If you want to use your own dataset：

Our dataset consists of:  
(I) 512x512 pixel centered images obtained by processing the "LIFULL HOME High-Resolution Floor Plan Data" ("LIFULL HOME Data" for short).  
Because the "LIFULL HOME Data" has significant differences in size and margins, we use 512x512 pixel centered images (rather than the "LIFULL HOME Data") in our model.  
(II) Annotations.

For where you need to prepare your own dataset:

Regarding images, you will need to prepare a dataset of floor plan images. You will need to process these images into 512x512 RGB images, centering the floor plan and leaving 64 pixels of margin on each side in the narrowest direction (more on this later). My implementation only supports this setup.

You will also need to prepare annotations:  
For annotations, ideally, your existing dataset already has annotations, such as cubicasa5k. You will only need to design some algorithm to convert them into the required format (based on my understanding of cubicasa5k, this algorithm may be not easy). If your dataset only contains images, you can only obtain annotations using a trained neural network, other algorithms, or manually.

In short, the data you need includes:  
annot\_json/instances\_test.json  
annot\_json/instances\_train.json  
annot\_json/instances\_val.json  
I'll use the instances\_test data as an example. It contains a list of 'categories', a list of 'images', and a list of 'annotations'.

The output for 'categories' is as follows:  
[  
{  
"supercategory": "Corner",  
"id": 1,  
"name": "Corner"  
}  
]  
We won't use this list; it's just a historical relic. I'm using it here simply to demonstrate what our data contains.

The 'images' list contains the [Number of test set samples] item. I randomly print one item:  
{  
"height": 512,  
"width": 512,  
"id": "00-48-fabaa2bf5993a8d42e4a24db40ac-0002",  
"file\_name": "00-48-fabaa2bf5993a8d42e4a24db40ac-0002.jpg"  
}  
Height and width are both set to 512. id is the file name (corresponding to the image file name). It is actually linked to the LIFULL naming convention. For other datasets, the easiest way to run the code directly is to randomly generate a hexadecimal file name according to the above format. file\_name is id + '.jpg'.

The 'annotations' list contains the [Test set junction count] item. I'll print a random item:  
{  
"image\_id": "09-48-d0913f62af59db9b45f3823b3bfd-0001",  
"category\_id": 1,  
"id": 40152590,  
"point": [  
302,  
403  
],  
"edge\_code": 11,  
"semantic": [  
"bedroom",  
"closet",  
"balcony",  
"balcony"  
]  
}  
image\_id and the id above are the same thing, indicating the test sample to which the junction belongs. Just set category\_id to 1. id is a random number, you can randomly choose from 40000000 to 49999999. point represents the (x, y) coordinates of the junction. edge\_code is a category ranging from 0 to 15. For their meaning, see util/edges\_utils.py. The semantic order is "upper left - upper right - lower right - lower left", indicating the room type in each of the four directions of the junction.

You also need to prepare the topological relationships of the corner points.  
This is stored in the annot\_npy/xxxxxxxxxxxx.npy file, which contains [number of test set] files. xxxxxxxxxxxxx is the 'id' in the 'images' list. I print a random item:  
{(158, 64): [(-1, -1), (-1, -1), (158, 176), (193, 64)], (193, 64): [(-1, -1), (158, 64), (193, 176), (352, 64)], (352, 64): [(-1, -1), (193, 64), (352, 176), (-1, -1)], (158, 176): [(158, 64), (-1, -1), (158, 326), (193, 176)], (193, 176): [(193, 64), (158, 176), (-1, -1), (352, 176)], (352, 176): [(352, 64), (193, 176), (352, 356), (-1, -1)], (158, 326): [(158, 176), (-1, -1), (158, 374), (277, 326)], (277, 326): [(-1, -1), (158, 326), (277, 374), (301, 326)], (301, 326): [(-1, -1), (277, 326), (301, 356), (-1, -1)], (301, 356): [(301, 326), (-1, -1), (301, 374), (352, 356)], (352, 356): [(352, 176), (301, 356), (352, 447), (-1, -1)], (158, 374): [(158, 326), (-1, -1), (158, 447), (258, 374)], (258, 374): [(-1, -1), (158, 374), (258, 447), (277, 374)], (277, 374): [(277, 326), (258, 374), (-1, -1), (301, 374)], (301, 374): [(301, 356), (277, 374), (301, 447), (-1, -1)], (158, 447): [(158, 374), (-1, -1), (-1, -1), (258, 447)], (258, 447): [(258, 374), (158, 447), (-1, -1), (301, 447)], (301, 447): [(301, 374), (258, 447), (-1, -1), (352, 447)], (352, 447): [(352, 356), (301, 447), (-1, -1), (-1, -1)],  
'quatree': [{0: [(158, 64)], 1: [(193, 64), (158, 176)], 2: [(352, 64), (193, 176), (158, 326)], 3: [(352, 176), (277, 326), (158, 374)], 4: [(352, 356), (301, 326), (277, 374), (258, 374), (158, 447)], 5: [(301, 356), (352, 447), (301, 374), (258, 447)], 6: [(301, 447)]}]}

It can be seen that it contains two parts. For the first part, for example (258, 374): [(-1, -1), (158, 374), (258, 447), (277, 374)], indicating that the corner (258, 374) is not connected to any corners above it, is connected to (158, 374) on the left, (258, 447) on the bottom, and (277, 374) on the right.

For the latter part 'quatree', for example, 0: [(158, 64)], represents the leftmost junction of the topmost junctions of the structural graph. Starting from 1: [(193, 64), (158, 176)] (no order), the construction method is based on breadth-first search to contain all adjacent points of (158, 64). 2: [(352, 64), (193, 176), (158, 326)] represents all the adjacent vertices of [(193, 64), (158, 176)] that are not contained in [(158, 64)] or [(193, 64), (158, 176)]. You can refer to this diagram:

