1. Edge_index (边索引）
The first row contains the starting node (source) of the edge.
The second row contains the ending node (target) of the edge.

Imagine we have a graph with 4 nodes (0, 1, 2, 3) and 3 edges connecting these nodes:

- Edge from node 0 to node 1
- Edge from node 1 to node 2
- Edge from node 2 to node 3

Code:
Edge_index = [[0,1,2],
[1,2,3]]
E.g.

```
edge_index:              torch.Size([2, 10556])
tensor([[ 633, 1862, 2582,  ...,  598, 1473, 2706],
        [   0,    0,    0,  ..., 2707, 2707, 2707]])
```

2.training mask, testing mask, validation mask
One dimensional tensor contains Boolean values representing which nodes are on the train set and which are on the test set or on the validation set.

E.g.

```
train_mask:              torch.Size([2708])
tensor([ True,  True,  True,  ..., False, False, False])
```

3. X and y
X is the node feature, y is the node labeling so every node belongs a category

```
x:              torch.Size([2708, 1433])
tensor([[0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.]])


y:              torch.Size([2708])
tensor([3, 4, 4,  ..., 3, 3, 3])
```

4.创建自定义网络模型

```
#自定义网络模型
class Net(torch.nn.Module):#创建网络模型
  def __init__(self):
    super(Net, self).__init__()
    #super(Net, self).__init__()这段代码的作用是确保父类torch.nn.Module的构造器被正确调用
    self.conv = SAGEConv(dataset.num_features,#x
                         dataset.num_classes,#classes
                         aggr="max")#在聚合邻居节点特征时使用最大值聚合策略
    #初始sageconv卷积图层

  def forward(self):
    x = self.conv(data.x,data.edge_index)
    return F.log_softmax(x,dim=1)
```

5.深度学习设置

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model, data = Net().to(device),data.to(device)
optimizer = torch.optim.Adam(model.parameters(),lr=0.01, weight_decay=5e-04)
```

第一段代码:检查是否有CUDA支持的GPU可用。如果有，就使用'cuda'设备(即GPU)来加速计算;如果没有，就使用'cpu'

第二段代码:把之前自定义的Net()作为实例化对象，使用.to(device)方法将模型和数据都移动到之前选择的设备上(GPU或CPU)

第三段代码: 使用adam优化器

6.train & test

```
def train():
 model.train()  # 将模型设置为训练模式。这对于具有特定层（如Dropout和BatchNorm）的模型特别重要，因为这些层在训练和评估时的行为不同。
 optimizer.zero_grad()  # 清除之前的梯度。这是因为梯度是累积的，如果不清零，梯度会包含多个batch的累积梯度，导致训练不准确。
 # 计算损失函数。这里使用负对数似然损失（NLL Loss），它常用于多分类问题。model()[data.train_mask]是模型对训练集的预测，data.y[data.train_mask]是训练集的真实标签。
 F.nll_loss(model()[data.train_mask], data.y[data.train_mask]).backward()
 optimizer.step()  # 进行一步梯度下降，更新模型的参数。


def test():
```

```
 model.eval()  # 将模型设置为评估模式。与train()相对，这会关闭Dropout和BatchNorm等层的特定训练行为，使模型在评估/测试时保持稳定。
 logits, accs = model(), []  # 获取模型的输出（logits）并初始化一个空列表来存储准确率。
 # 循环处理train_mask, val_mask, test_mask，这些掩码用于指示哪些数据应该用于训练、验证和测试。
 for _, mask in data('train_mask', 'val_mask', 'test_mask'):
    pred = logits[mask].max(1)[1]  # 对于每个掩码，选取logits中最大的预测值作为预测类别。
    # 计算准确率。eq函数比较预测和真实标签是否相等，sum().item()计算总的正确预测数，mask.sum().item()得到掩码中True的数量，即样本数。
    acc = pred.eq(data.y[mask]).sum().item() / mask.sum().item()
    accs.append(acc)  # 将计算得到的准确率添加到accs列表中。
 return accs  # 返回包含训练集、验证集和测试集准确率的列表。
```