这个action是绝对坐标我担心难以收敛，我想按如下方式进行构建。首先主芯片，就是一个环境中最大，且与其他器件链接最多的芯片放到画布正中间。然后其他器件根据贪婪的形式进行预布局，例如根据和主芯片pin角的链接关系，将器件放到欧式距离最短的地方。然后action改为对初级布局结果的调整，其分为3类，第一类是粗调整：包括不调整，调整到以芯片核心器件为中心划分的象限中例如化为16个象限中，然后再action决定离主芯片的距离。第二级是细调整，分为不调整，上、下、左、右、移动，然后再决定移动多少格。第三级是将器件旋转，分为不旋转，正时针90读，逆时针90度，180度旋转。然后其中action是除了主芯片以外，每个器件都要决策下，这样一轮决策完毕，才最终给出reward，并且注意下后面执行action的时候不要和前面调整过的action重合，但是可以不管后面未调整的器件。环境的reward就先统计pin之间飞线的欧式距离，距离越短分数越高，

好的，现在我们来开始新写一个python准备给强化学习训练用的state数据，其调用placement\_game里面的PCBRLEnv Class。读入PCBRLEnv后，提取state信息，State我想用融合的形势来表征，包括图信息graph，图片信息，还有基于transformer的序列信息。图信息：每个pin是一个node，里面包括他所属的器件，他当前的坐标，和其他pin链接的关系。图信息构建好后用GCN来提取特征。图片信息就是按256x256的图片，其中每个器件都填充起来，不同器件填充的灰度值不一样，注意归一化下到0-1，然后用个小的resnet来提取特征。Transfomer主要表示一些特征信息，格式如下

Transformer 的编码如下SOS，MOS,模块ID（从小到大）,Node ID1，Node ID2和之前graph的node对应上，Node ID1和Node ID2当前的距离；Node ID3，Node ID4和之前graph的node对应上，Node ID1和Node ID2当前的距离.。。。MOE, MOS,模块ID…描述完每个模块，最后MOE，SOE

SOS，序列开始

SOS: 编码为0

SOE: 序列结束

MOS：模块开始

MOE：模块结束

没有node连线的可以不描述

好了 现在我们可以开始构建一个基于PPO的强化学习框架了，输入包括pcb\_fused\_rl\_state.pkl里面每个json文件（当前验证正确性的化先只读入'pcb\_cells\_1.json'，但是接口要预留好，后面随机读入），的graph，image，sequence，其中graph用GCN提取信息image用小的resnet提取信息，sequence用一个transfomer提取信息，提取后，三个融合到一起作为state的特征。Action是逐模块进行的，模块顺序和transformer的模块顺序保持一致。我把env代码附在下面帮助你来构建这个程序,你可以直接调用，如果需要修改也进行修改。另外注意graph信息是不变的所以action执行后不用去更新他，image和sequence是要改变的所以存到记忆尺里面。

class PCBRLEnv:  
 def \_\_init\_\_(self, pcb\_json\_path):  
 with open(pcb\_json\_path, "r", encoding="utf-8") as f:  
 data = json.load(f)  
 self.raw\_data = deepcopy(data)  
 self.grid\_size = 256  
 self.reset()  
 # action space定义  
 self.n\_quadrant = 16  
 self.n\_fine\_move = 5 # 不动、上、下、左、右  
 self.n\_fine\_step = 5 # 1~5格  
 self.n\_rotate = 4 # 不转，顺时针90，逆时针90，180  
 # 编码方案：[粗调类型, 粗调象限, 粗调距离], [细调方向, 细调步数], [旋转方式]  
  
 def reset(self):  
 self.data = deepcopy(self.raw\_data)  
 self.cell\_list = self.data["cellList"]  
 self.net\_list = self.data["netList"]  
 self.main\_idx = self.\_find\_main\_idx()  
 self.n\_cells = len(self.cell\_list)  
 self.actions = []  
 self.\_refresh\_cache()  
 return self.\_get\_state()  
  
 def \_find\_main\_idx(self):  
 # 默认主芯片为面积最大者  
 areas = []  
 for cell in self.raw\_data["cellList"]:  
 contour = eval(cell["contour"])  
 xs = [p[0] for p in contour]  
 ys = [p[1] for p in contour]  
 areas.append((max(xs)-min(xs))\*(max(ys)-min(ys)))  
 return np.argmax(areas)  
  
 def \_refresh\_cache(self):  
 # 更新所有cell的center, contour, pin的center/contour到np数组，旋转等  
 self.cell\_pos = []  
 self.cell\_rot = []  
 for cell in self.cell\_list:  
 center = np.array(eval(cell["center"]))  
 self.cell\_pos.append(center)  
 # rotation一定要是整数  
 self.cell\_rot.append(int(cell.get("rotation", "0")))  
  
 def \_get\_state(self):  
 # 返回所有cell的(center, rotation)，可附加feature  
 state = []  
 for i, cell in enumerate(self.cell\_list):  
 center = np.array(eval(cell["center"]))  
 rot = int(cell.get("rotation", "0"))  
 state.append(np.concatenate([center, [rot]]))  
 return np.stack(state)  
  
 def get\_cell\_names(self):  
 return [cell["cellName"] for cell in self.cell\_list]  
  
 def step(self, actions):  
 # actions: n\_cells-1个，每个形如(action1, action2, action3)  
 assert len(actions) == self.n\_cells-1, "必须对每个非主芯片都出决策"  
 for i, act in enumerate(actions):  
 cell\_idx = i if i < self.main\_idx else i+1 # 跳过主芯片  
 self.\_apply\_action(cell\_idx, act)  
 self.\_refresh\_cache()  
 reward = self.\_compute\_reward()  
 done = True # 一轮就done  
 return self.\_get\_state(), reward, done, {}  
  
 def \_apply\_action(self, idx, action):  
 # action = (粗调类型, 粗调象限, 粗调距离, 细调方向, 细调步数, 旋转)  
 # idx: cell\_list下标  
 cell = self.cell\_list[idx]  
 center = np.array(eval(cell["center"]))  
 rot = int(cell.get("rotation", "0"))  
  
 # 1. 粗调  
 quadrant\_type, quadrant\_idx, quadrant\_dist = action[0:3]  
 if quadrant\_type != 0:  
 # 象限参考主芯片  
 main\_center = np.array(eval(self.cell\_list[self.main\_idx]["center"]))  
 step = self.grid\_size // 4 # 每象限边长  
 row = quadrant\_idx // 4  
 col = quadrant\_idx % 4  
 quad\_x = int(main\_center[0] + (col-1.5)\*step)  
 quad\_y = int(main\_center[1] + (row-1.5)\*step)  
 # 距离向主芯片中心靠拢（可自定义），或取象限中心到main中心的向量，再缩放  
 dir\_vec = main\_center - np.array([quad\_x, quad\_y])  
 dist = int(quadrant\_dist\*step/4)  
 new\_center = np.array([quad\_x, quad\_y]) + dir\_vec/np.linalg.norm(dir\_vec+1e-8)\*dist  
 center = new\_center.astype(int)  
 # 2. 细调  
 fine\_dir, fine\_step = action[3:5]  
 move\_delta = np.zeros(2)  
 if fine\_dir == 1: # 上  
 move\_delta[1] = -fine\_step  
 elif fine\_dir == 2: # 下  
 move\_delta[1] = fine\_step  
 elif fine\_dir == 3: # 左  
 move\_delta[0] = -fine\_step  
 elif fine\_dir == 4: # 右  
 move\_delta[0] = fine\_step  
 center = (center + move\_delta).clip(0, self.grid\_size-1)  
  
 # 3. 旋转  
 rot\_mode = action[5]  
 if rot\_mode == 1:  
 rot = (rot + 90)%360  
 elif rot\_mode == 2:  
 rot = (rot - 90)%360  
 elif rot\_mode == 3:  
 rot = (rot + 180)%360  
 # 更新数据  
 cell["center"] = str(center.tolist())  
 cell["rotation"] = str(rot)  
 # contour和pin需要根据新的center/rotation同步重算（略，可用工具函数实现）  
  
 def \_compute\_reward(self):  
 # 全局pin-to-pin欧氏距离之和的负数  
 # 先统计所有pin全局坐标  
 pin\_coord\_dict = {}  
 for cell in self.cell\_list:  
 center = np.array(eval(cell["center"]))  
 rot = int(cell.get("rotation", "0"))  
 for pin in cell["pinList"]:  
 pin\_local = np.array(eval(pin["center"])) - np.array(eval(cell["center"])) # local offset  
 # 旋转  
 angle = np.deg2rad(rot)  
 mat = np.array([[np.cos(angle), -np.sin(angle)], [np.sin(angle), np.cos(angle)]])  
 pin\_real = np.dot(mat, pin\_local) + center  
 pin\_coord\_dict[(cell["cellName"], pin["pinName"])] = pin\_real  
 # 统计所有net的pin距离  
 total\_dist = 0  
 for net in self.net\_list:  
 pinList = net["pinList"]  
 for i in range(len(pinList)-1):  
 for j in range(i+1, len(pinList)):  
 p1 = pin\_coord\_dict[(pinList[i]["cellName"], pinList[i]["pinName"])]  
 p2 = pin\_coord\_dict[(pinList[j]["cellName"], pinList[j]["pinName"])]  
 total\_dist += np.linalg.norm(p1-p2)  
 return -total\_dist # 距离越短，reward越高