Final Report

Submitted by Benjamin Debbarma

TASK 1:

Overview:

The task 1 involves preparing a semantic segmentation dataset derived from the COCO 2017 dataset. A custom Python script (notebook) was developed to download, subset, and preprocess data to generate RGBA multi-class masks for segmentation model training.

Dataset Selection:

- Dataset: COCO 2017
- Target Classes:
  - Person (`category_id=1`)
  - Bicycle (`category_id=2`)
  - Car (`category_id=3`)
- Subset Size: 2000 images per class (total 5813 images) & one annotations.json file
- Selection Strategy: Balanced sampling by image count per class.

Pipeline Steps:

1. Download and Extraction
   - `**train2017.zip**` and `**annotations_trainval2017.zip**` downloaded and extracted automatically.

2. Subset Generation
   - Images containing the specified categories are randomly shuffled and selected.
   - Only annotations for the selected categories are retained.
   - Selected images are copied to a new subset directory.

3. Mask Generation
   - RGBA masks are created for each image.
   - Each category is assigned a unique color:
     - Person → Red `(255, 0, 0, 255)`
     - Bicycle → Green `(0, 255, 0, 255)`
     - Car → Blue `(0, 0, 255, 255)`
   - Masks are constructed ‘per class’ using logical OR over all object instances.

- Final RGBA mask is built by ‘overlaying per-class masks in insertion order’.

4. Multiprocessing Support:
Mask generation is parallelized using all available CPU cores to process large datasets efficiently.

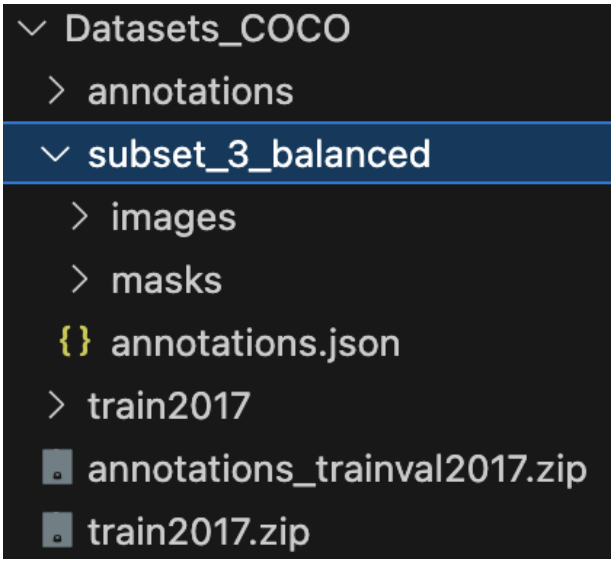
Design Considerations:

- **Overlaps**: Overlaps are resolved based on dictionary order (person → bicycle → car).
- **Alpha-blended RGBA masks:** Allows clear differentiation and visualization of overlapping regions.
- **Modular pipeline**: Subsetting and mask generation can be reused for any COCO-style dataset.

Edge Case Handling:

- **Missing images or annotations**: Caught using `try-except` blocks and skipped.
- **Unknown category IDs**: Mapped to a default gray color `(128, 128, 128, 255)`.
- **Empty masks or blank annotations**: Handled safely without crashing or incorrect file output.

Sample Output Directory Structure:



Conclusion:

The project delivers a clean, balanced dataset with pixel-level semantic labels, well-suited for training semantic segmentation models. Its modularity, multiprocessing support, and COCO compatibility make it extensible to other categories or datasets.

Task 2:

Objective:

This task focuses on training a semantic segmentation model using the dataset prepared in Task 1 (subset of COCO 2017). The goal is to segment images into meaningful categories while ensuring generalization, computational efficiency, and clarity of implementation.

Model Architecture:

- Model Used: **U-Net**
- Input Size: 3×512×512 RGB images
- Number of Classes: 4 (background, person, bicycle, car)
- Loss Function: Focal Loss with class balancing
- Optimizer: Adam with weight decay
- Regularization:
  - Dropout in convolutional layers
  - Early stopping based on validation loss
- Normalization: BatchNorm2D layers

Design Decisions:

- Why U-Net?
  - Lightweight and effective for dense segmentation tasks
  - Well-suited for limited computational budgets

- Why Focal Loss?
  - Better performance on class imbalance.
  - Emphasizes hard-to-classify pixels

- Bounding Box-based Loss Masking
  - To focus the loss computation only inside regions of interest
  - Helps suppress noisy background gradient signals

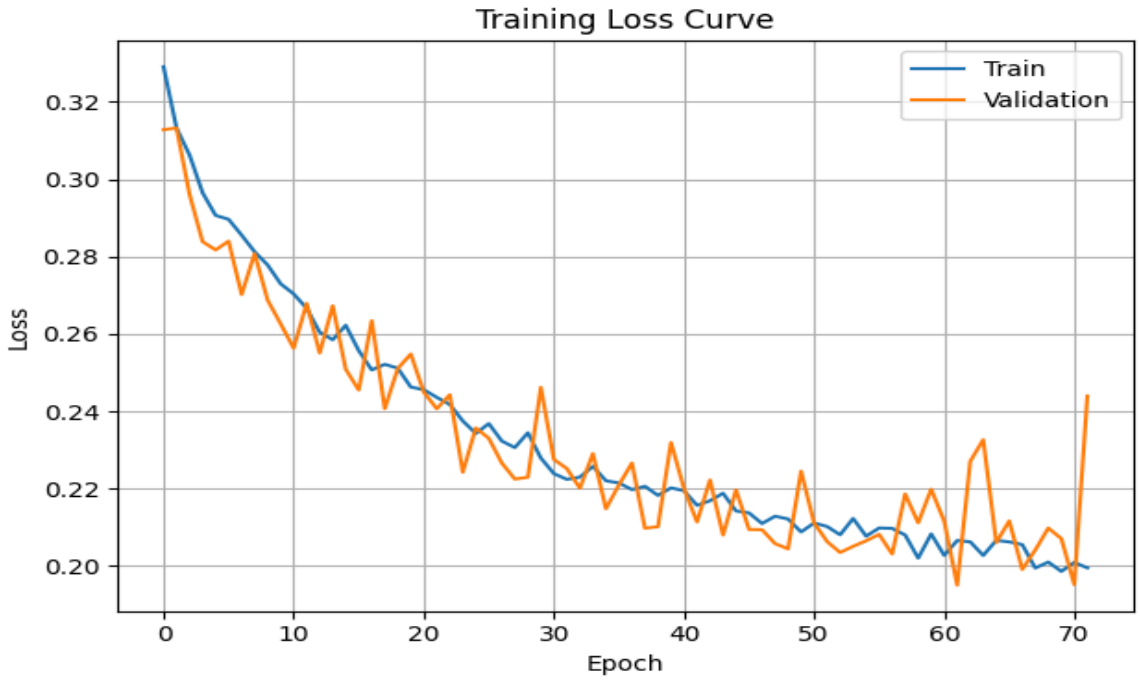
- TensorBoard Integration
  - Used to monitor training and validation losses in real-time

Training Summary:

- Training Set: 70% of 5813 images (~4069 images)
- Validation Set: 20% (~1162 images)
- Test Set: 10% (~581 images)
- Hardware: Single GPU system (CUDA-enabled), batch size = 8
- Runtime: Full training completed in ~1 hour with early stopping at optimal validation loss
- Mixed Precision: Used PyTorch AMP for faster and memory-efficient training

Training Performance:

- Training Metrics Tracked:
  - Loss per epoch (train/val)
  - Early stopping and best model checkpoint saved
- Tool Used: TensorBoard (`runs/segmentation_experiment`)



Training Monitoring:

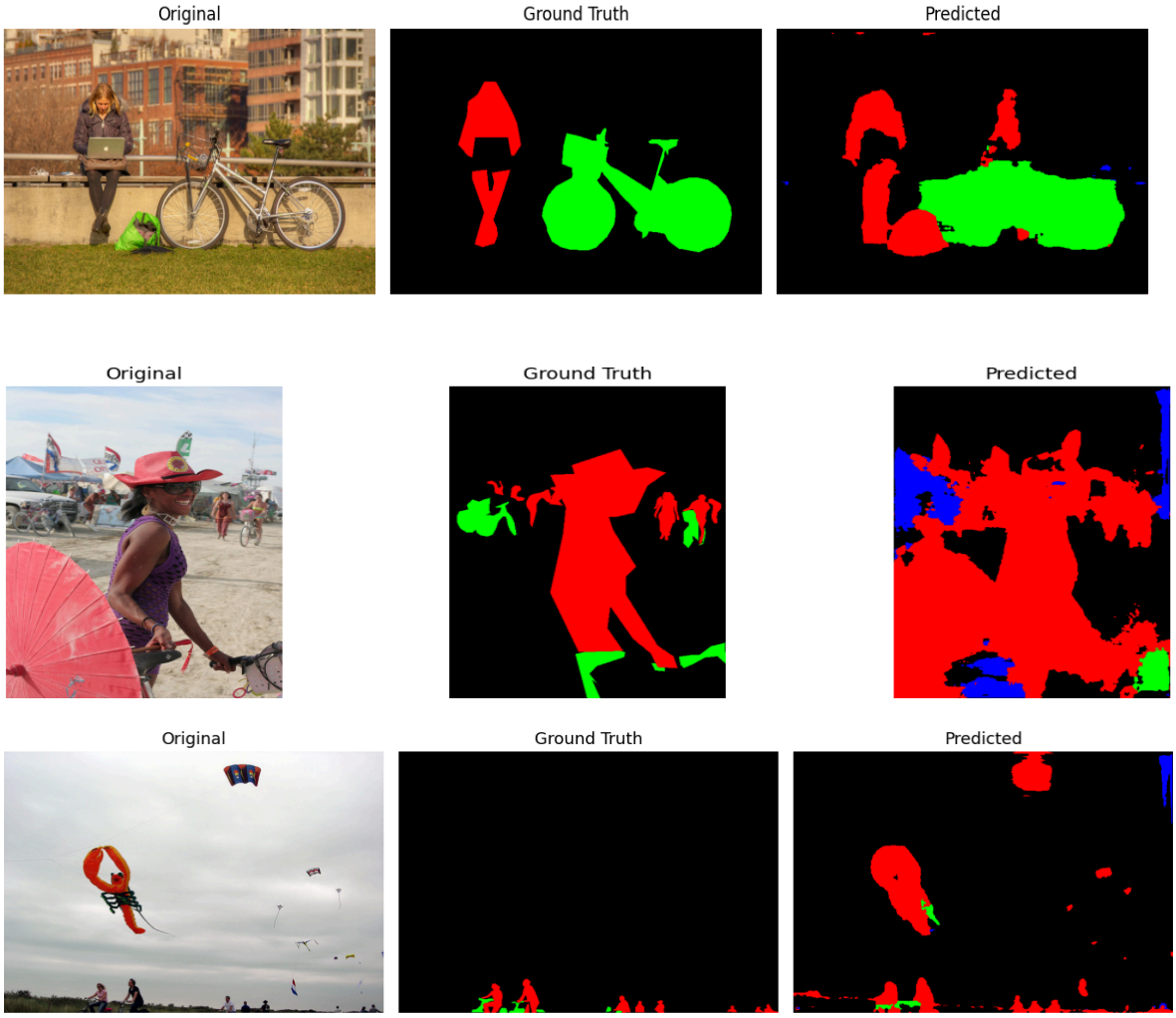
TensorBoard was used to track training and validation losses over epochs. Since this is a local setup, TensorBoard logs are not hosted publicly.

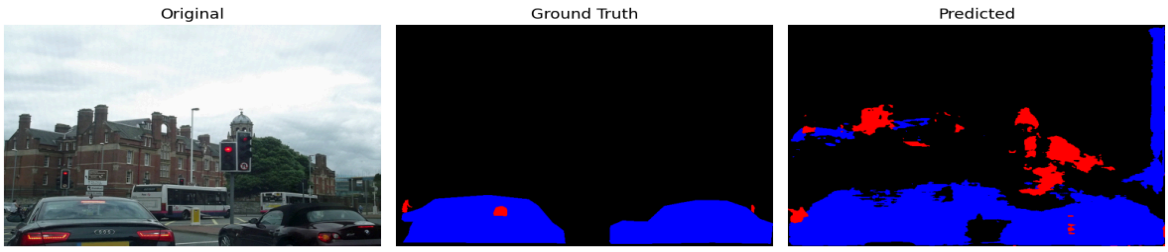
Instead, key metrics were:
- Captured as ‘screenshots’ from TensorBoard scalar plots (included in the repository).
- Exported as a .csv file for further reference or plotting.

Evaluation Results:

Pixel Accuracy: 0.6281300783157349
Mean IoU: 0.38335143996182375
Mean F1 Score: 0.5443127784377904

Model Inference & Visualization:





Color coding:
- Background: Black
- Person: Red
- Bicycle: Green
- Car: Blue

Challenges & Fixes:

- Challenge: Background pixels dominated the image, making the model biased.
  - Fix: Focal loss + loss masking inside bounding boxes

- Challenge: Overfitting beyond 30 epochs
  - Fix: Early stopping with patience = 10

- Challenge: Mask aliasing during resizing
  - Fix: Used nearest-neighbor interpolation to preserve discrete class values

Reproducibility Instructions: The whole code was run on

- Python Version: 3.8.10
- torch: 2.4.0+cu121
- matplotlib: 3.7.5
-numpy: 1.24.3
-Pillow: 10.4.0
-requests: 2.32.3
-torchvision: 0.19.1+cu121
-tqdm: 4.66.5
-pycocotools: 2.0

- The total time taken to run the code for task 1 was 60 mins to 90 mins (including downloading the ‘COCO 2017’ datasets)
- The total time taken to run the code for task 2 was 60 mins to 100 mins.
- Total time taken to run the whole code from start to finish: 2 hrs to 3.5 hrs.(approx.)

- Run `VJT.ipynb` from start to finish
- Use TensorBoard for metrics: `tensorboard --logdir=runs`

  ```
  For e.g.python -m tensorboard.main
  --logdir=/home/benjamin/ADRL/VJT_Task1/runs/segmentation_experime
  nt
  ```

Note: Due to the use of random sampling (e.g., for creating balanced subsets and selecting samples for visualization), the model's exact evaluation metrics (e.g., Mean IoU, F1 Score) may slightly vary across runs. To achieve bitwise reproducibility, setting a global random seed would be necessary, but was not enforced in this implementation to keep the code lightweight and flexible.

GitHub Repo Link: https://github.com/benjamin90s/VJTechnologies.git