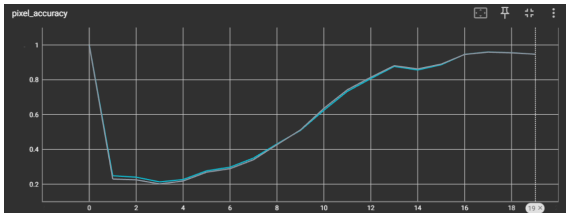

Figure 8: pixel$_{a}ccuracyfortrainandval$