**代码说明**

**环境配置**

dgl==0.3

scipy==1.1.0

torch\_scatter==2.0.6

torch-cluster==1.5.9

torch-sparse==0.6.9

torch\_geometric==1.4.3

numpy==1.19.5

hyperopt==0.2.5

scikit\_learn==0.21.3

requests

**步骤1：训练延迟预测器**

1. step1：随机生成2000个架构，获取架构在特定设备上的推理延迟构建预测器的训练数据集
2. 修改内容：hardware\_platform
3. 运行命令：python RandomSearch.py --dataset Cora/Citeseer/SIoT --platform 3090/i5/jetson
4. step2：训练预测器
5. 将step1生成的文件放到文件夹./predictor/latency\_dataset/
6. 修改内容：

|  |
| --- |
| Python self.pdatamanager=PredictorDatasetManager("./latency\_dataset/3090\_Cora\_actions\_latency\_2\_layers.csv")  def test(self,best\_mape\_avg,model, time\_rand,model\_save\_path="./predictor\_model\_save/Cora\_extend\_latency/",return\_loss=False):  torch.save(model.state\_dict(), model\_save\_path+"RTX3090\_Cora\_layer2\_depth\_4\_hidden\_dim\_600\_lr\_e6\_brp\_gcn\_weight\_"+str(time\_rand)+".pth")  df\_test\_loss.to\_csv(model\_save\_path+"RTX3090\_Cora\_layer2\_depth\_4\_hidden\_dim\_600\_lr\_e6\_brp\_mape\_"+str(time\_rand)+".csv", index=False) |

1. 运行命令：python train\_latency.py

**步骤2：架构搜索**

1. 将训练好的预测器权重放到文件夹./hwgnas/predictor\_model\_pth
2. 运行命令：python hwgnas/main.py --dataset Cora/Citeseer/SIoT

**部分代码详解**

1. 架构编码

|  |
| --- |
| Python def get\_adj\_and\_feature\_and\_label(self, actions):  *'''start===========得到每个操作在其操作类型中的索引==================='''* actions\_without\_anchor = actions  skip\_connection\_list = [[]]  action\_index = 0  op\_index\_list = []  for i in range(0, self.args.layers\_of\_child\_model):  for key in self.search\_space.keys():  op\_index = self.search\_space[key].index(actions\_without\_anchor[action\_index])  action\_index += 1  op\_index\_list.append(op\_index)  actions\_index = torch.tensor(op\_index\_list, device='cuda:0')    '''start=========生成节点的输入特征矩阵=========================='''  # 节点个数 = 操作类型数\*层数(最后一层的最后一个hidden\_num可表示数据集种类——>表示输入特征维度、输出特征维度) + 1（全局节点1个，）  # op\_type\_num=len(self.search\_space.keys())  node\_num = self.args.layers\_of\_child\_model \* self.num\_state + 1  candidate\_num\_list = []  for key in self.search\_space.keys():  candidate\_num\_list.append(len(self.search\_space[key]))  max\_candidate\_num = max(candidate\_num\_list)  # feature\_dim = 1 + self.num\_state + max\_candidate\_num # 得到特征维度：1+5+11 = 17, 1为全局节点，op\_type\_num为操作类型个数，max\_candidate\_num为最大的候选操作数  feature\_dim=17  feature\_matrix = np.zeros((node\_num, feature\_dim)) # 定义一个初始化输入特征矩阵  feature\_matrix[0][0] = 1 # 代表索引为0的节点为全局节点   '''start=========处理中间的操作节点的特征=========================='''  for layer\_index in range(0, self.args.layers\_of\_child\_model):  for op\_type in range(0, self.num\_state):  '''处理操作类型'''  feature\_matrix[1 + layer\_index \* self.num\_state + op\_type][1 + op\_type] = 1 # 操作类型的特征索引为：1+op\_type  '''处理选中的候选操作'''  # 行坐标=1(第1行是全局节点)+层索引\*操作类型数量+候选操作  row = 1 + layer\_index \* self.num\_state + op\_type  # 列索引 =1(第1列是全局节点)+ 候选操作种类(第2~op\_type\_num列表示操作类型) + 选中的操作在其操作类型列表中的索引  col = 1 + self.num\_state + actions\_index[layer\_index \* self.num\_state + op\_type]  feature\_matrix[row][col] = 1 # 候选操作的特征索引为：8+op\_i  # print("feature\_matrix[{}][{}]=1".format(row,col))   '''==========================构建邻接矩阵=========================='''  '''构建邻接矩阵'''  edge\_row\_list = []  edge\_col\_list = []   adj\_matrix = np.zeros(shape=(node\_num, node\_num))  adj\_matrix[0][0] = 1  for node\_i in range(1, node\_num):  # print("node\_i={}".format(node\_i))  adj\_matrix[node\_i][node\_i] = 1 # 将对角元素置为1 ——>自环   adj\_matrix[0][node\_i] = 1 # 全局节点与所有其它节点相连   # 每个操作直接与后一个操作相连  if node\_i < self.args.layers\_of\_child\_model \* self.num\_state:  adj\_matrix[node\_i][node\_i + 1] = 1   for i in range(node\_num):  for j in range(node\_num):  if (adj\_matrix[i][j] == 1):  edge\_row\_list.append(i)  edge\_col\_list.append(j)  edge\_index=[edge\_row\_list, edge\_col\_list]   return adj\_matrix, feature\_matrix, edge\_index |