ML-Eng-Test – Solutions

In this task, I tackled the problem of detecting walls and rooms from architectural blueprints using two different approaches:

1. A custom OpenCV image processing and wall detection algorithm.
2. The CubiCasa model to detect walls and rooms.

The first approach involves applying classic image processing techniques to detect walls, while the second approach uses a pre-trained machine learning model for room and wall detection.

First Approach: Custom OpenCV Image Processing and Wall Detection

Introduction

The first approach to detect walls in architectural blueprints is a custom image processing solution using OpenCV. This approach processes the input image to detect edges (which often represent walls) and merges overlapping lines, while simultaneously using OCR to identify text labels such as room names. The following steps outline how this algorithm works, explaining the key components and the overall workflow.

1. **Image Preparation and Splitting**

   Before processing begins, the image is analyzed to determine if it should be split based on its size. Splitting large images ensures better manageability and efficiency during image processing. Here's the breakdown:

   Function determine_split(image): Determines how to split the image based on its dimensions. Smaller images may be processed as a whole, while larger ones are split into 2, 4, 6, or 9 parts depending on the size range.

   Function split_image(image, rows, cols): Once the image is deemed large, it's divided into sub-images based on the number of rows and columns determined in the previous step. This division helps manage processing for complex, large blueprints.

2. **Edge Detection and Line Merging**

   To detect potential walls, edge detection is applied, followed by Hough line transformation to identify straight lines representing walls. The walls (lines) are further refined to handle overlaps.

   Canny Edge Detection: This technique highlights the edges in the image where there are sharp changes in intensity, which in blueprints, often corresponds to wall boundaries.

   Hough Line Transformation: This transformation is applied to detect straight lines that represent the walls. Hough lines are detected based on edge pixels, and the resulting lines are considered to represent walls.

   Function merge_overlapping_lines(lines): The detected lines are checked for overlaps. This function merges lines that are more than 70% overlap. The merging reduces redundancy and ensures cleaner wall detection. Overlapping is calculated based on distances and lengths of the lines.

3. **Optical Character Recognition (OCR) and Text Box Merging**

   In addition to detecting walls, the algorithm uses Optical Character Recognition (OCR) to detect and merge room labels.

   PaddleOCR: Used to detect text, such as room names, from the image. This is especially useful in architectural blueprints, where room names help in understanding the blueprint's structure.

   Function merge_text_boxes(text_boxes): This function merges OCR text boxes that are overlapping or touching. It ensures that the detected room names are represented with clean, non-overlapping bounding boxes.

4. **Line and Text Interaction Handling**

   Once both lines (walls) and text boxes (room labels) are detected, it's important to ensure that text labels do not interfere with wall detection.

   Function does_line_intersect_box(x1, y1, x2, y2, boxes): This checks whether any detected wall lines intersect with text boxes. Walls should not be drawn over room names or other text elements, ensuring clear labeling.

5. **Drawing and Visualization**

   Finally, the detected walls and room names are drawn on the image for visualization.

   Red Dots and Green Lines: Green lines represent the detected walls, and blue dots are placed at the start and end of each wall, providing a clear indication of the wall's boundaries.

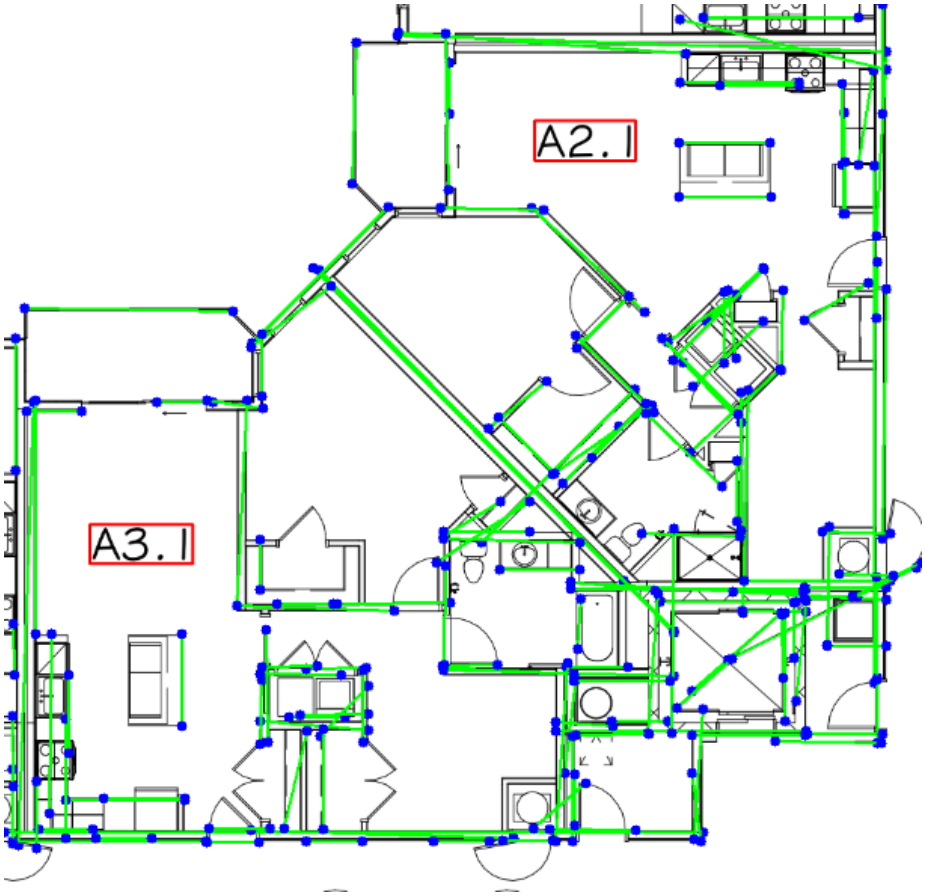
   Merged Text Boxes: Room labels are outlined in red. Overlapping text boxes are merged into a single bounding box for cleaner visualization.

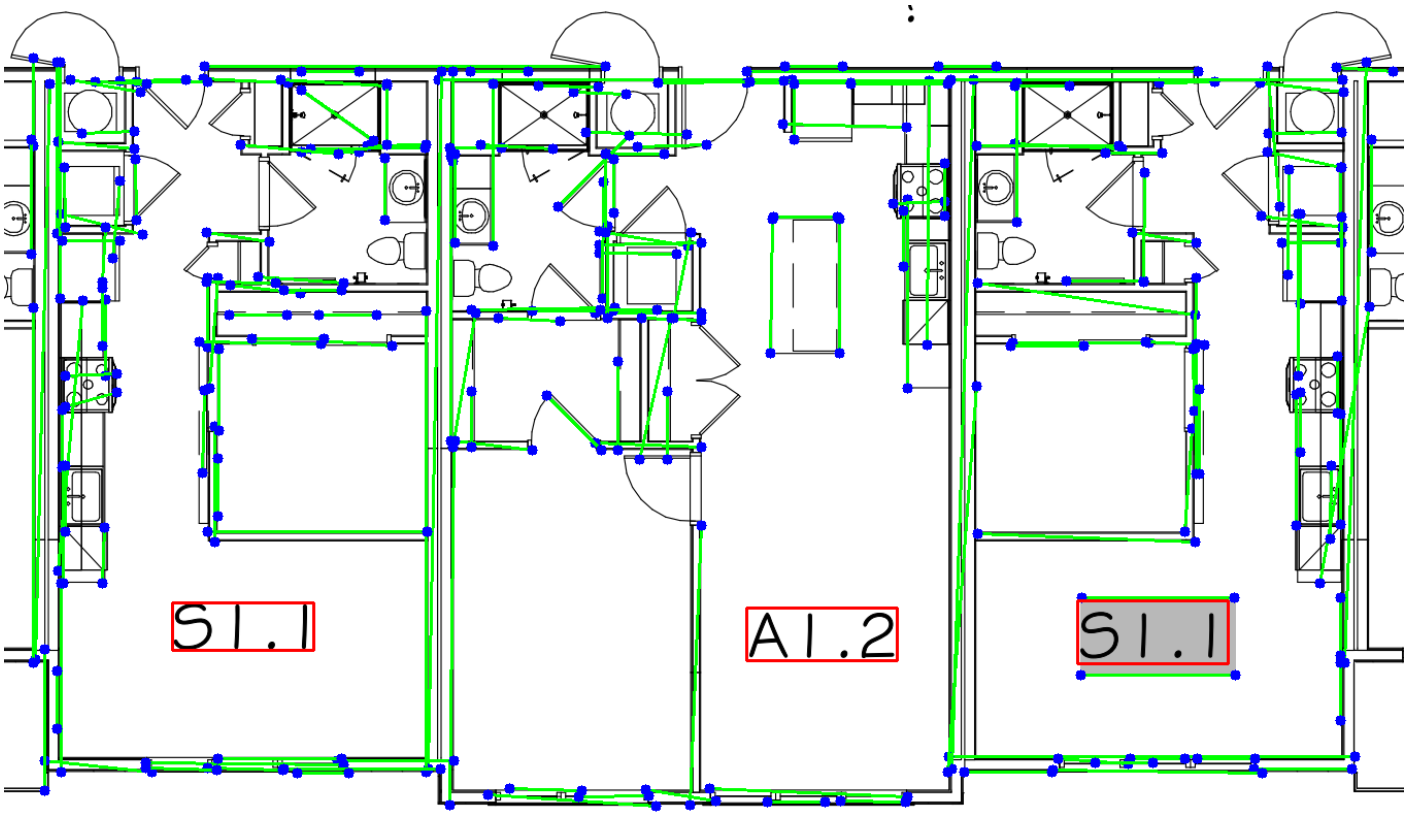
6. **Stitching Processed Sub-Images:** For large images that were split earlier, the sub-images are processed independently. Once processing is complete, they are stitched back together.

   Function stitch_images(sub_images, rows, cols, sub_height, sub_width): This function stitches the processed sub-images back into a single image, maintaining the layout and structure of the original blueprint.

Conclusion: This custom OpenCV-based wall detection approach effectively handles large blueprint images by splitting, detecting, and merging edges and text labels. The combination of edge detection, line merging, and OCR ensures that walls and rooms are accurately identified without interfering with room labels.

Below are output images of the code:





Second Approach: CubiCasa Model for Room and Icon Detection

Introduction

The second approach utilizes a machine learning-based method to detect rooms and icons in architectural blueprints. This approach uses a pre-trained Cubicasa model that processes the image, performs segmentation, and returns polygons representing rooms and icons. The steps below provide a detailed description of how this model is applied for both room detection and icon detection.

1. **Image Preprocessing**

   The input image is preprocessed before being fed into the model. This ensures that the image is in the correct format for the Cubicasa model.

   Function preprocess_image(image)**:** Converts the image into a tensor and normalizes it with a predefined mean and standard deviation. This is essential for making the input image compatible with the model's expected input format.

   Normalization: Normalization is applied to standardize the image's pixel values, improving model performance by ensuring consistent input data.

2. **Rotations and Model Prediction:**

   To enhance prediction accuracy, multiple rotations are applied to the image, and the predictions from each rotation are averaged. This rotation helps capture different orientations of rooms and icons in the blueprint.

   Rotating the Image: The image is rotated by different angles to capture various orientations. Each rotated version of the image is fed into the model for prediction.

   Function rot(image, 'tensor', forward)**:** Rotates the image before the forward pass through the model. After obtaining predictions, the image is rotated back to its original orientation.

   Model Prediction: The model predicts both room segmentation and icon segmentation by analyzing each rotated version of the image. The predictions are interpolated to maintain the original image size.

   Averaging the Predictions: The model outputs multiple predictions (one for each rotation). These predictions are averaged to create a final, refined prediction.

3. **Segmentation and Polygon Extraction**

   Once the predictions are averaged, they are processed to extract room and icon segments from the image.

   Segmentation Output: The model outputs two main components:

   - **Room Segmentation:** Indicates which pixels belong to rooms.
   - **Icon Segmentation:** Indicates the location of icons like doors, windows, and furniture.

Function split_prediction(prediction): This function splits the model's output into heatmaps, room segmentation, and icon segmentation components. These components are essential for identifying the specific regions in the image where rooms and icons are located.

Polygon Extraction: The segmented areas are further processed into polygons representing the outlines of rooms and icons. These polygons help accurately localize and label rooms and icons in the blueprint.

Function get_polygons(heatmaps, rooms, icons)**:** This function extracts the polygons for rooms and icons from the heatmap and segmentation results. These polygons define the boundaries of different rooms and icons.

4. **Visualizing the Room and Icon Segments**

The detected rooms and icons are visualized using a colormap to make the results more interpretable.

Function apply_colormap(segmentation, cmap_name)**:** This function applies a colormap (e.g., 'viridis') to the segmented image, converting the grayscale segmentation into a visually appealing RGB image.

Room Segmentation Visualization: The room segments are colored using a colormap to visually distinguish different rooms in the blueprint. The output is an image where each room is represented by a distinct color.

Icon Segmentation Visualization: The icons are also segmented and visualized in a similar way, with the colormap applied to make each icon distinct in the final output image.

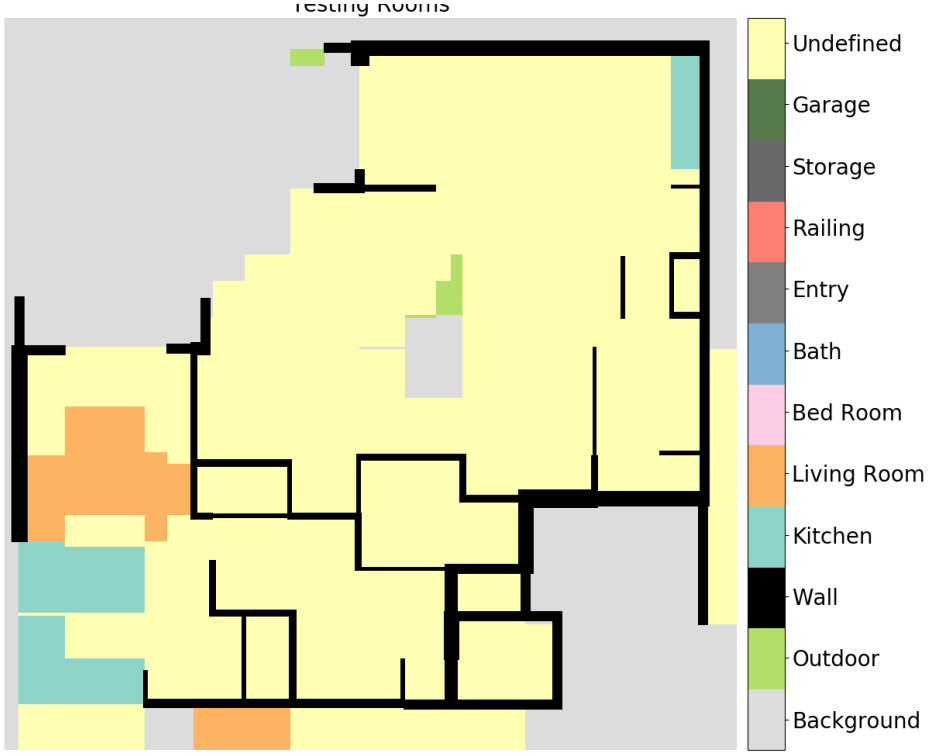
5. **Room and Icon Detection Workflow**

For Room Detection: The model processes the image, performs segmentation, and extracts polygons representing room boundaries. These polygons are colored and returned as the final room detection result.

For Icon Detection: The same code is reused for icon detection. The model segments the image for icons, extracts polygons representing icons like doors and windows, and visualizes them using a colormap.

Conclusion: The second approach using the CubiCasa model allows for efficient and accurate room and icon detection from architectural blueprints. By applying rotations, averaging predictions, and visualizing the segments, this method provides a clear and interpretable output. Both rooms and icons are detected and colored, making it easier to understand the blueprint’s layout.

Below are the output images of the code:



Improvements

The current implementations for both approaches have room for enhancement:

First Approach (Custom OpenCV):

> **Wall Detection Logic:** The wall detection logic can be improved by incorporating more advanced OpenCV image processing techniques such as morphological transformations, contour detection, or watershed algorithms. This could improve the precision of line merging and handle noisy blueprints more effectively. Given that this approach is faster due to its reliance on classical methods, it remains a solid choice for real-time or low-computation scenarios. By refining these techniques, we could push the accuracy close to machine learning models while maintaining speed.

Second Approach (CubiCasa Model):

> **Custom Dataset:** The performance of the model could be significantly improved by training it on a custom dataset of architectural blueprints that reflect real-world variability. This would enhance its ability to generalize to different blueprint styles and provide more accurate room and icon segmentation. However, this approach may require more computational resources and time for inference, making it slower compared to traditional methods.

Ultimately, the **first approach** stands out as a faster solution, especially for applications requiring quick processing with acceptable accuracy. However, combining the strengths of both approaches—such as using the custom image processing for speed and the ML model for fine-tuning specific detections—could offer a balanced solution for diverse use cases.