

MGC-RM

GAT

GAT 的核心思想

GAT 的数学描述

1. 初始化

2. 注意力机制

3. 聚合

4. 更新

点积表示相关性

解释各个组成部分

为什么可以用来表示相关度

公式的意义

激活函数的作用

GAT的基本训练流程:

训练的模型架构:

MSE

相似度系数

Resilience

Softmax

Reference indicators

1. Natural Connectivity ↑

2. MSE ↓

3. AEC ↓

4. KL MI

5. Ds ↓

6. R_g

不同模型—Node feature distribution

Test Results

Similarity Score

5. GP : 75 (shanshili)

Reference indicators

1. Natural Connectivity

2. MSE - single node

3. ~~res_energy_avg_communication_circle~~

公式:

实验:

4. ACE

实验:

5. DS

公式:

实验:

LOG

241120

241119 梯度中断

241118

241117

241115

241114

241113 性能指标选点结果

241112

241111

241110

241106

241105

241104

241103
241102
241101
241025
241024
241023※
241022
241017
241016
241015
241014
241013
241012

MGC-RM

GAT

GAT (Graph Attention Networks, 图注意力网络) 是一种图神经网络 (GNN) 的变体，它通过引入自适应权重分配机制来改进传统的图卷积网络 (GCN)。GAT 通过学习邻居节点之间的注意力系数来动态地聚合邻居的信息，从而更好地捕捉图中节点之间的关系（某种节点表示，对自己的表示中有权重的考虑邻居节点）。

GAT 的核心思想

GAT 的核心思想在于通过注意力机制为每个节点的不同邻居分配不同的权重，这样可以更加灵活地处理不同节点的重要性。具体来说，GAT 的关键步骤包括：

1. **初始化**: 每个节点都有自己的特征表示（通常是特征向量）。
2. **注意力机制**: 通过计算注意力系数 (attention coefficients) 来决定邻居节点的重要性。
3. **聚合**: 根据注意力系数加权聚合邻居节点的信息。
4. **更新**: 使用聚合的信息来更新每个节点的特征表示。

GAT 的数学描述

以下是 GAT 的数学描述：

1. 初始化

假设有图 ($G = (V, E)$)，其中 (V) 是节点集，(E) 是边集。每个节点 (v_i) 有一个特征向量 (\mathbf{h}_i)。

2. 注意力机制

定义注意力系数 ($\alpha_{i,j}$)，它是节点 (v_i) 和 (v_j) 之间的关系强度的度量。GAT 使用共享的注意力机制 (α) 来计算注意力系数：

$$[e_{ij} = \text{LeakyReLU}(\mathbf{a}^T [\mathbf{W}\mathbf{h}_i || \mathbf{W}\mathbf{h}_j])]$$

单层的全连接层，表示一种：一对向量到一个标量的映射，表示两个向量的相关度

其中：

- (\mathbf{a}) 是一个可学习的向量，用于计算注意力分数。
- (\mathbf{W}) 是一个权重矩阵，用于变换节点特征。

- (\parallel) 表示特征向量的拼接 (concatenation)。
- (e_{ij}) 是未归一化的注意力分数。

接下来，对每个节点 (v_i) 的邻居节点计算注意力系数：

$$[\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik})}]$$

针对一个节点，其邻居对其的权重系数。所有邻居归一化处理，分配权重
softmax用于归一化

其中 ($\mathcal{N}(i)$) 是节点 (v_i) 的邻居集。

3. 聚合

使用注意力系数加权聚合邻居节点的信息：

$$[\mathbf{h}'_i = \sigma \left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} \mathbf{W} \mathbf{h}_j \right)]$$

其中 (σ) 是激活函数，如 $ReLU$ 。

4. 更新

更新节点 (v_i) 的特征表示：

$$[\mathbf{h}'_i = \text{Concat}(\mathbf{h}_i^{(1)}, \mathbf{h}_i^{(2)}, \dots, \mathbf{h}_i^{(K)})]$$

这里 (K) 是注意力头的数量，通常使用多头注意力机制来增强模型的表达能力。

内积表示向量的相似度

点积表示相关性

在 GAT (Graph Attention Networks) 中， $\text{LeakyReLU}(\mathbf{a}^T [\mathbf{W} \mathbf{h}_i \parallel \mathbf{W} \mathbf{h}_j])$ 用于计算两个节点 (v_i) 和 (v_j) 之间的注意力系数。这个表达式实际上是一个自定义的函数，用于衡量两个节点之间的相似度或相关性。下面详细解释一下各个组成部分及其意义：

解释各个组成部分

1. $(\mathbf{W} \mathbf{h}_i)$ 和 $(\mathbf{W} \mathbf{h}_j)$:

- (\mathbf{W}) 是一个权重矩阵，用于将原始节点特征向量 (\mathbf{h}_i) 和 (\mathbf{h}_j) 转换成新的特征表示。
- ($\mathbf{W} \mathbf{h}_i$) 和 ($\mathbf{W} \mathbf{h}_j$) 分别表示节点 (v_i) 和 (v_j) 在经过线性变换后的特征向量。

2. $([\mathbf{W} \mathbf{h}_i \parallel \mathbf{W} \mathbf{h}_j])$:

- (\parallel) 表示向量拼接 (concatenation) 操作，即将两个向量合并成一个新的向量。这样做是为了保留两个节点的特征信息，并通过拼接后的向量来计算它们之间的关系。
- 例如，如果 ($\mathbf{W} \mathbf{h}_i$) 和 ($\mathbf{W} \mathbf{h}_j$) 都是 (d)-维向量，则 ($[\mathbf{W} \mathbf{h}_i \parallel \mathbf{W} \mathbf{h}_j]$) 将是一个 ($2d$)-维向量。

3. (\mathbf{a}^T) 和 (\mathbf{a}) :

- (\mathbf{a}) 是一个可学习的向量，用于衡量两个节点特征向量的相似度或相关性。
- (\mathbf{a}^T) 表示 (\mathbf{a}) 的转置，使用转置后可以进行点积运算。

4. $(\mathbf{a}^T [\mathbf{W} \mathbf{h}_i \parallel \mathbf{W} \mathbf{h}_j])$:

- 这个表达式计算了 (\mathbf{a}) 向量与拼接后的特征向量 ($[\mathbf{W} \mathbf{h}_i \parallel \mathbf{W} \mathbf{h}_j]$) 的点积。
- 点积的结果是一个标量，用于衡量两个节点特征向量的相似度或相关性。

5. $(\text{LeakyReLU}(x))$:

- LeakyReLU 是一个激活函数，用于引入非线性。LeakyReLU 的定义如下：

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}$$

- 其中 (α) 是一个小的正数 (例如 0.01)，用于防止梯度消失问题。

为什么可以用来表示相关度

1. 衡量相似度：

- $(\mathbf{a}^T [\mathbf{Wh}_i || \mathbf{Wh}_j])$ 计算的结果是一个标量，这个标量反映了节点 (v_i) 和 (v_j) 之间的相似度或相关性。
- 如果两个节点的特征向量相似，则它们的拼接向量与 (\mathbf{a}) 向量的点积也会较大。
- 假设 (\mathbf{Wh}_i) 和 (\mathbf{Wh}_j) 相似，意味着它们在向量空间中的位置接近，即它们之间的角度较小。在这种情况下：
 1. **点积的大小**：如果 (\mathbf{Wh}_i) 和 (\mathbf{Wh}_j) 相似，它们在相同方向上的投影较长，因此 (\mathbf{Wh}_i) 和 (\mathbf{Wh}_j) 的拼接向量 $([\mathbf{Wh}_i || \mathbf{Wh}_j])$ 在相同方向上的投影也较长。因此，当与 (\mathbf{a}) 向量进行点积运算时，结果较大。
 2. **向量 (\mathbf{a}) 的作用**：向量 (\mathbf{a}) 是一个可学习的向量，用于衡量两个节点特征向量的相似度。如果 (\mathbf{a}) 在 $([\mathbf{Wh}_i || \mathbf{Wh}_j])$ 的某些维度上有较大的权重，则这些维度上的相似性会对最终的点积结果贡献更大。

2. 引入非线性：

- LeakyReLU 作为激活函数，可以进一步增强模型的表达能力，使得注意力系数更具区分性。
- LeakyReLU 保留了负值部分的信息，使得模型在处理负数特征时也能保持一定的敏感度。

公式的意义

综合起来，公式 $(\text{LeakyReLU}(\mathbf{a}^T [\mathbf{Wh}_i || \mathbf{Wh}_j]))$ 的意义在于通过一个自定义的函数来衡量两个节点之间的相似度或相关性，并通过非线性激活函数进一步增强这种衡量的效果。

理解为什么“如果两个节点的特征向量相似，则它们的拼接向量与 (\mathbf{a}) 向量的点积也会较大”可以从向量空间的角度来解释。这里的关键在于理解点积运算的性质以及特征向量相似性的含义。

在 GAT (Graph Attention Networks) 中，激活函数在计算注意力系数 (attention coefficients) 的过程中起着重要的作用。具体来说，在 GAT 的注意力机制中，使用了 LeakyReLU 激活函数来处理节点特征向量的点积结果。下面我们详细探讨激活函数在注意力系数公式中的意义。

激活函数的作用

1. 引入非线性：

- 激活函数的主要作用之一是引入非线性，使得模型能够拟合更复杂的函数关系。如果没有激活函数，整个模型就相当于一个线性模型，其表达能力有限。
- LeakyReLU 是一种常用的激活函数，它在正数部分保持线性，而在负数部分引入了一个小的斜率，避免了 ReLU 的“死区”问题。

2. 标准化输出：

- 激活函数可以对输出进行标准化，使其落在一个特定的范围内。在 GAT 中，LeakyReLU 可以确保注意力系数 (e_{ij}) 的值不会过大或过小，从而影响后续的归一化操作。
- 通过 LeakyReLU，即使输入为负数，输出也不会完全为零，而是保留了一定的信息。

3. 增强区分性：

- 激活函数可以增强不同节点之间的区分性。LeakyReLU 可以放大或缩小输入信号的幅度，从而使得不同节点之间的注意力系数更有区分性。
- 例如，对于两个相似的节点，它们的注意力系数可能会非常接近，但在经过 LeakyReLU 激活之后，它们的差异会被放大，从而更好地捕捉节点之间的关系。

4. 数值稳定性：

- 激活函数可以帮助提高数值稳定性。LeakyReLU 可以避免 ReLU 在负数区域的梯度消失问题，从而有助于训练过程中的梯度传播。

GAT (Graph Attention Networks) 是一种基于图结构数据的深度学习模型。它主要用于处理图数据中的节点分类、边预测等问题。GAT的设计目的是为了在图中捕捉局部结构信息，并且赋予不同的邻居节点不同的权重，这通过注意力机制来实现。

GAT的基本训练流程：

1. 数据准备：
 - 首先需要准备好图数据，包括**节点特征矩阵**、**邻接矩阵**（表示节点之间的连接关系）等。
 - 对于监督学习任务，还需要准备标签数据。
2. 模型构建：
 - 定义GAT层：每一层中，节点会根据其邻居的信息以及它们之间的关系进行信息传递，并通过注意力机制计算出不同邻居的重要性权重。
 - 可以堆叠多个GAT层来构建深层的网络结构。
3. 损失函数定义：
 - 根据具体任务选择合适的损失函数，如交叉熵损失用于分类任务。
4. 优化器设置：
 - 选择一个优化算法，如Adam或SGD，并设定学习率等超参数。
5. 训练过程：
 - 使用前向传播计算输出。
 - 计算损失值，并使用反向传播更新模型参数。
 - 重复上述步骤直到达到预设的训练轮数或满足停止条件。
6. 评估与调整：
 - 在验证集上评估模型性能，根据结果调整超参数或模型结构。
 - 使用测试集评估最终模型性能。

训练的模型架构：

GAT通常包含以下组件：

- 输入层：接收节点特征。
- 多个GAT层：每个GAT层都会为输入的节点特征计算一个注意力系数矩阵，然后利用这个矩阵对邻居节点的信息进行加权求和，得到新的节点特征表示。
- 输出层：根据任务需求，可能是分类层或其他类型的层。

在实际应用中，GAT可以和其他神经网络层结合使用，比如全连接层、池化层等，以适应更复杂的数据结构和任务需求。此外，为了防止过拟合，还可以在模型中加入正则化项或使用dropout技术。

MSE

均方误差 (Mean Squared Error, MSE) 是一个非常常用的损失函数，特别是在回归任务中。它也可以用于其他类型的机器学习任务，比如自编码器中的重构误差、强化学习中的价值函数估计等。以下是MSE作为损失函数的一些原因：

1. 易于理解和计算：

MSE公式简单直观，计算也非常直接。对于给定的一组预测值 (\hat{y}) 和真实值 (y)，MSE定义为：

$$[\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2]$$

这里 (n) 是样本数量，(y_i) 是真实值，(\hat{y}_i) 是预测值。

2. 对异常值敏感：

由于MSE是预测误差的平方和，因此它会对较大的预测误差（即异常值）更加敏感。这意味着模型会被驱动去减小大的误差，从而可能提高总体的预测精度。

3. 导数简单：

MSE的导数也很容易计算，这对于使用梯度下降法优化参数非常重要。MSE的导数为：

$$[\frac{\partial \text{MSE}}{\partial \hat{y}} = 2(\hat{y} - y)]$$

这意味着优化过程中计算梯度比较容易。

4. 平滑性：

MSE是一个连续可微的损失函数，这使得它非常适合使用梯度下降法进行优化。相比之下，像绝对误差这样的损失函数在某些点上不可微，可能会导致优化过程中的问题。

总之，MSE作为一种损失函数具有许多优点，使得它在多种机器学习任务中都是一个有效的选择。然而，需要注意的是，MSE对于异常值的敏感性有时可能会成为缺点，特别是当数据集中存在很多异常值时。在这种情况下，可能需要考虑使用其他损失函数，如均值绝对误差（MAE）或Huber损失等。

相似度系数

$$\beta_{mj} = \frac{\cos(h_m, h_j)}{\sum_{k=1}^N \cos(h_m, h_k)}$$

您提供的公式看起来像是一个基于余弦相似度的加权版本，用于衡量两个向量(h_m)和(h_j)之间的相似度。在这个公式中，(N)表示一组参考向量的数量，(h_k)是这一组参考向量中的第(k)个向量。让我们逐步解析这个公式：

1. 分子部分：

- ($\cos(h_m, h_j)$)是向量(h_m)和(h_j)之间的余弦相似度。余弦相似度衡量了这两个向量的方向上的相似性，而不是大小。它的取值范围在[-1, 1]之间，其中1表示完全相同的方向，0表示方向完全不同，-1表示完全相反的方向。

$$(\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{|\mathbf{A}| |\mathbf{B}|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}})$$

2. 分母部分：

- ($\sum_{k=1}^N \cos(h_m, h_k)$)是向量(h_m)与所有(N)个参考向量的余弦相似度之和。这个和反映了(h_m)与整个参考集的平均相似度。

3. 整体意义：

- 这个公式实际上是对(h_m)和(h_j)之间的余弦相似度进行了加权，权重由(h_m)与其他所有参考向量的相似度决定。换句话说，如果(h_m)与大多数参考向量都高度相似，那么(h_m)和(h_j)之间的相似度就会降低，反之亦然。

Resilience

关键节点识别（优化问题）：目标是找到一组节点，当这些节点从图中移除时，会导致网络的某种功能指标（例如连通性、传递效率等）显著下降。

问题定义

设 $G = (V, E)$ 是一个无向图，其中 V 是节点集合， E 是边集合。我们的目标是识别一个关键节点集合 $S \subseteq V$ ，使得当 S 中的节点被移除后，图 G 的某种功能指标 $f(G)$ 下降最大。

功能指标

功能指标 $f(G)$ 可以是多种不同的度量，例如：

- **连通性**：图的连通分量数目或最大连通分量的大小。
- **传递效率**：节点之间的最短路径长度的平均值。
- **中心性**：图的中心性度量，如度中心性、介数中心性或接近中心性。

优化问题

我们可以将关键节点识别问题形式化为以下优化问题：

$$S^* = \operatorname{argmax}_{S \subseteq V, |S|=k} [f(G) - f(G \setminus S)]$$

其中：

- (S^*) 是最优的关键节点集合。
- (k) 是要移除的节点数。
- ($G \setminus S$) 表示从图 (G) 中移除节点集合 (S) 后的图。
- ($f(G)$) 是原始图的功能指标。
- ($f(G \setminus S)$) 是移除节点集合 (S) 后的图的功能指标。

目标函数

目标函数 ($f(G) - f(G \setminus S)$) 衡量了移除节点集合 (S) 后功能指标的下降幅度。我们的目标是最大化这个下降幅度。

示例

假设我们选择连通性作为功能指标，具体来说，我们使用最大连通分量的大小来衡量连通性。那么，优化问题可以写成：

$$S^* = \operatorname{argmax}_{S \subseteq V, |S|=k} [\text{LCC}(G) - \text{LCC}(G \setminus S)]$$

其中：

- ($\text{LCC}(G)$) 表示图 (G) 的最大连通分量的大小。
- ($\text{LCC}(G \setminus S)$) 表示移除节点集合 (S) 后图的最大连通分量的大小。

求解方法

求解上述优化问题通常是 NP-hard 的，因此在实际应用中，常常使用启发式算法或近似算法来找到近似解。常见的方法包括：

- **贪心算法**：每次选择导致功能指标下降最多的单个节点，直到选择 (k) 个节点。
- **遗传算法**：通过进化算法搜索最优的节点集合。
- **模拟退火**：通过模拟物理退火过程来寻找全局最优解。

总结

关键节点识别问题可以形式化为一个优化问题，目标是在移除特定数量的节点后，使图的功能指标下降最大。通过选择合适的功能指标和求解方法，可以有效地识别出这些关键节点。

$$G = (V, E)$$

$$S \subseteq V, |S| = 1$$

$T \subseteq E$, $|T| = 1$

$$S^* = \operatorname{argmax}_{S \subseteq V, |S|=k} [f(G) - f(G \setminus S)]$$

$$T^* = \operatorname{argmax}_{T \subseteq E, |T|=k} [f(G) - f(G \setminus T)]$$

$f(G)$ 由图鲁棒性度量，在评估时，每次删除一个节点/链接，即 $k = 1$ 。

(根据节点/链接删除的顺序分配临界性超出了本文的范围，将作为未来的工作。)

解决关键节点识别问题的方法，通过整合节点/链接的局部邻域信息来实现。该方法假定存在一种**功能性关系/映射**，将局部拓扑和特征与节点的重要程度联系起来。如GraphSAGE，通过训练从数据中学习这种映射。

想要学习的节点 (u) 的重要性分数 (r_u)，第一步是生成节点 (u) 的特征向量 (h_u) 以及其邻域内所有节点 ($v \in N(u)$) 的特征向量 (h_v)，其中 ($N(u)$) 表示节点 (u) 的邻域。根据该方法的假设，这些特征与重要性分数 (r_u) 之间的潜在映射可以用以下方式表示：

$$r_u = f(h_u, h_v), \forall v \in N(u)$$

这意味着节点 (u) 的重要性分数是由其自身及邻域内其他节点的特征共同决定的，通过函数 (f) 来实现这种映射。

图稳健性归纳学习(ILGR)

$$\hat{f}(h_u, h_v, N(u))$$

节点嵌入

ILGR的第一个模块通过利用图结构和节点目标（临界性）分数来学习每个节点的嵌入向量。这是通过一种方式实现的，即在图空间中靠近的节点也在嵌入空间中靠近。初始化时，每个节点的嵌入仅由节点的度数组成，后面跟着预定义数量的1。节点嵌入的学习基于GraphSAGE，这在前一节中简要描述。然而，在实施中进行了一些修改。节点嵌入模块进一步细分为两个任务。第一个任务基于节点的邻域节点的表示组合来学习每个节点的表示，参数化为一个量K，该量量化了节点邻域的大小。具体来说，参数K控制要考虑的邻域跳数的数量。例如，如果K=2，则离选定节点2跳远的所有节点都将被认为是邻居。这定义了节点v的邻域为：

$$N(v) = \{u : D(u, v) \leq K, \forall u \in G\}$$

其中 $D(u, v)$ 是一个返回节点u和v之间最小距离的函数。在定义邻域之后，使用聚集器函数为每个邻居的嵌入分配权重，并为所选节点创建邻域嵌入。不像以前的工作[28,29]，其中权重是预先定义的，这项工作使用注意力机制自动学习对应于每个邻居节点的权重，如[30]所示：

$$h_{N(v)}^l = \operatorname{Attention}(Q^l h_k^{l-1}) \quad \forall k \in N(v)$$

其中，(Q^l) 是第l层的注意力权重，(h_k^{l-1}) 是k节点在第(l-1)层的嵌入。最后，节点v在第l层的嵌入可以通过以下公式计算得出：

$$h_v^l = \operatorname{ReLU}\left(W^l[h_v^{l-1} \| h_v^{l-2} \| h_{N(v)}^l]\right)$$

其中，(W^l) 是第l层的权重矩阵，(|) 表示拼接操作。

边嵌入

回归模块

这段文字描述了一个回归模块的结构和功能。以下是详细解释：

回归模块接收嵌入模块的输出。

该模块由多个前馈层组成，这些层对输入进行非线性变换，并最终生成一个标量，表示节点的关键性得分。

第 (m) 层的输出可以表示为：

$$y_v^m = f(W^m y_v^{m-1} + b^m)$$

其中：

- (W^m) 和 (b^m) 分别是第 (m) 层的权重和偏置。
- (f) 是激活函数，如ReLU、Softmax 等。
- (y_v^m) 是第 (m) 层的输出。

前馈层通过非线性变换来处理输入数据。

最终输出是一个标量，用于表示节点的关键性得分。

这段文字描述了一个模型的目标是通过最小化排名中的逆序数量来优化节点的排序。具体来说，它希望减少那些在真实情况中应该排在前面的节点被错误地排在后面的情况。

损失函数 L_{ij} 定义如下：

$$L_{ij} = -f(r_{ij}) \log(\sigma(\hat{y}_{ij})) - (1 - f(r_{ij})) \log(1 - \sigma(\hat{y}_{ij}))$$

其中，

- (r_i) 是节点 i 的实际重要性分数。
- ($r_{ij} = r_i - r_j$) 是节点 i 和 j 之间的实际排名顺序差值。
- ($\hat{y}_{ij} = \hat{y}_i - \hat{y}_j$) 是模型预测的节点 i 和 j 之间的排名顺序差值。
- (f) 是一个 sigmoid 函数。
- (σ) 是 sigmoid 函数，用于将预测值转换为概率形式。
- 在公式 $L_{ij} = -f(r_{ij}) \log(\sigma(\hat{y}_{ij})) - (1 - f(r_{ij})) \log(1 - \sigma(\hat{y}_{ij}))$ 中，`log` 函数具有重要的意义。这个公式实际上是一个二元交叉熵损失函数 (Binary Cross-Entropy Loss)，用于评估模型预测的概率与真实标签之间的差异。下面详细解释每个部分的意义：

公式解释

1. Sigmoid 函数：

$$\sigma(\hat{y}_{ij}) = \frac{1}{1+e^{-\hat{y}_{ij}}}$$

Sigmoid 函数将模型的输出 (\hat{y}_{ij}) 映射到 (0, 1) 区间内，表示预测的概率。

2. 真实标签：

$$f(r_{ij})$$

($f(r_{ij})$) 是一个函数，通常表示真实的标签。在二分类问题中，($f(r_{ij})$) 的取值为 0 或 1。

3. 交叉熵损失：

$$L_{ij} = -f(r_{ij}) \log(\sigma(\hat{y}_{ij})) - (1 - f(r_{ij})) \log(1 - \sigma(\hat{y}_{ij}))$$

交叉熵损失的意义

1. 当 ($f(r_{ij}) = 1$) 时：

$$L_{ij} = -\log(\sigma(\hat{y}_{ij}))$$

- 如果 ($\sigma(\hat{y}_{ij})$) 接近 1 (即模型预测正确)，($-\log(\sigma(\hat{y}_{ij}))$) 会接近 0，因此损失 (L_{ij}) 会很小。
- 如果 ($\sigma(\hat{y}_{ij})$) 接近 0 (即模型预测错误)，($-\log(\sigma(\hat{y}_{ij}))$) 会接近负无穷大，因此损失 (L_{ij}) 会很大。

2. 当 ($f(r_{ij}) = 0$) 时：

$$L_{ij} = -\log(1 - \sigma(\hat{y}_{ij}))$$

- 如果 $\hat{\sigma}(\hat{y}_{ij})$ 接近 0 (即模型预测正确), $\log(1 - \hat{\sigma}(\hat{y}_{ij}))$ 会接近 0, 因此损失 L_{ij} 会很小。
- 如果 $\hat{\sigma}(\hat{y}_{ij})$ 接近 1 (即模型预测错误), $\log(1 - \hat{\sigma}(\hat{y}_{ij}))$ 会接近负无穷大, 因此损失 L_{ij} 会很大。

Log 函数的意义

1. 对数函数的性质:

- 当 x 接近 1 时, $\log(x)$ 接近 0。
- 当 x 接近 0 时, $\log(x)$ 接近负无穷大。

2. 惩罚机制:

- `log` 函数在损失函数中起到惩罚错误预测的作用。当模型的预测与真实标签相差较大时, `log` 函数的值会变得非常大, 从而增大损失值。
- 相反, 当模型的预测与真实标签非常接近时, `log` 函数的值会接近 0, 从而减小损失值。

二元交叉熵损失的直观理解

- 正确预测: 当模型正确预测时, 损失值会很小, 因为 `log` 函数在接近 1 时的值接近 0。
- 错误预测: 当模型错误预测时, 损失值会很大, 因为 `log` 函数在接近 0 时的值接近负无穷大。

这个损失函数的作用是对所有训练数据对进行聚合, 并通过优化该损失来更新模型权重。最终的训练损失可以表示为:

$$Loss = \sum_{(i,j) \in E} L_{ij}$$

其中 (i, j) 表示属于图链接集 E 中的一个边对。

一旦学习了权重, 就可以根据测试节点/链路的特征和邻域信息来预测其嵌入向量和相应的节点/链路得分。

优化后节点个数 $N_{op} = 14$

$$R_g = \frac{2}{N_{op}} \sum_{i=1}^{N_{op}+1-c} \frac{1}{\lambda_i}$$

$$N_r = N - N_{op}$$

$$r_{(1)}, r_{(2)}, \dots, r_{(n)}$$

$$r_{(1)} \leq r_{(2)} \leq \dots \leq r_{(n)}$$

$$r_i = softmax(R_{g_1}, \dots, R_{g_{N_r}})$$

$$\hat{y}_i = softmax(y_i, \dots, y_{N_r})$$

Softmax

`softmax` 函数能够将一个向量转换为概率分布, 并且在软排序中实现平滑的排序效果, 其背后的原理可以从以下几个方面来理解:

1. Softmax 函数的定义

`softmax` 函数的定义如下：

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

其中， (\mathbf{x}) 是一个长度为 (n) 的向量， $\text{softmax}(\mathbf{x})_i$ 表示向量 (\mathbf{x}) 经过 `softmax` 函数变换后第 (i) 个元素的值。

2. 归一化

`softmax` 函数将每个元素 (x_i) 映射到一个范围在 $((0, 1))$ 之间的值，并且所有这些值的和为 1。这意味着 `softmax` 函数将向量 (\mathbf{x}) 转换为一个概率分布。

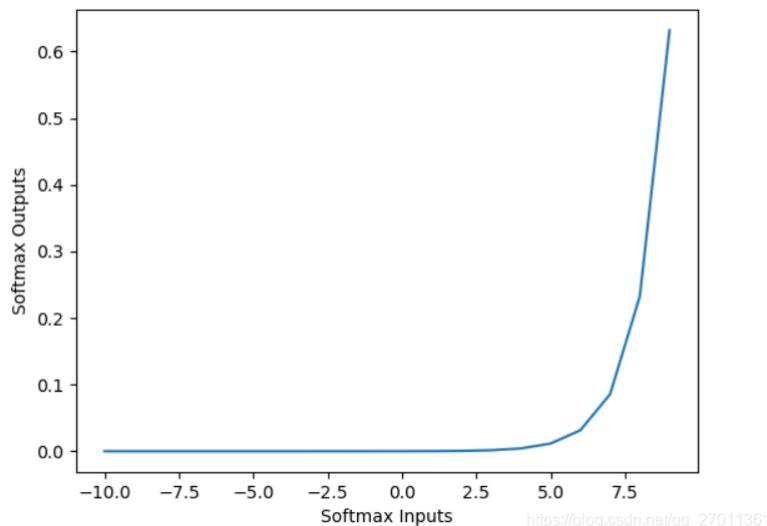
3. 温度参数 (τ)

在软排序中，通常会引入一个温度参数 (τ) ，使得 `softmax` 函数的定义变为：

$$\text{softmax}(\mathbf{x}/\tau)_i = \frac{e^{x_i/\tau}}{\sum_{j=1}^n e^{x_j/\tau}}$$

温度参数 (τ) 控制着 `softmax` 函数的平滑程度：

- **当 (τ) 较小时：** `softmax` 函数的结果更加尖锐，接近于 one-hot 编码。这是因为较小的 (τ) 会使指数项之间的差异放大，导致较大的值占据主导地位，而其他值接近于 0。
- **当 (τ) 较大时：** `softmax` 函数的结果更加平滑，每个值都比较接近。这是因为较大的 (τ) 会使指数项之间的差异缩小，从而使每个值都有一定的概率。



Reference indicators

1. Natural Connectivity ↑

自然连通性 (Natural Connectivity) 是一种衡量网络或图的鲁棒性和连通性的指标。它不仅考虑了网络的连通性，还考虑了节点之间的路径多样性。自然连通性越高，表明网络在面对节点或边的删除时更加鲁棒。

自然连通性的定义

自然连通性 ($\phi(G)$) 定义为：

$$\phi(G) = \frac{1}{n} \sum_{k=1}^n \log(\lambda_k)$$

其中：

- (G) 是一个图。
- (n) 是图的节点数。
- (λ_k) 是图的拉普拉斯矩阵的第 (k) 个特征值 (不包括零特征值)。

计算自然连通性

要计算自然连通性，你需要以下几个步骤：

1. **构建图的邻接矩阵**：表示图中节点之间的连接关系。
2. **计算图的拉普拉斯矩阵**：拉普拉斯矩阵 (L) 定义为 ($L = D - A$)，其中 (D) 是度矩阵（对角线上的元素是每个节点的度），(A) 是邻接矩阵。
3. **计算拉普拉斯矩阵的特征值**：特征值反映了图的结构特性。
4. **计算自然连通性**：根据上述公式计算自然连通性。

示例代码

以下是一个使用 Python 和 NetworkX 库来计算自然连通性的示例：

```
import networkx as nx
import numpy as np

# 创建一个示例图
G = nx.Graph()
G.add_edges_from([(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)])

# 计算图的拉普拉斯矩阵
L = nx.laplacian_matrix(G).todense()

# 计算拉普拉斯矩阵的特征值
eigenvalues = np.linalg.eigvals(L)

# 去除零特征值
non_zero_eigenvalues = eigenvalues[eigenvalues > 1e-10]

# 计算自然连通性
n = G.number_of_nodes()
natural_connectivity = (1 / n) * np.sum(np.log(non_zero_eigenvalues))

print(f"Natural Connectivity: {natural_connectivity}")
```

解释

1. **创建图**：使用 `networkx` 库创建一个简单的无向图。
2. **计算拉普拉斯矩阵**：使用 `nx.laplacian_matrix` 函数计算图的拉普拉斯矩阵。
3. **计算特征值**：使用 `numpy.linalg.eigvals` 函数计算拉普拉斯矩阵的特征值。
4. **去除零特征值**：自然连通性的定义中不包括零特征值，因此需要过滤掉接近零的特征值。
5. **计算自然连通性**：根据公式计算自然连通性。

注意事项

- **数值稳定性**: 在计算特征值时，可能会遇到数值稳定性问题，特别是当特征值非常接近零时。因此，通常会设置一个小阈值（如 $1e-10$ ）来过滤掉这些值。
- **图的连通性**: 自然连通性适用于连通图。如果图不是连通的，可能需要分别计算每个连通分量的自然连通性。

希望这些信息对你理解自然连通性及其计算方法有所帮助！如果有其他问题或需要进一步的解释，请随时提问。

衡量网络或图的鲁棒性和连通性的指标

自然连通性 (Natural Connectivity) 作为一种衡量网络或图的鲁棒性和连通性的指标，具有以下几个关键特点和优势，使其成为一种有效的度量工具：

1. 考虑了所有路径

自然连通性不仅考虑了图的连通性，还考虑了节点之间的路径多样性。这意味着它不仅关注图是否连通，还关注连通的程度和路径的丰富性。这种全面的视角使得自然连通性能够更准确地反映网络的结构特性。

2. 基于谱理论

自然连通性的定义基于图的拉普拉斯矩阵的特征值。拉普拉斯矩阵的特征值包含了图的大量结构性信息，如连通性、聚类性、路径长度等。通过分析这些特征值，可以更深入地了解图的拓扑结构。

3. 鲁棒性

自然连通性能够有效衡量网络在面对节点或边的删除时的鲁棒性。具体来说，自然连通性较高的网络在删除某些节点或边后，仍然能够保持较好的连通性。这是因为高自然连通性意味着网络中存在多条路径连接节点，即使某些路径被破坏，网络仍能通过其他路径保持连通。

4. 数学性质

自然连通性的定义具有良好的数学性质，使其易于计算和分析。通过特征值的对数平均，自然连通性提供了一个平滑且稳定的度量，能够在不同规模和结构的网络之间进行比较。

5. 反映全局和局部特性

自然连通性既反映了网络的全局连通性，也反映了局部连通性。全局连通性是指整个网络的连通程度，而局部连通性是指网络中各个子区域的连通性。自然连通性通过特征值的组合，能够同时捕捉这两种特性。

6. 对随机扰动的敏感性

自然连通性对网络中的随机扰动（如节点或边的随机删除）具有较高的敏感性。这意味着它可以有效检测网络在受到小概率事件影响时的性能变化，这对于评估网络的可靠性和稳定性非常重要。

7. 与其他度量的互补性

自然连通性可以与其他网络度量（如平均路径长度、聚类系数、介数中心性等）互补使用，提供更全面的网络特性描述。通过结合多种度量，可以更准确地评估网络的鲁棒性和连通性。

数学解释

自然连通性 ($\phi(G)$) 的定义为：

$$\phi(G) = \frac{1}{n} \sum_{k=1}^n \log(\lambda_k)$$

其中：

- (G) 是一个图。
- (n) 是图的节点数。
- (λ_k) 是图的拉普拉斯矩阵的第 (k) 个特征值（不包括零特征值）。

为什么特征值重要？

- **特征值的物理意义：**拉普拉斯矩阵的特征值反映了图的结构性质。例如，**最小非零特征值**（代数连通性）反映了图的连通性强度，而**较大的特征值反映了图的局部结构特性**。
- **对数平均：**通过对特征值取对数平均，自然连通性能够平滑地反映图的整体连通性，避免了单一特征值的极端影响。

总结

自然连通性通过综合考虑图的路径多样性和结构特性，提供了一种有效的度量网络鲁棒性和连通性的方法。它不仅能够反映网络的全局连通性，还能捕捉局部连通性，使其成为评估网络性能的重要工具。希望这些解释能帮助你更好地理解自然连通性的重要性。如果有其他问题或需要进一步的解释，请随时提问。

特征值log的求和是什么含义

特征值的对数求和在自然连通性的定义中起着重要的作用。为了更好地理解这一点，我们需要从几个角度来解释其含义：

1. 数学背景

特征值的意义

- **拉普拉斯矩阵：**对于一个图 (G) ，其拉普拉斯矩阵 (L) 定义为 $(L = D - A)$ ，其中 (D) 是度矩阵（对角线上的元素是每个节点的度）， (A) 是邻接矩阵。
- **特征值：**拉普拉斯矩阵的特征值 $(\lambda_1, \lambda_2, \dots, \lambda_n)$ 包含了图的大量结构性信息。特别是，**最小非零特征值**（代数连通性）反映了图的连通性强度，而**较大的特征值反映了图的局部结构特性**。

对数的性质

- **对数函数：**对数函数 $(\log(x))$ 是一个单调递增函数，且 $(\log(1) = 0)$ 。对数函数的一个重要性质是它可以将**乘法转换为加法**，即 $(\log(a \cdot b) = \log(a) + \log(b))$ 。
- **对数平均：**通过对特征值取对数再求平均，可以平滑地**反映特征值的分布**，避免了单一特征值的极端影响。

2. 对数求和的含义

平滑化

- **平滑化：**对数函数可以将特征值的乘积转换为对数的和，从而平滑地反映特征值的分布。这意味着**即使某些特征值非常大或非常小，对整体结果的影响也会被平滑化**。
- **避免极端值：**对数函数可以**避免特征值中的极端值对结果产生过大的影响**，从而使自然连通性更加稳定和可靠。

信息熵

- **信息熵：**对数求和可以看作是一种信息熵的形式。在信息论中，信息熵用于衡量信息的不确定性。类似地，**特征值的对数求和可以看作是图的结构信息的不确定性的度量**。
- **多样性：**对数求和反映了特征值的多样性。如果**特征值分布较为均匀，对数求和的结果会较大**，反之则较小。这表明图的结构较为均匀，路径多样性较高。

3. 物理意义

连通性

- **连通性**: 特征值的对数求和反映了图的连通性。较大的对数求和值意味着图的连通性较强，路径多样性较高。
- **鲁棒性**: 自然连通性较高的图在面对节点或边的删除时，能够更好地保持连通性。这是因为高自然连通性意味着图中存在多条路径连接节点，即使某些路径被破坏，网络仍能通过其他路径保持连通。

4. 总结

特征值的对数求和在自然连通性的定义中起到了平滑化和信息熵的作用，能够更全面地反映图的结构特性。具体来说：

- **平滑化**: 避免了单一特征值的极端影响，使自然连通性更加稳定和可靠。
- **信息熵**: 反映了特征值的多样性，从而衡量图的结构信息的不确定性。
- **连通性**: 反映了图的连通性和路径多样性，高自然连通性意味着图在面对节点或边的删除时具有较强的鲁棒性。

特征值的连乘积

特征值的连乘积在数学和图论中有着重要的意义，特别是在分析图的结构和性质时。特征值的连乘积通常与图的拉普拉斯矩阵相关联，可以提供关于图的一些深刻见解。以下是特征值连乘积的一些主要含义和应用：

1. 行列式的值

对于一个 ($n \times n$) 的矩阵 (A)，其特征值的连乘积等于该矩阵的行列式 (determinant)。

对于图的拉普拉斯矩阵 (L)，其行列式具有特殊的意义：

$$\det(L) = \prod_{i=1}^n \lambda_i$$

其中，(λ_i) 是拉普拉斯矩阵的特征值。对于连通图，拉普拉斯矩阵的一个特征值总是 0，因此行列式实际上等于非零特征值的连乘积。

2. 图的生成树数量

对于一个无向图 (G)，其生成树的数量可以通过拉普拉斯矩阵的非零特征值的连乘积来计算。具体来说，生成树的数量 ($T(G)$) 可以通过以下公式计算：

$$T(G) = \frac{1}{n} \prod_{i=1}^{n-1} \lambda_i$$

其中，($\lambda_1, \lambda_2, \dots, \lambda_{n-1}$) 是拉普拉斯矩阵的非零特征值，(n) 是图的节点数。

3. 图的谱半径

图的谱半径 (spectral radius) 是拉普拉斯矩阵的最大特征值。虽然谱半径本身不是特征值的连乘积，但它与特征值的连乘积一起可以提供关于图的结构和性质的更多信息。谱半径可以反映图的连通性和扩张性。

4. 图的鲁棒性和连通性

特征值的连乘积可以用于评估图的鲁棒性和连通性。高连乘积值通常意味着图的连通性较强，路径多样性较高，因此在网络受到攻击或节点故障时，图能够更好地保持连通性。

5. 图的稳定性

在控制理论和动力系统中，特征值的连乘积可以用于评估系统的稳定性。对于一个线性系统，特征值的连乘积可以反映系统的动态行为和稳定性。

2. MSE ↓

(1) 均方误差 (MSE): MSE 用于比较选择出的节点经重构后的特征和原始特征之间的差异。MSE 越低，说明通过所选节点重构原始信息的能力越强。

$$MSE = \frac{1}{N} \sum_{i=1}^N (x_i - \hat{x}_i)^2 \quad (3-41)$$

其中， x_i 为节点 v_i 的原始特征， \hat{x}_i 为重构特征。

3. AEC ↓

(2) 平均能量消耗 (Average Energy Consumption, AEC): AEC 度量了传感网的总体能量消耗。AEC 越低，说明网络中的能量效率越高，有利于网络的可持续性。

1.

$$AEC = \frac{1}{N} \sum_{i=1}^N \left(E_0 - L \sum_{N_i} E_T - L \sum_{N_i} E_R \right) \quad (3-42)$$

其中， E_0 为传感器节点的初始能量， L 为网络寿命， E_T 为发送能耗， E_R 为接收能耗。根据无线传感网的无线传输能耗模型 (Radio Energy Dissipation Model, REDM) [93]，每个传感器节点在单次通信中的发送能耗和接收能耗分别为：

$$E_T = k(E_{elec} + \beta d^\alpha) \quad (3-43)$$

$$E_R = kE_{elec} \quad (3-44)$$

其中， E_{elec} 为单位信息的电子损耗， k 为信息量， β 为单位信息的功放损失， α 为传播衰减指数， d 为通信距离。

2. 重新定义 AEC 和网络寿命

1. 节点的初始能量：

- 每个节点的初始能量 (E_0)。

2. 通信能耗：

- 发送能耗 (E_{tx}): 每次发送单位数据所需的能量。
- 接收能耗 (E_{rx}): 每次接收单位数据所需的能量。

3. 数据传输频率：

- 每个节点的数据传输频率 (f) (单位时间内发送的数据包数量)。

4. 数据包大小：

- 每个数据包的大小 (l) (单位: 比特或字节)。

5. 网络中的节点数量:

- 网络中的传感器节点数量 (N)。

计算每个节点的总能耗

每个节点在单位时间内的总能耗 (E_{total}) 可以表示为: $E_{\text{total}} = f \cdot l \cdot (E_{\text{tx}} + E_{\text{rx}})$

计算网络寿命

网络寿命 (L) 可以表示为:

$$L = \frac{E_0}{E_{\text{total}}}$$

计算 AEC

AEC 是指在给定的时间内, 每个节点的平均能量消耗。我们可以通过以下公式来计算 AEC:

$$\text{AEC} = \frac{\sum_{i=1}^N E_{\text{total},i}}{N}$$

其中, ($E_{\text{total},i}$) 是第 (i) 个节点的总能耗。

1. 参数:

- E_0 : 每个节点的初始能量 (焦耳)。
- E_{tx} : 发送能耗 (nJ/比特)。
- E_{rx} : 接收能耗 (nJ/比特)。
- f : 每个节点的数据传输频率 (次/秒)。
- l : 每个数据包的大小 (比特)。
- N : 网络中的传感器节点数量。

2. 计算总能耗:

- E_{total} : 每个节点在单位时间内的总能耗 (焦耳)。

3. 计算 AEC:

- AEC : 每个节点的平均能量消耗 (焦耳/秒)。

4. 计算网络寿命:

- L : 网络寿命 (秒)。

$$E_{\text{tx}} = k(E_{\text{elec}} + \beta d^\alpha)$$

4. KL MI

评价指标在第三章的三个指标的基础上，本章从信息角度增加两个指标来评估原始网络的特征和优化后网络的特征所包含语义信息的差异：互信息（Mutual Information, MI）和 KL 散度。MI 用于衡量变量之间共享信息的数量，而 KL 则衡量概率分布之间的相似性。较高的 MI 和较低的 KL 表明优化方法在过程中有效地保留了相关信息。

$$MI = \sum_x \sum_{x'} p(X, X') \log \frac{p(X, X')}{p(X)p(X')} \quad (4-40)$$

$$KL = \sum_x p(X) \log \frac{p(X)}{p(X')} \quad (4-41)$$

其中， X 为原始网络特征， X' 为优化后网络特征， $p(\square)$ 表示特征分布密度。

5. Ds ↓

稀疏性偏差 (Sparseness Discrepancy) 反映了优化后传感网节点分布的均匀性[94]。均匀的节点分布象征着更好的优化以及资源分配，Ds越大，说明节点分布得越均匀。Ds 越大，说明节点分布得越不均匀

$$Ds = \frac{R_{st}}{r_{max}}$$

其中， r_{max} 为传感器节点的最大空洞半径， R_{st} 为理想布点标准半径：

$$R_{st} = \left\{ \frac{\Gamma\left(\frac{m}{2} + 1\right) \cdot \frac{1}{n}}{\pi^{\frac{m}{2}}} \right\}^{\frac{1}{m}} \quad (3-46)$$

其中， $\Gamma(x) = \int_0^{+\infty} t^{x-1} e^{-t} dt$ 为伽马函数。

最大空洞半径

在图论中，“最大空洞半径” (Maximum Hole Radius) 通常是指在一个图中，最大的没有连接的子图 (即空洞) 的半径。这个概念在图论和网络分析中非常重要，特别是在研究图的连通性和结构特征时。

定义

1. 空洞 (Hole) :

- 在图论中，一个空洞是指一个没有连接的子图。具体来说，一个空洞是一个诱导子图，其中任意两个顶点之间的最短路径长度都大于某个给定的阈值。

2. 半径 (Radius) :

- 一个图的半径是指从图中某个中心顶点到图中最远顶点的最短路径长度。对于一个空洞，其半径可以定义为从空洞内部的某个中心顶点到空洞边界最远顶点的最短路径长

度。

计算最大空洞半径

1. 识别空洞：

- 首先，需要识别图中的空洞。这可以通过多种方法实现，例如使用深度优先搜索（DFS）、广度优先搜索（BFS）或其他图遍历算法来查找未连接的子图。

2. 计算空洞的半径：

- 对于每个识别出的空洞，计算其半径。这通常涉及到找到空洞内部的某个中心顶点，并计算从该中心顶点到空洞边界最远顶点的最短路径长度。

3. 确定最大空洞半径：

- 在所有识别出的空洞中，找到半径最大的空洞，其半径即为最大空洞半径。

示例代码

以下是一个简单的 Python 示例，使用 NetworkX 库来计算图的最大空洞半径：

```
import networkx as nx

def find_holes(graph):
    # 使用 BFS 查找连通分量
    holes = []
    visited = set()

    for node in graph.nodes:
        if node not in visited:
            component = set(nx.bfs_tree(graph, node).nodes)
            visited.update(component)
            holes.append(component)

    return holes

def compute_radius(hole, graph):
    # 计算空洞的半径
    max_radius = 0
    for node in hole:
        distances = nx.single_source_shortest_path_length(graph, node)
        radius = max(distances.values())
        max_radius = max(max_radius, radius)
    return max_radius

def maximum_hole_radius(graph):
    holes = find_holes(graph)
    max_radius = 0

    for hole in holes:
        hole_graph = graph.subgraph(hole)
        radius = compute_radius(hole, hole_graph)
        max_radius = max(max_radius, radius)

    return max_radius

# 创建一个示例图
G = nx.Graph()
G.add_edges_from([(1, 2), (2, 3), (3, 4), (4, 1), (5, 6), (6, 7), (7, 8),
(8, 5)])
```

```

# 计算最大空洞半径
max_radius = maximum_hole_radius(G)
print(f"最大空洞半径: {max_radius}")

```

解释

1. `find_holes` 函数:
 - 使用广度优先搜索 (BFS) 查找图中的连通分量，每个连通分量被视为一个空洞。
2. `compute_radius` 函数:
 - 计算给定空洞的半径。通过从空洞中的每个顶点出发，计算到其他顶点的最短路径长度，找到最大值。
3. `maximum_hole_radius` 函数:
 - 遍历所有空洞，计算每个空洞的半径，并返回最大值。

总结

通过上述方法，你可以计算图的最大空洞半径。这个概念在图论和网络分析中非常有用，特别是在研究图的连通性和结构特征时。如果有更多问题或需要进一步的帮助，请随时提问！

稀疏性偏差 (D_s) 的定义

稀疏性偏差 (D_s) 的计算公式为：

$$D_s = \frac{R_{st}}{r_{max}}$$

其中：

- (R_{st}) 是理想布点标准半径，通常是一个固定的值，表示节点之间期望的距离或覆盖范围。
- (r_{max}) 是最大空洞半径，表示图中最大连通分量的半径。

(D_s) 的意义

1. (D_s) 越大：
 - 表示 (R_{st}) 相对于 (r_{max}) 较大。
 - 这意味着实际的节点分布相对于理想布点标准来说较为稀疏，即节点之间的距离较大，存在较大的空洞。
 - 因此，(D_s) 越大，说明节点分布得越不均匀。
2. (D_s) 越小：
 - 表示 (R_{st}) 相对于 (r_{max}) 较小。
 - 这意味着实际的节点分布相对于理想布点标准来说较为密集，即节点之间的距离较小，空洞较小。
 - 因此，(D_s) 越小，说明节点分布得越均匀。

举例说明

假设我们有一个传感器网络，随着节点数量的增加，节点之间的连接变得更加密集，最大空洞半径 (r_{max}) 逐渐增加，但增加的速度较慢。例如：

- **初始状态：**1个节点，($r_{max} = 0$)，($R_{st} = 10$)，($D_s = \frac{10}{0 + \epsilon} \approx 10000000000$)
 - 节点分布非常不均匀，因为只有一个节点，没有其他节点与其连接。
- **中期状态：**10个节点，($r_{max} = 2$)，($R_{st} = 10$)，($D_s = \frac{10}{2} = 5$)
 - 节点分布有所改善，但仍不够均匀。

- **后期状态**: 100个节点, ($r_{\max} = 4$), ($R_{st} = 10$), ($D_s = \frac{10}{4} = 2.5$)

◦ 节点分布更加均匀, 因为节点之间的距离较小, 空洞较小。

结论

- (D_s)**越大**, 说明节点分布得越不均匀。
- (D_s)**越小**, 说明节点分布得越均匀。

因此, 随着图的规模增大, 如果节点分布更加均匀, 最大空洞半径 (r_{\max}) 会增加, 但增加的速度较慢, 导致 (D_s) 逐渐变小。这反映了图的连通性和节点分布的改善, 使得网络更加密集和均匀。如果有更多问题或需要进一步的帮助, 请随时提问!

Rsr

要实现给定公式的 (R_{st}) 的计算, 我们可以使用 Python 编程语言, 并利用数学库如 NumPy 来进行伽马函数的计算。以下是根据公式(3-46) 实现 (R_{st}) 的 Python 代码:

首先, 确保已经安装了 NumPy 库。如果没有安装, 可以通过以下命令安装:

```
pip install numpy
```

然后, 你可以编写如下代码来计算 (R_{st}):

```
import numpy as np

def calculate_R_st(m, n):
    gamma_value = np.math.gamma((m/2) + 1)
    denominator = ((gamma_value / np.pi**((m/2))) * (1/n)**(1/m))

    return denominator

# 参数 m 和 n 的值
m = 2 # 根据实际情况调整
n = 100 # 根据实际情况调整

# 计算 R_st
R_st = calculate_R_st(m, n)
print("R_st:", R_st)
```

在这个代码中:

1. 我们导入了 NumPy 库并使用 `np.math.gamma()` 函数来计算伽马函数。
2. 定义了一个名为 `calculate_R_st` 的函数, 接受参数 `m` 和 `n`。
3. 使用给定的公式计算 (R_{st}) 并返回结果。
4. 设置了参数 `m` 和 `n` 的值, 并调用 `calculate_R_st` 函数来计算 (R_{st})。

请根据你的具体情况调整 `m` 和 `n` 的值。如果你有任何疑问或需要进一步帮助, 请告诉我!

在公式(3-46)中, 参数 (`m`) 和 (`n`) 具有特定的含义, 它们通常与传感器网络的几何分布和节点数量有关。具体来说:

参数含义

1. (m):

- (m) 通常表示空间的维度。例如：
 - 在二维空间中, ($m = 2$)。
 - 在三维空间中, ($m = 3$)。
- 这个参数决定了空间的几何特性, 影响节点之间的距离和分布。

2. (n):

- (n) 表示传感器节点的数量。
- 这个参数反映了网络中节点的总数, 直接影响节点的分布密度。

公式 (3-46) 的解释

根据公式 (3-46), 理想布点标准半径 (R_{st}) 的计算公式为:

$$R_{st} = \left(\frac{\Gamma\left(\frac{m}{2} + 1\right)}{\pi^{\frac{m}{2}}} \cdot \frac{1}{n} \right)^{\frac{1}{m}}$$

其中:

- ($\Gamma(x)$) 是伽马函数, 它是阶乘的推广, 对于正整数 (k), ($\Gamma(k) = (k-1)!$)。
- (π) 是圆周率, 约等于 3.14159。
- (m) 是空间的维度。
- (n) 是传感器节点的数量。

代码实现

下面是完整的 Python 代码, 用于计算 (R_{st}):

```
import numpy as np

def calculate_R_st(m, n):
    # 计算伽马函数值
    gamma_value = np.math.gamma((m / 2) + 1)

    # 计算分母
    denominator = (gamma_value / np.pi ** (m / 2)) * (1 / n)

    # 计算 R_st
    R_st = denominator ** (1 / m)

    return R_st

# 参数 m 和 n 的值
m = 2 # 二维空间
n = 100 # 100 个传感器节点

# 计算 R_st
R_st = calculate_R_st(m, n)
print("R_st:", R_st)
```

解释

1. 伽马函数:

- o `np.math.gamma((m / 2) + 1)` 计算 ($\Gamma\left(\frac{m}{2} + 1\right)$)。

2. 分母计算:

- o `denominator = (gamma_value / np.pi**(m / 2)) * (1 / n)` 计算分母部分。

3. R_st 计算:

- o `R_st = denominator ** (1 / m)` 计算 (R_{st})。

示例

假设我们在二维空间中 ($m = 2$) 有 100 个传感器节点 ($n = 100$)，计算 (R_{st}):

```
m = 2 # 二维空间
n = 100 # 100 个传感器节点

R_st = calculate_R_st(m, n)
print("R_st:", R_st)
```

运行结果可能是：

```
R_st: 0.17841241161527818
```

总结

- (m): 空间的维度 (例如, 二维空间 ($m = 2$), 三维空间 ($m = 3$))。
- (n): 传感器节点的数量。

通过调整 (m) 和 (n) 的值, 你可以计算不同场景下的 (R_{st}), 从而评估传感器节点的分布均匀性。如果有更多问题或需要进一步的帮助, 请随时提问!

6. R_g

$$R_g = \frac{2}{N-1} \sum_{i=1}^{N-1} \frac{1}{\lambda_i}$$

不同模型—Node feature distribution

Test Results

Similarity Score

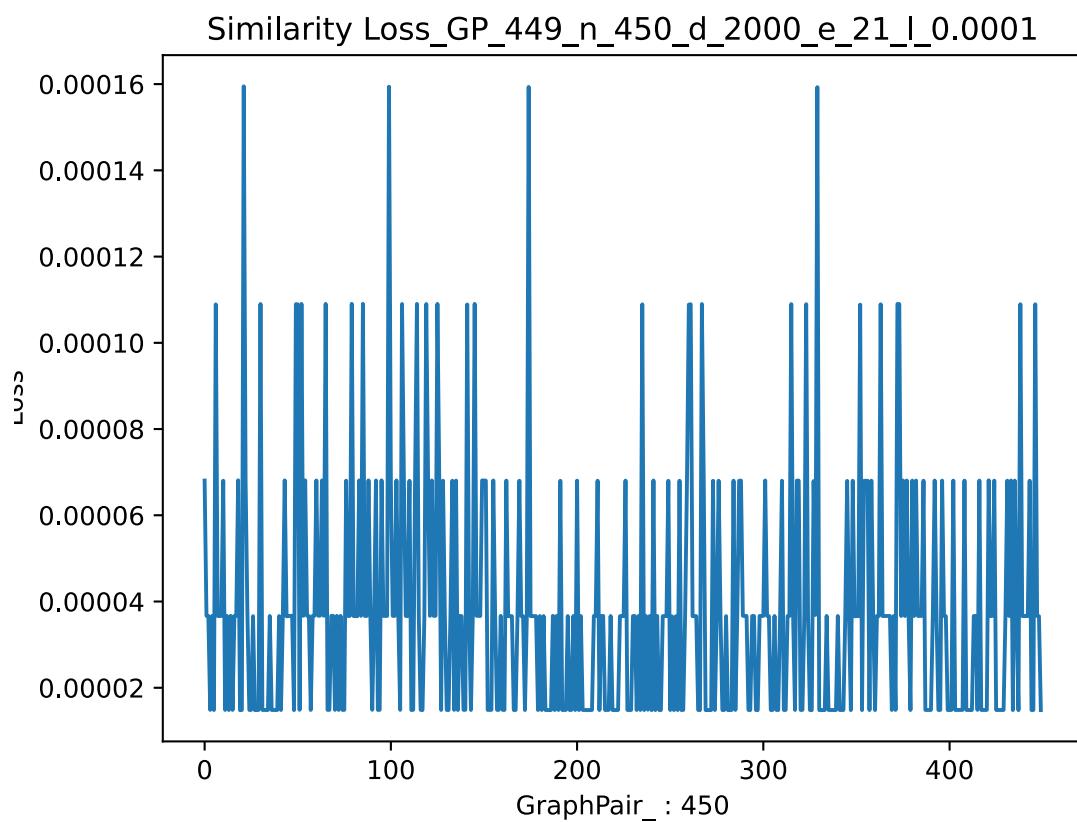
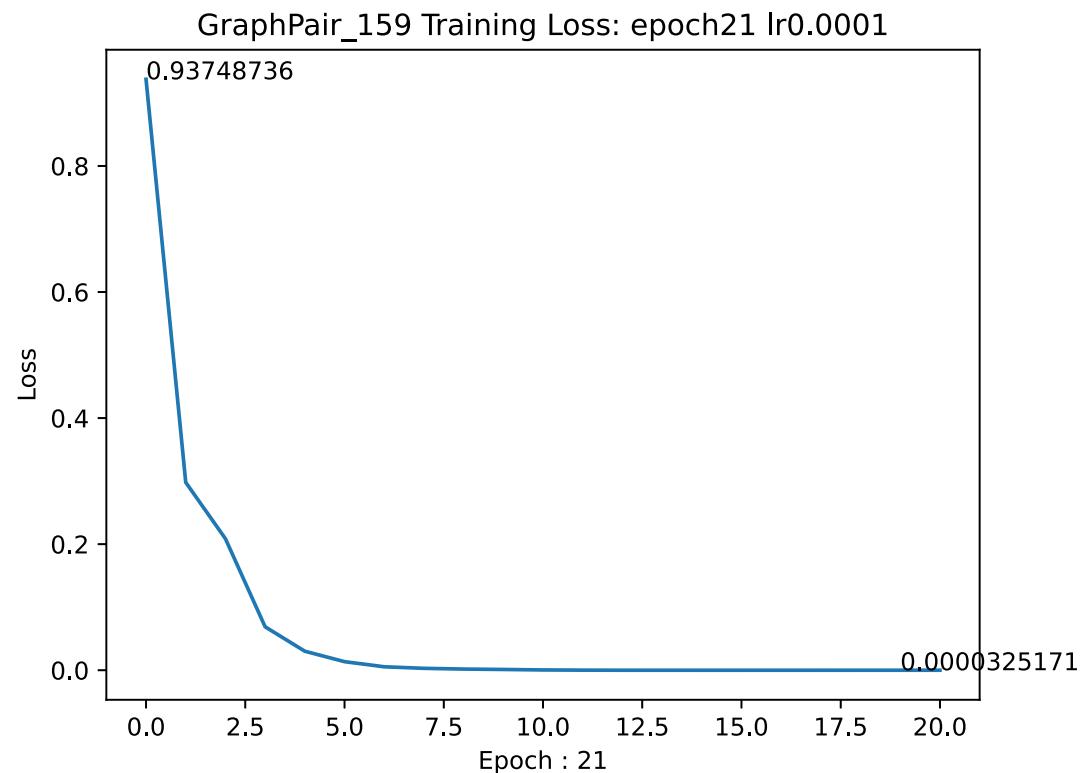
n450_d2000_Training_Loss_epoch_21_lr_0.0001

`prediction_loss`越低的点, 越适用于训练模型, (该点具有普适性, 越相同)

1. GP : 159 (lab)

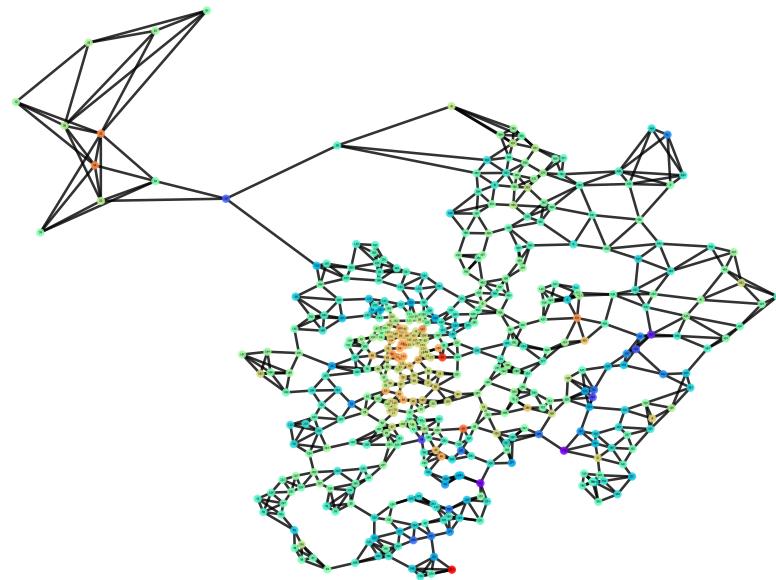


prediction_loss_1

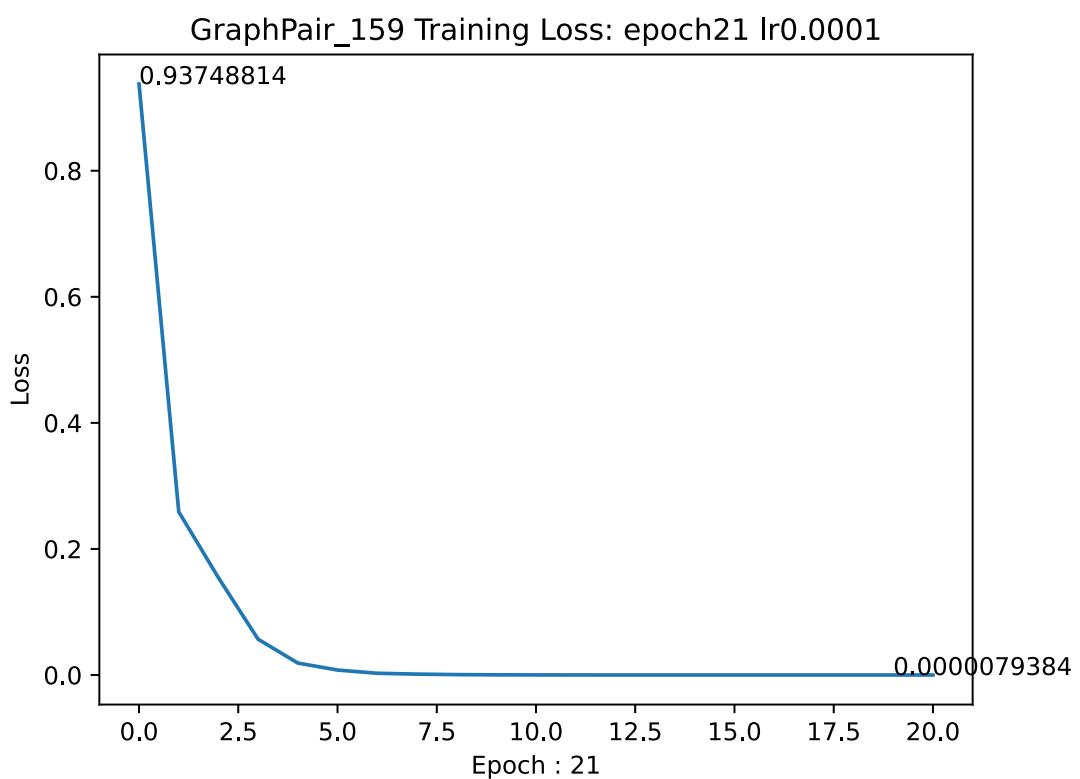


328异常: data[328] - 1.949527258053421974e-05 (借用75数据)

Graph Pair: 159



2. GP : 159 (508)

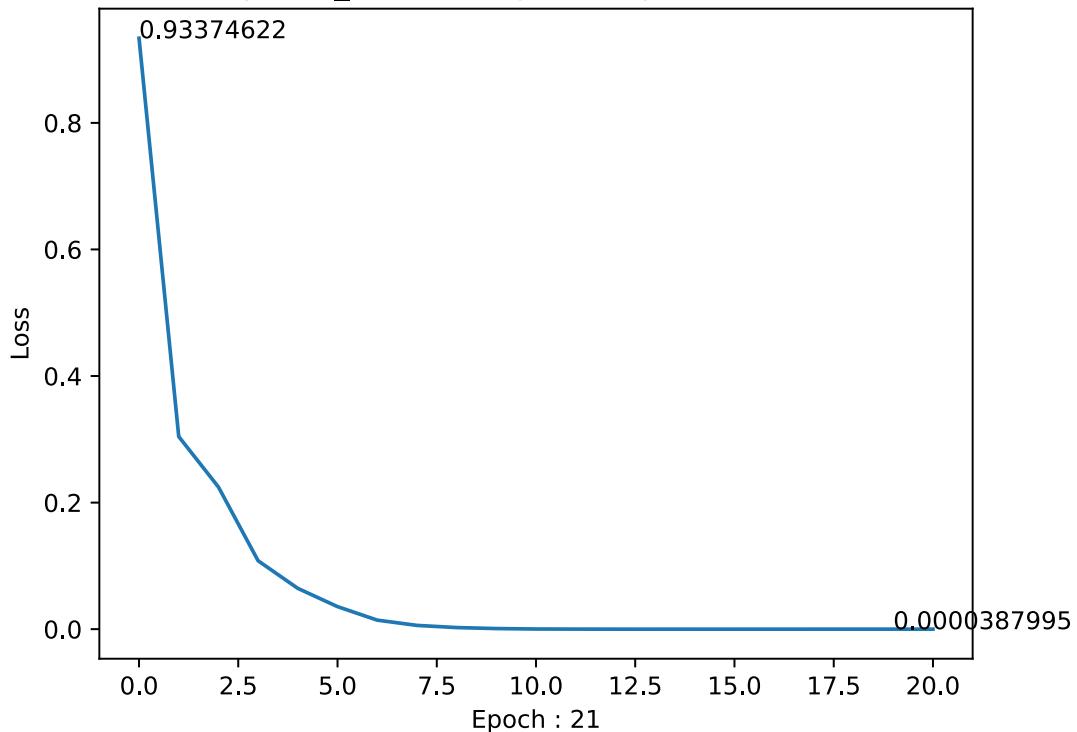


3. GP : 310 (shanshili)

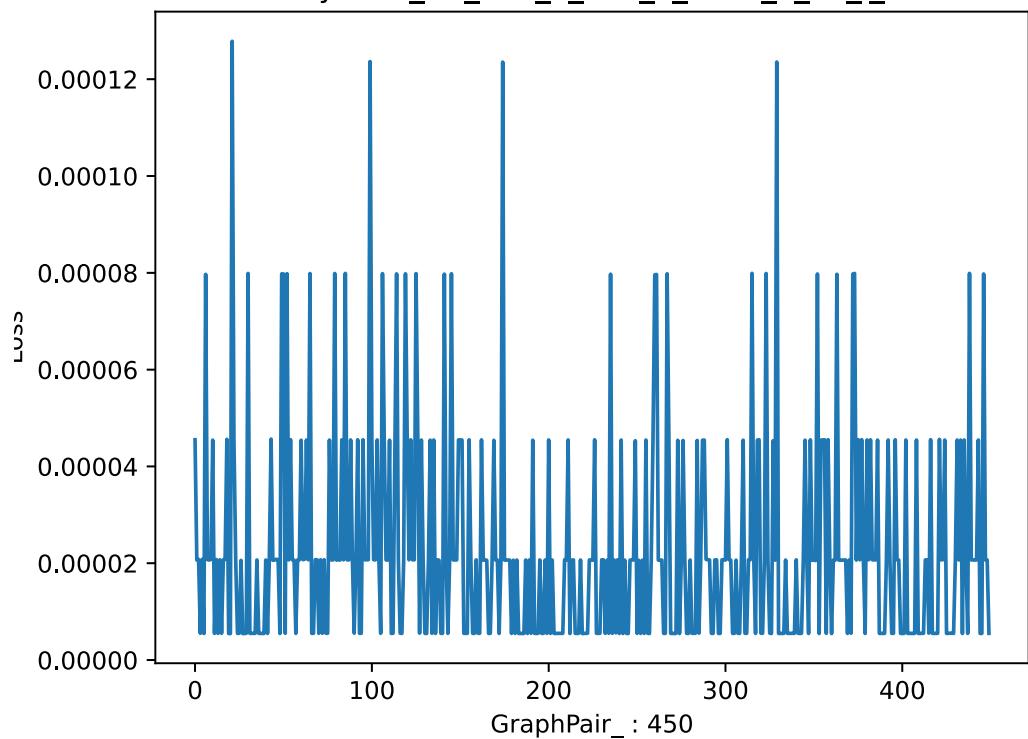


整体损失低一点

GraphPair_310 Training Loss: epoch21 lr0.0001

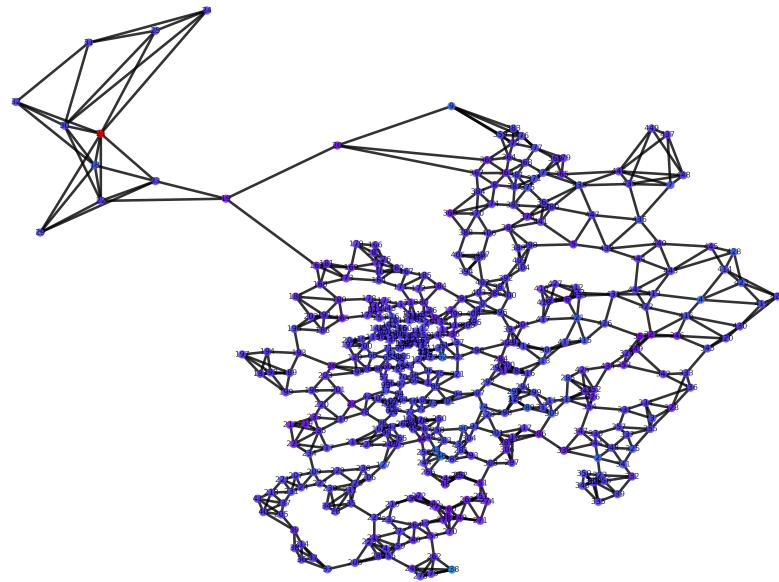


Similarity Loss_GP_449_n_450_d_2000_e_21_l_0.0001

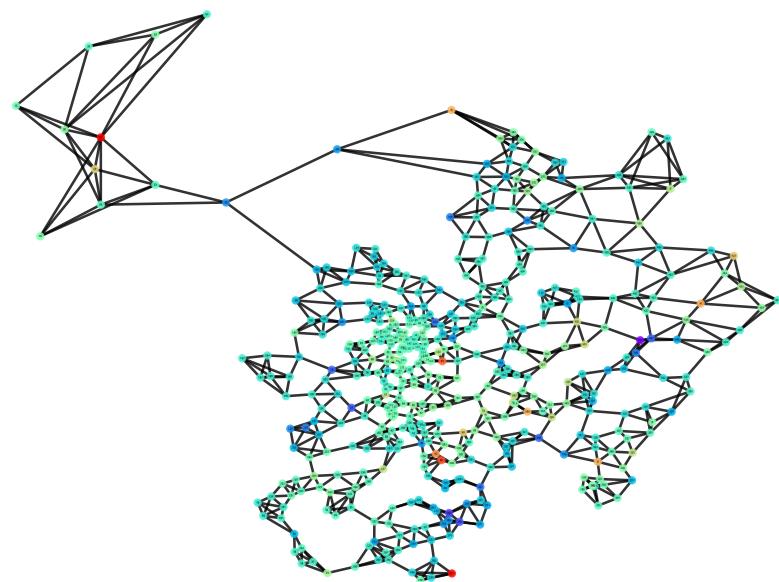


21异常 data[21] - 1.664528177166357636e-04 (调试数据)

Graph Pair: 1



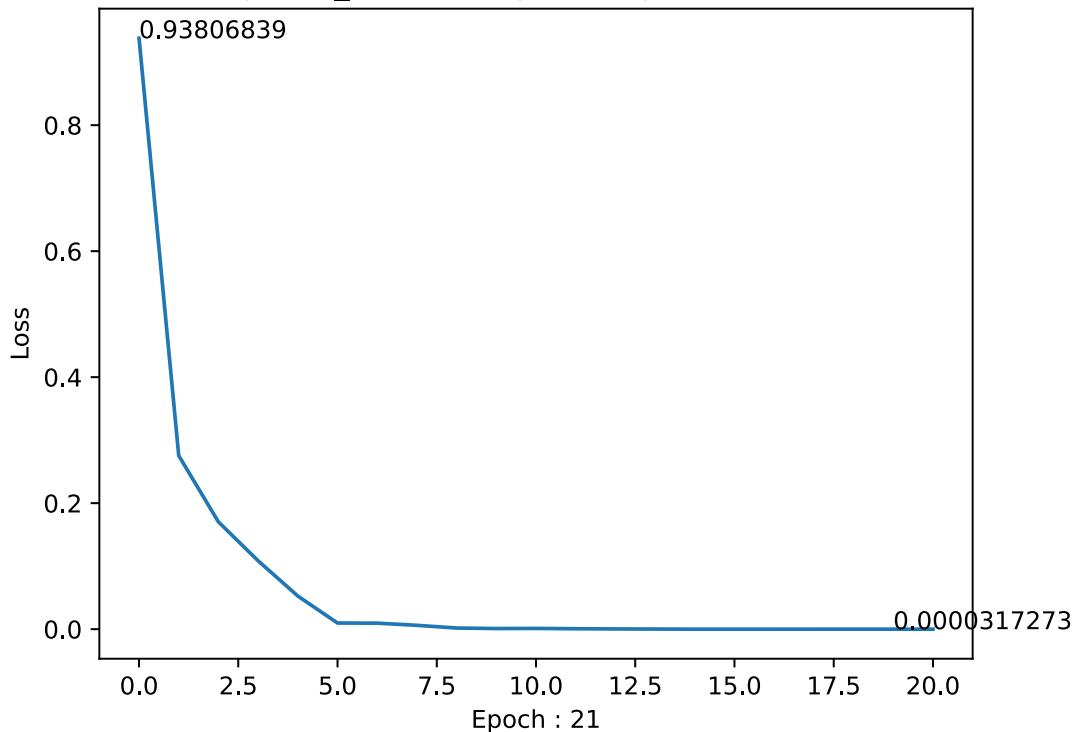
Graph Pair: 310



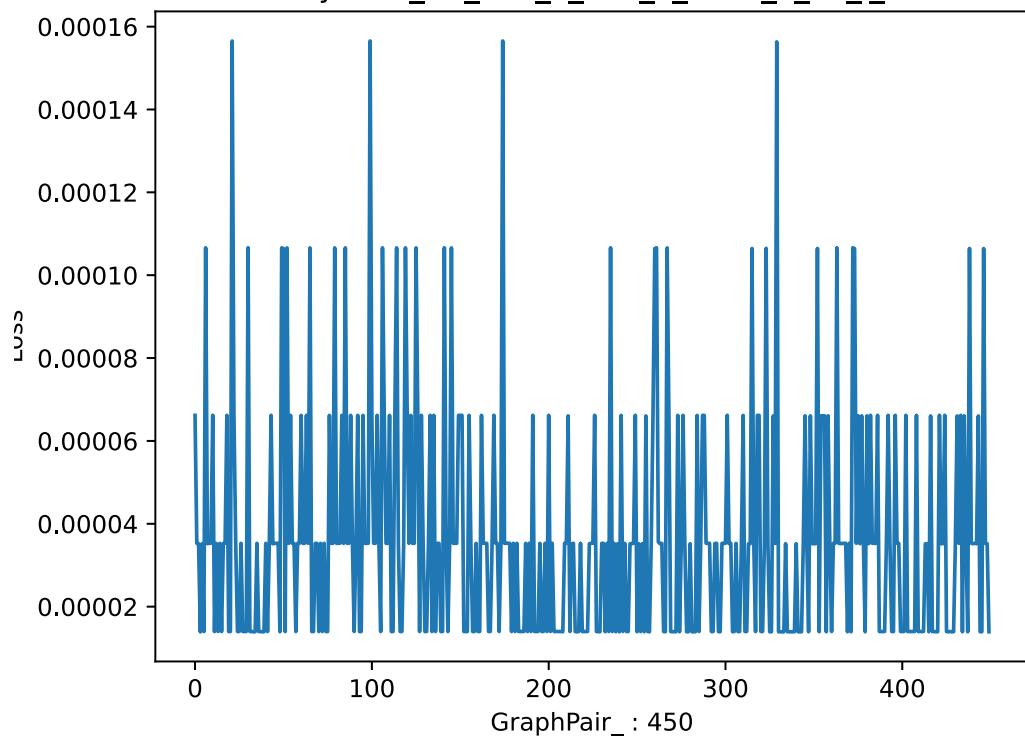
4. GP : 432 (508)

更平滑

GraphPair_432 Training Loss: epoch21 lr0.0001



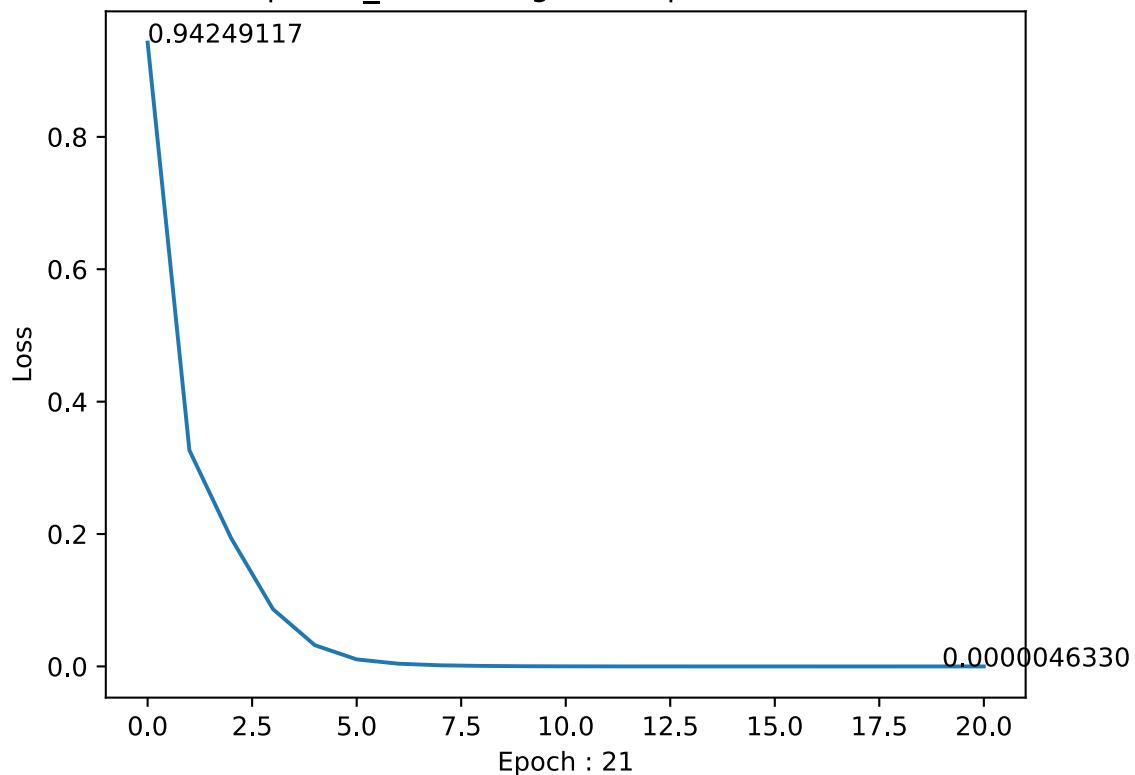
Similarity Loss_GP_449_n_450_d_2000_e_21_l_0.0001



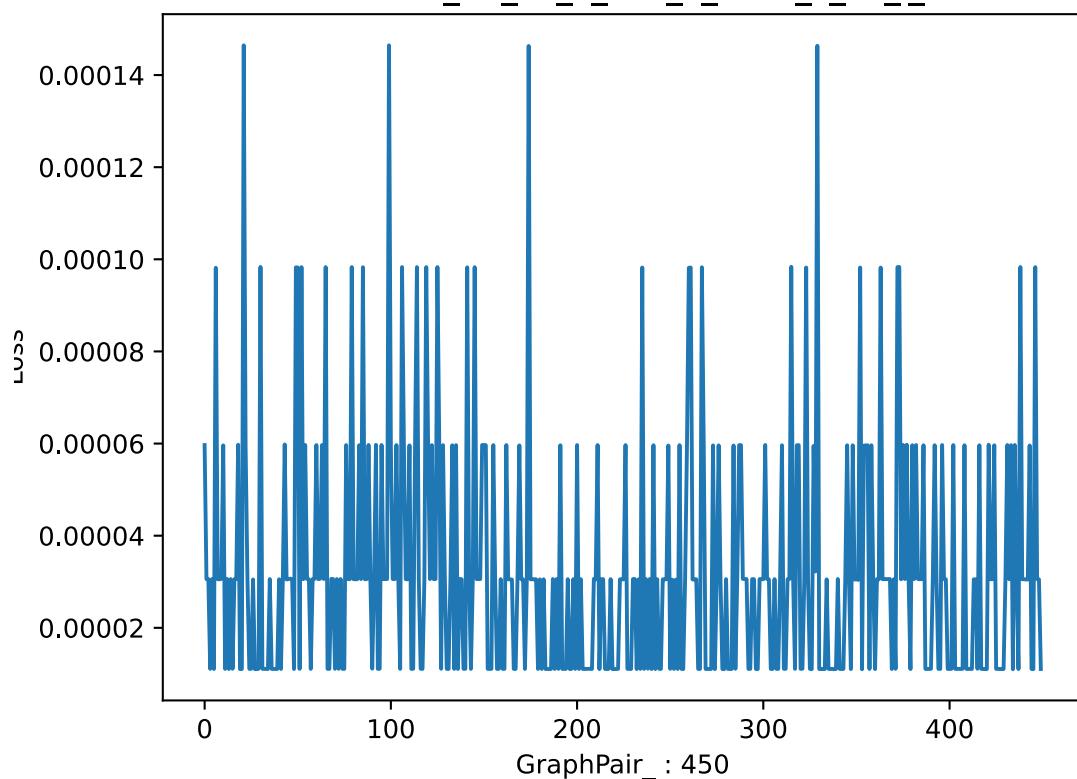
5. GP : 75 (shanshili)

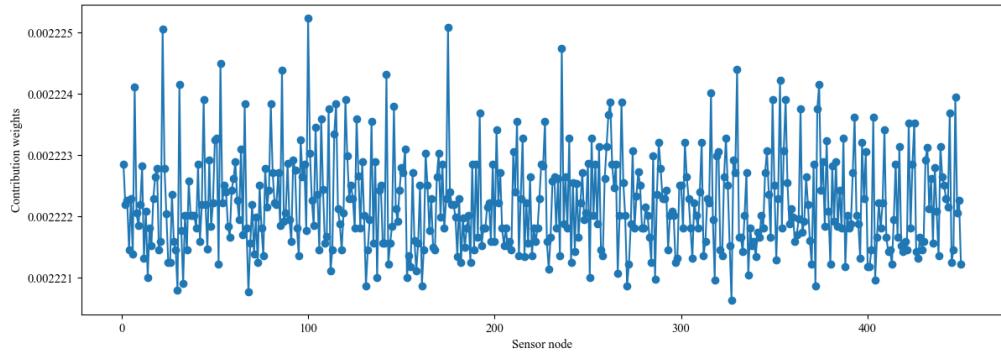


GraphPair_75 Training Loss: epoch21 lr0.0001



Prediction Loss_GP_75_n_450_d_2000_e_21_l_0.0001



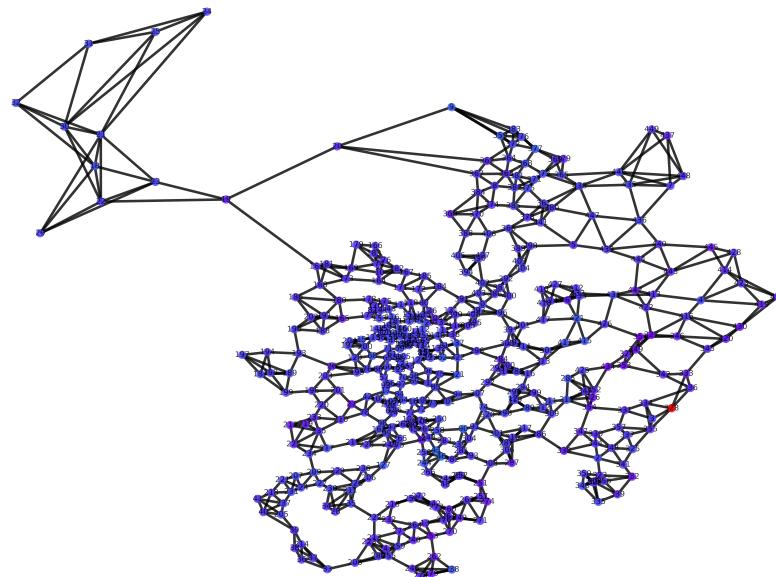


328异常 1.464528177166357636e-04 (收集数据)

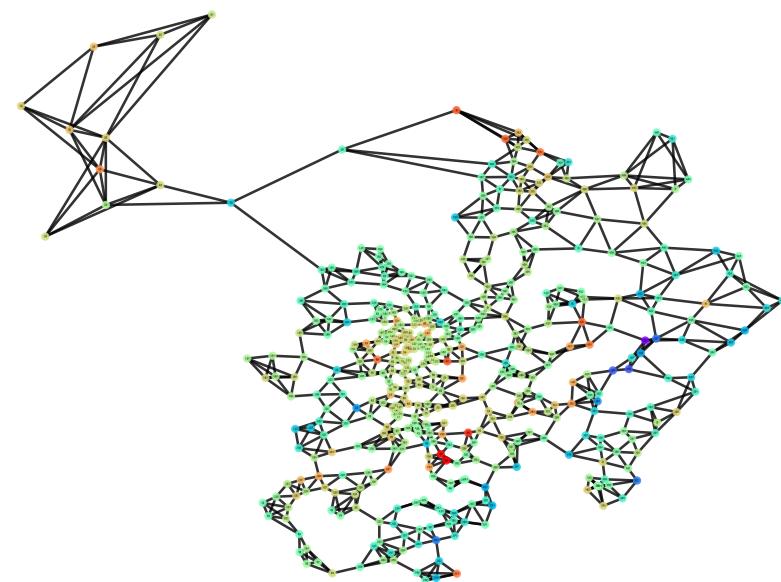
但是要用data[329]的数据修正data[328]

为什么异常点不同

Graph Pair: 2

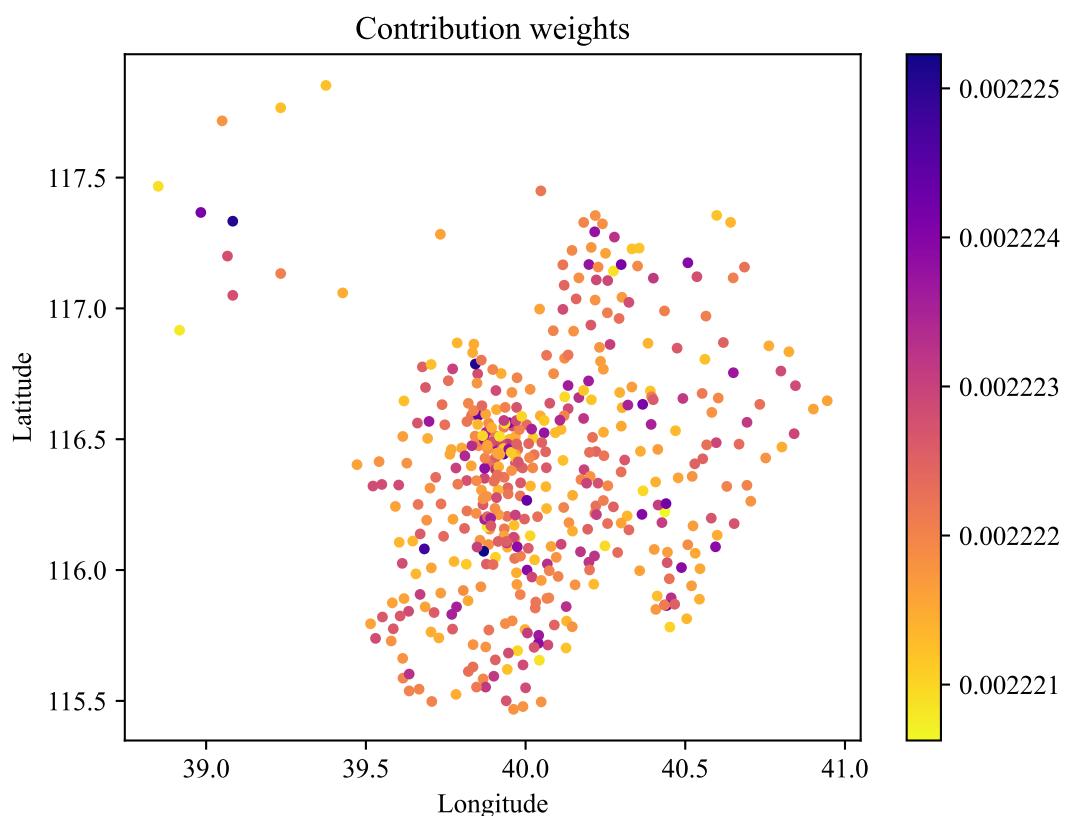


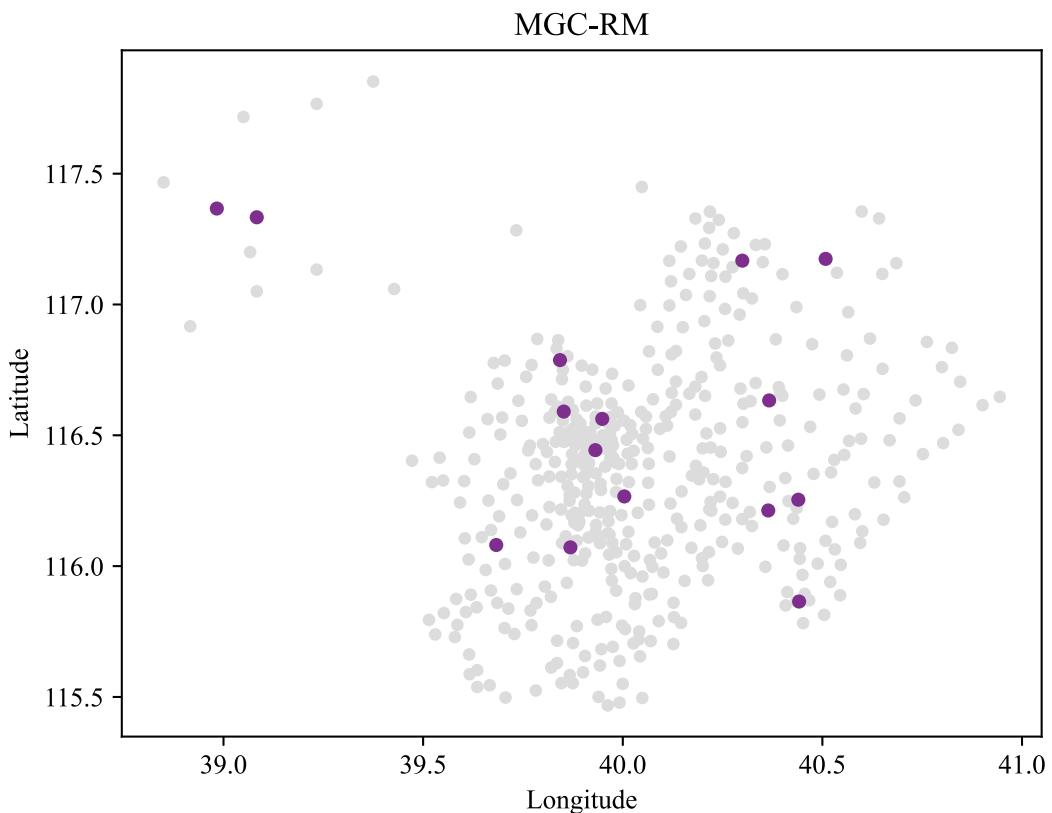
Graph Pair: 75



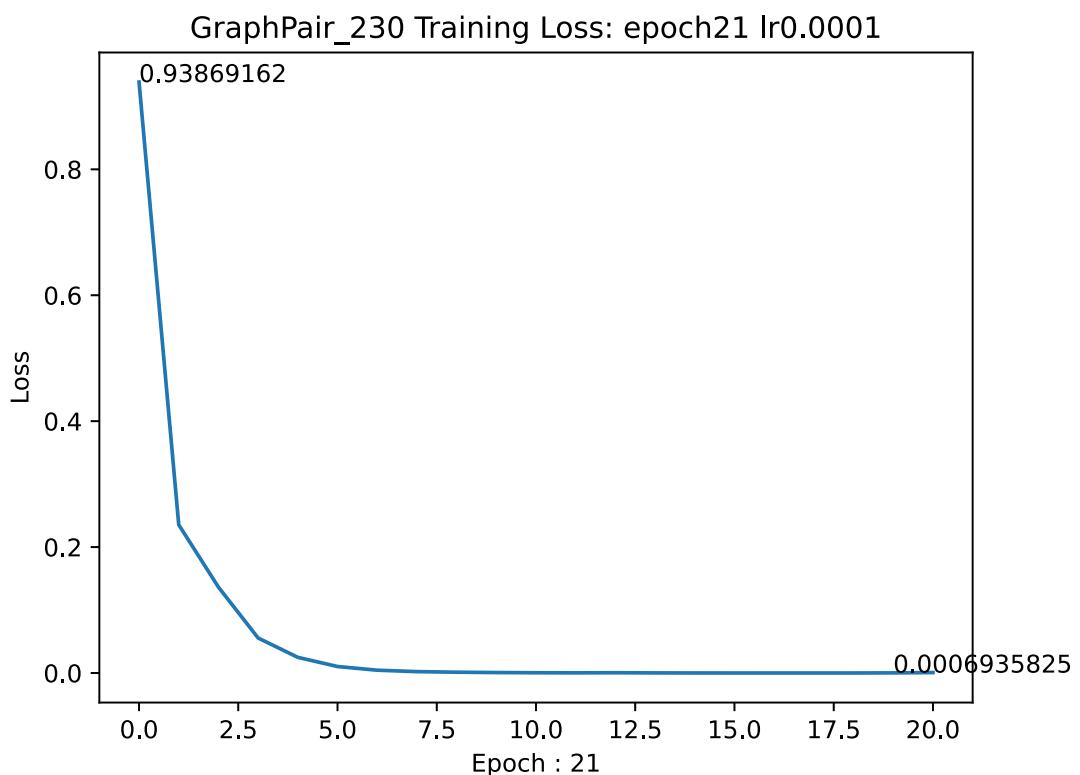
异常点不影响排序

Similarity Score—>(PageRank)—>Contribution weights





6. GP : 230 (lab)



7. GP : 100 (lab)

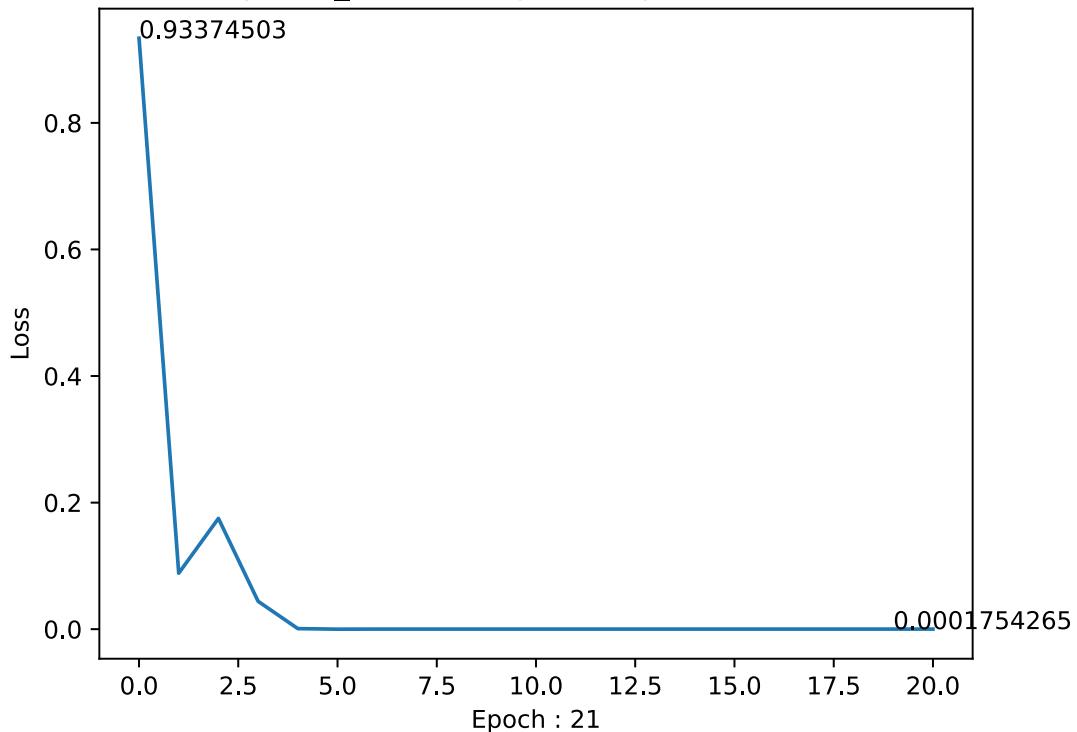
prediction loss 较大的点拿来训练似乎会出现尖刺。

但是每次会有不同。

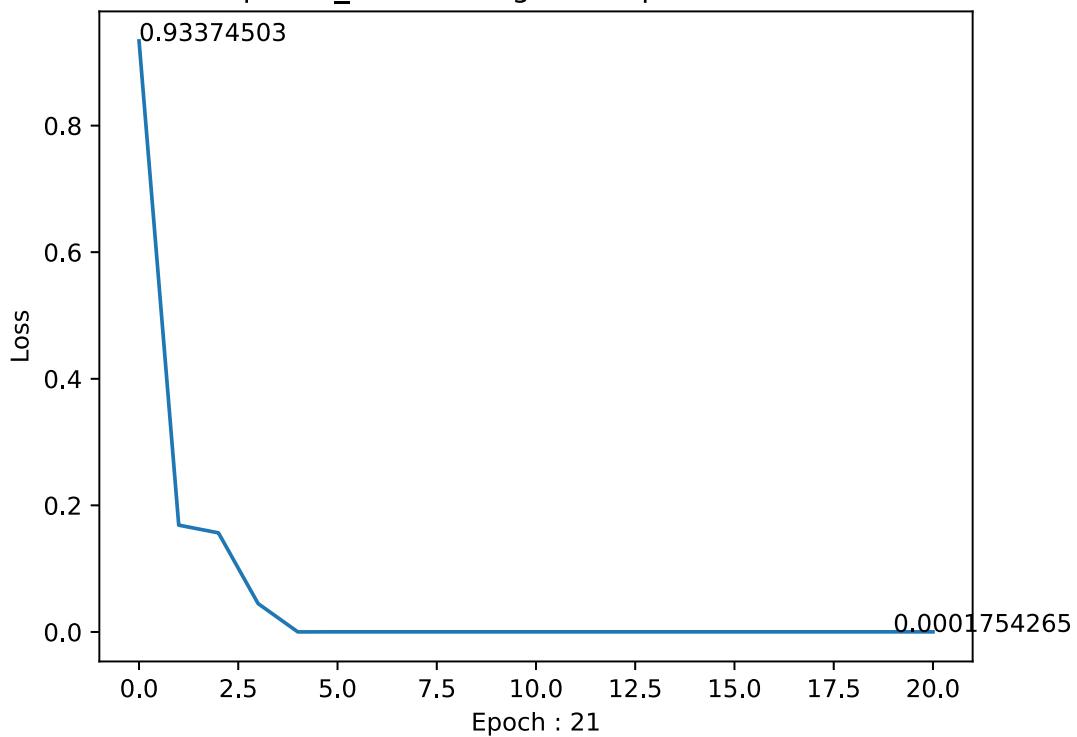
预测分数也总是1

GPU是不是需要单独设置种子,越跑越平滑了还

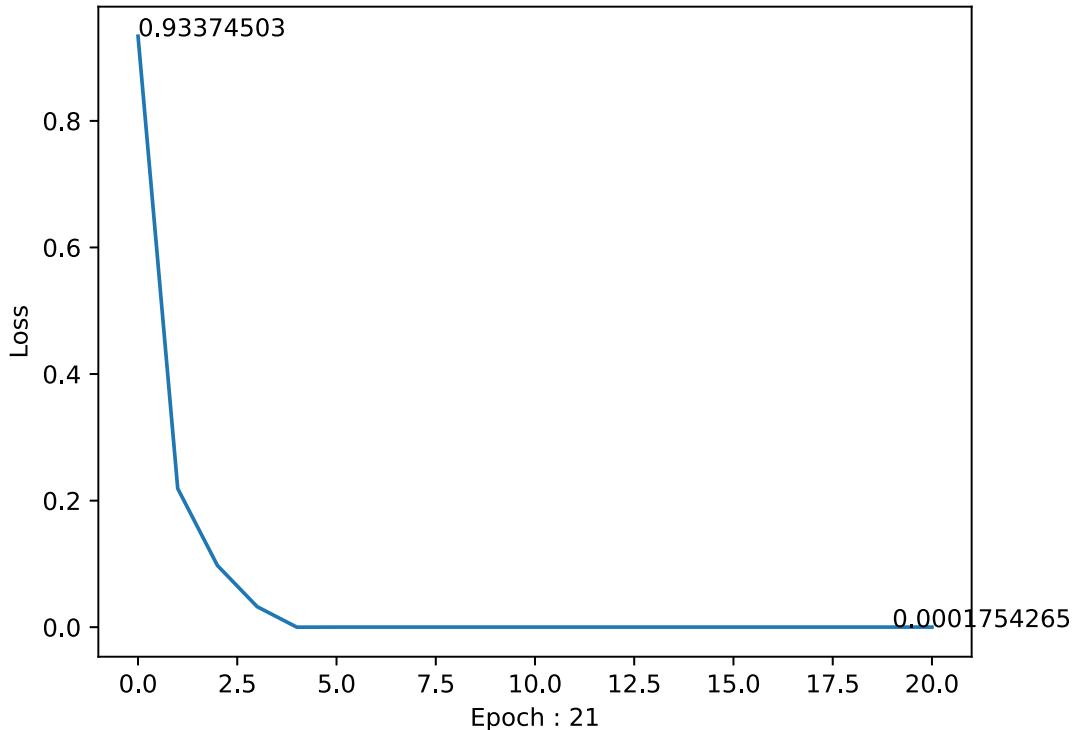
GraphPair_100 Training Loss: epoch21 lr0.0001



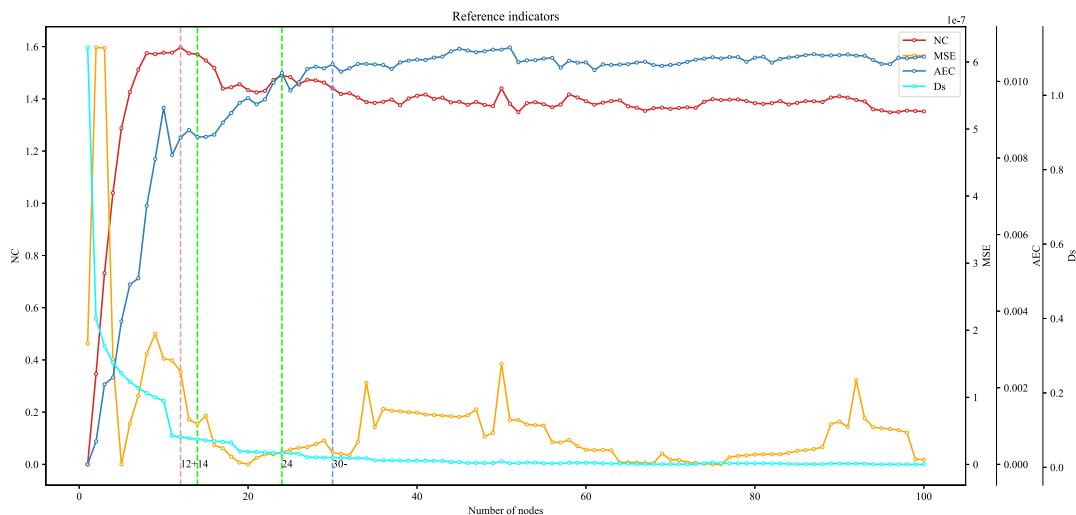
GraphPair_100 Training Loss: epoch21 lr0.0001



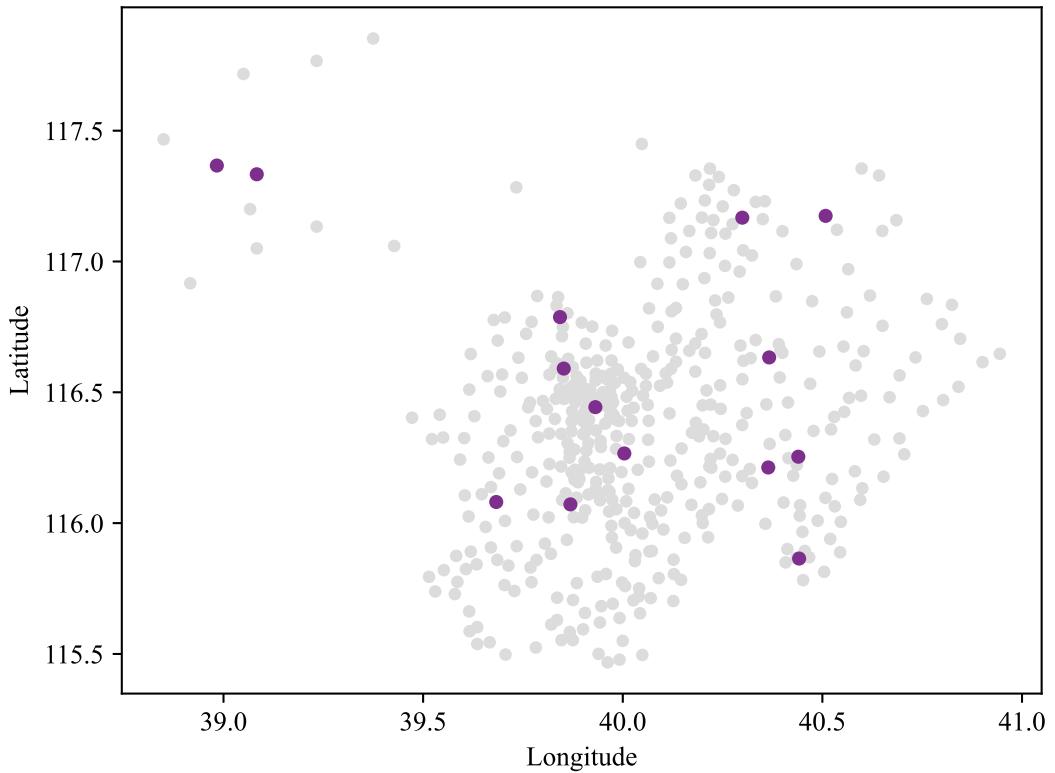
GraphPair_100 Training Loss: epoch21 lr0.0001



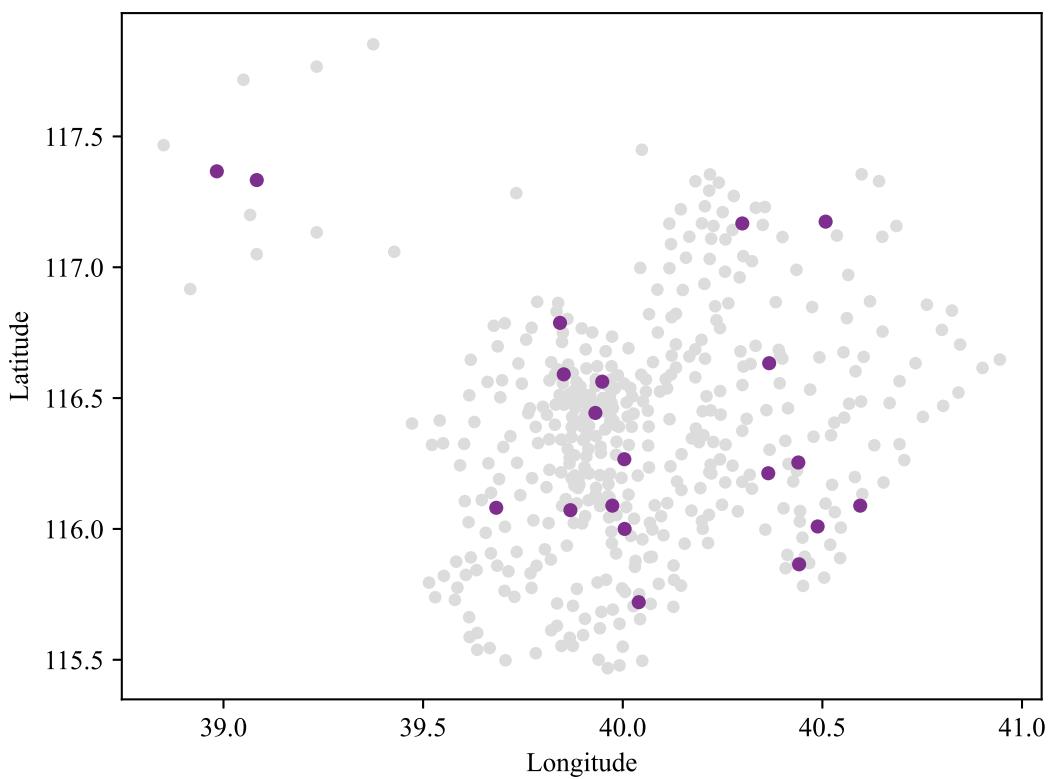
Reference indicators

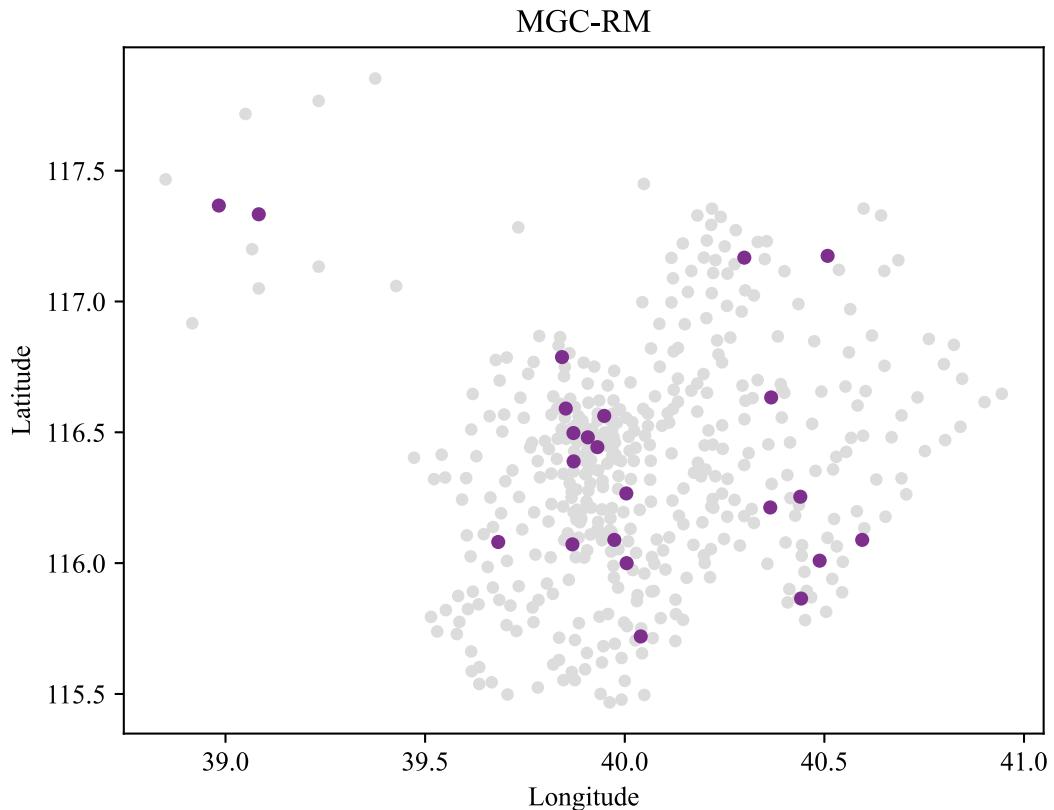


MGC-RM



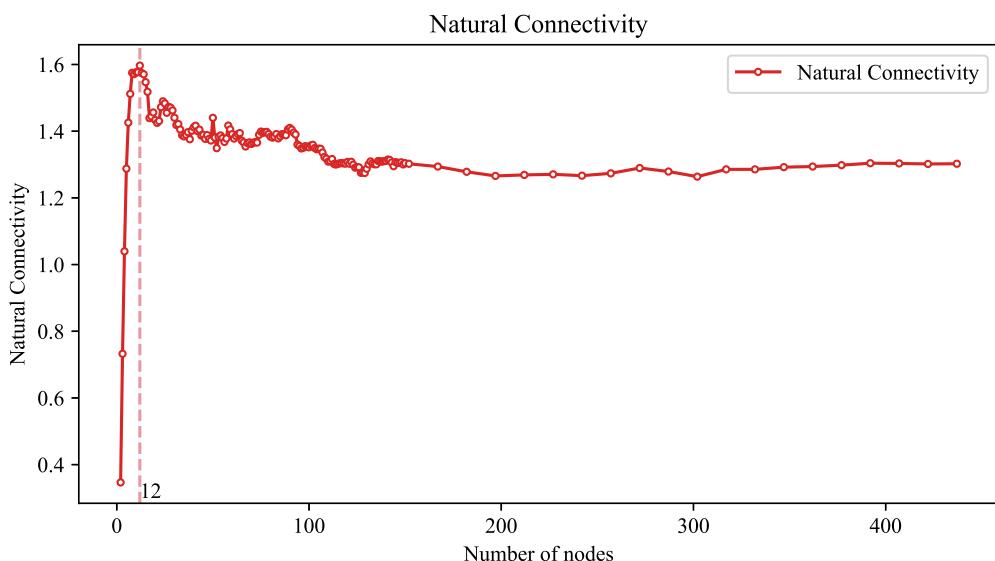
MGC-RM

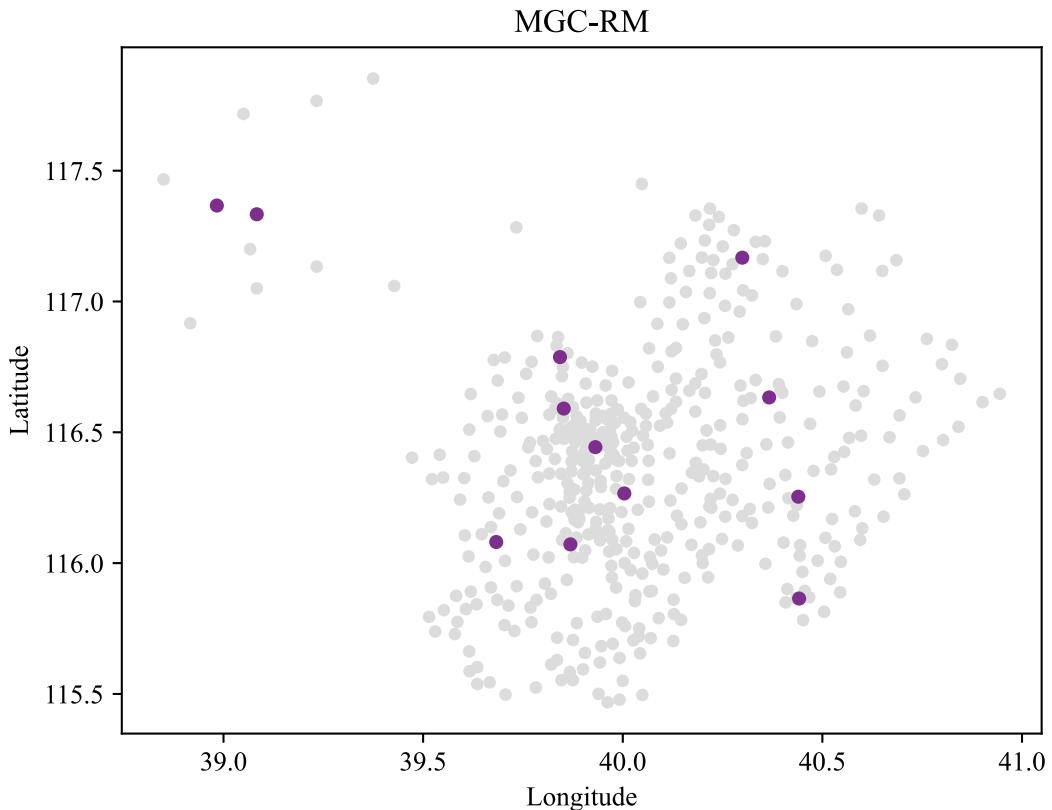




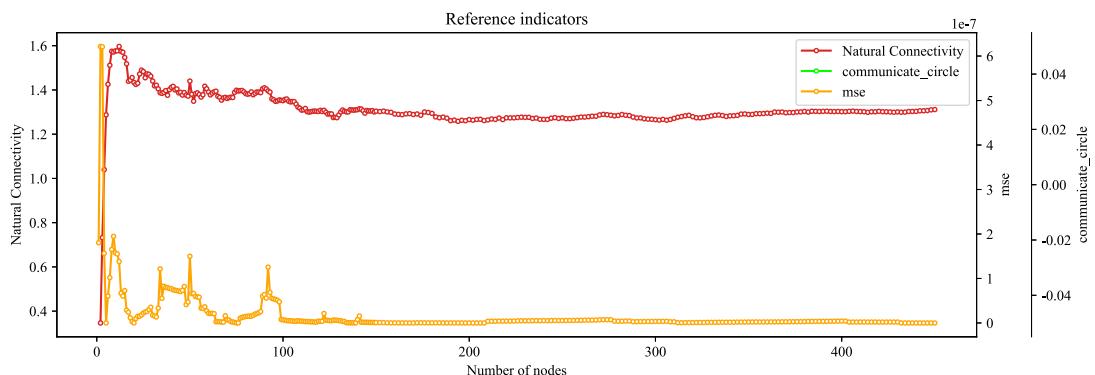
1. Natural Connectivity

selectrd_node = 12





2. MSE - single node



3. res_energy_avg communication_circle

公式：

网络初始能量： $E_{origin} = x \cdot N$ $x \in \mathbb{R}$ $N \in \mathbb{Z}^+$

网络半径：

如果是连通图：

$$r(G) = \min_{v \in V} \text{ecc}(v)$$

$$\text{ecc}(v) = \max_{u \in V} d(v, u)$$

如果非连通图，连通子图： C_1, C_2, \dots, C_k ， $C_t \subseteq V$

$$r(C_t) = \min_{m \in V} \text{ecc}(m)$$

$$\text{ecc}(m) = \max_{n \in V} d(m, n)$$

通信邻居 $V_{neighbor}$:

广度优先搜索 (BFS)

通信能量损失:

最短路径长度 $d(m, n) = \min_{P \in \mathcal{P}(m, n)} |P| , n \in V_{neighbor}$

$\mathcal{P}(m, n)$ 表示所有从节点 m 到节点 n 的路径集合

$E_i^j = E_i^{j-1} - (2 \cdot E_{elec} + \beta \cdot d(u, v)^\alpha) \cdot 10^{-9} \cdot \text{bit}, u \in V_{neighbor} , j \in (0, N)$

E_i^j 表示第 i 图第 j 轮通信的第 j 个节点, 即节点 v , 与范围 $d(u, v)$ 内的节点通信的能量消耗

$E_{loss} = [E_i^0, E_i^1, \dots, E_i^N]$

$E_{rest}^k = \{E_i^j \in E_{loss} \mid E_i^j > 0\} k \in \mathbb{Z}^+$

$$E_{average}^i = \frac{1}{N} \cdot \sum_{j=0}^K E_{rest}^k$$

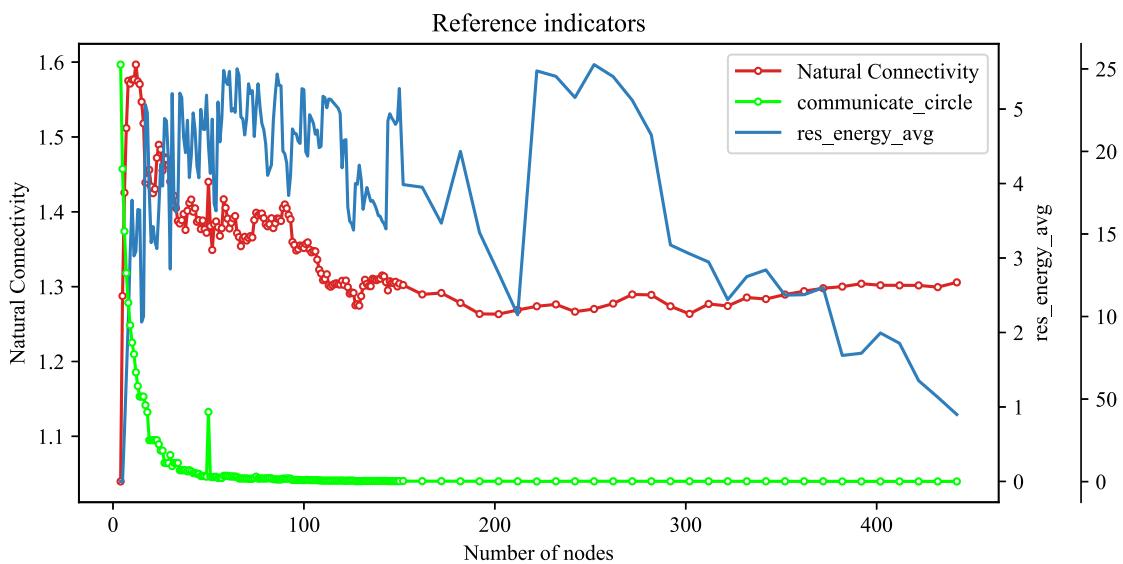
$$\exists \{E_i^j \in E_{loss} \mid E_i^j < 0\} \implies C_{network} = i$$

如果非连通图:

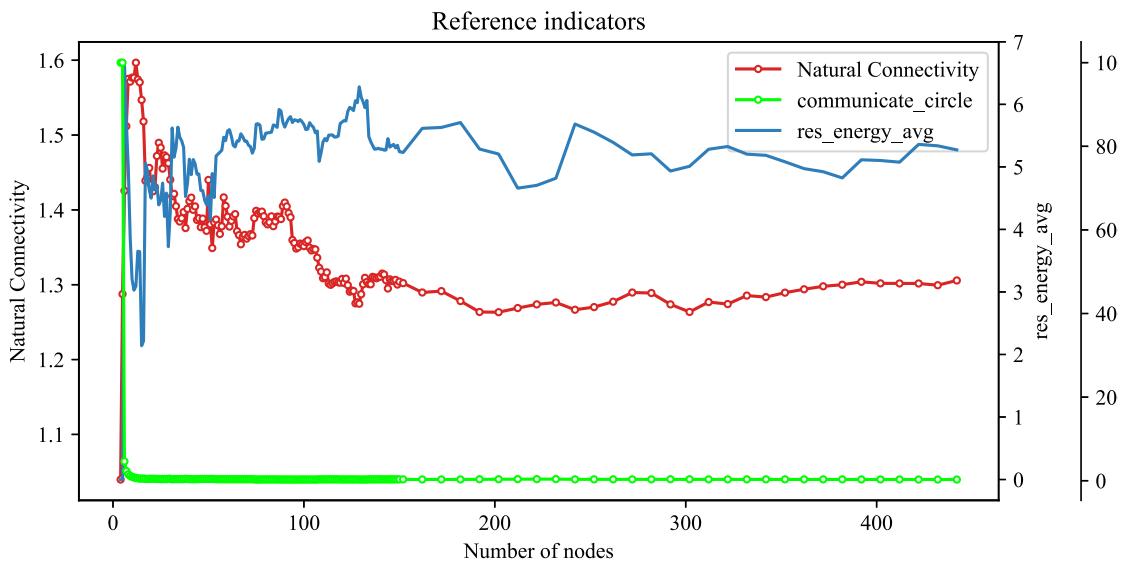
$$E_{average}^i$$

实验:

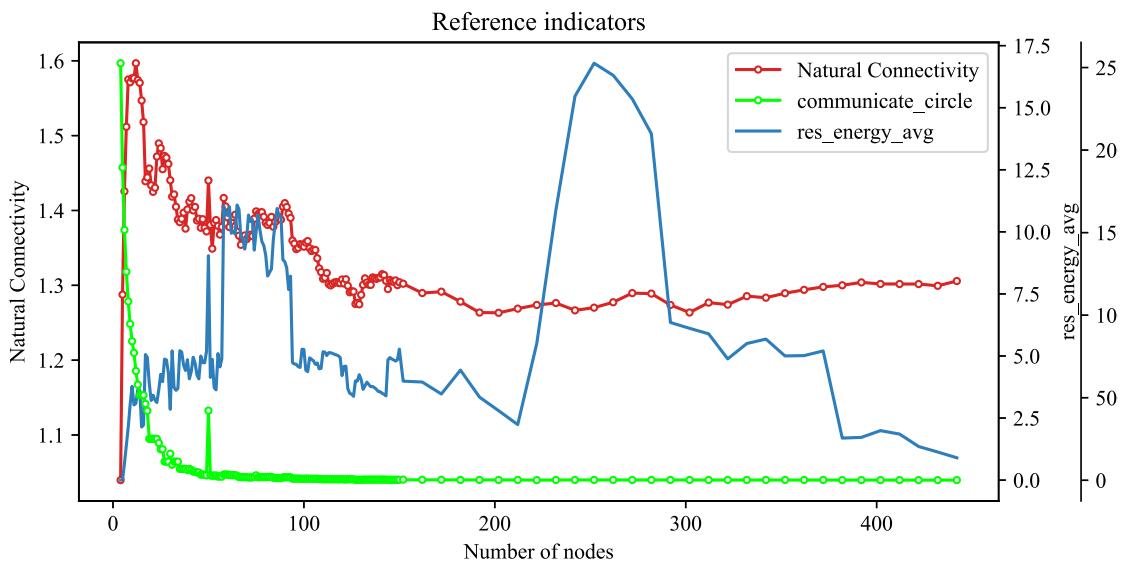
可以修改初始通信能量



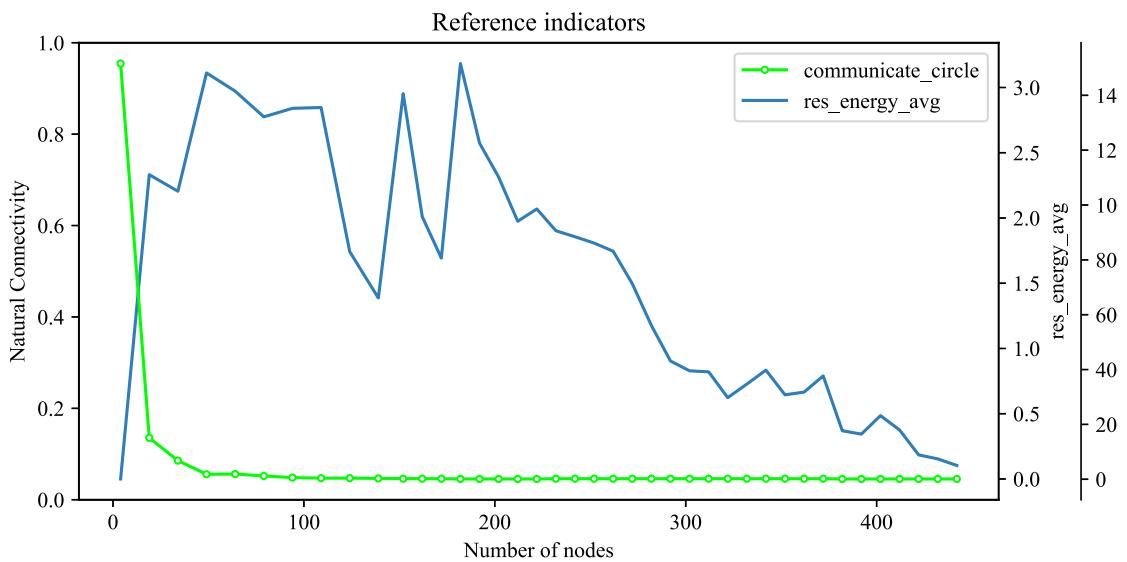
或者固定通信半径, 不计算图半径 (连接关系有争议)



平均剩余能量不求子连通图的平均

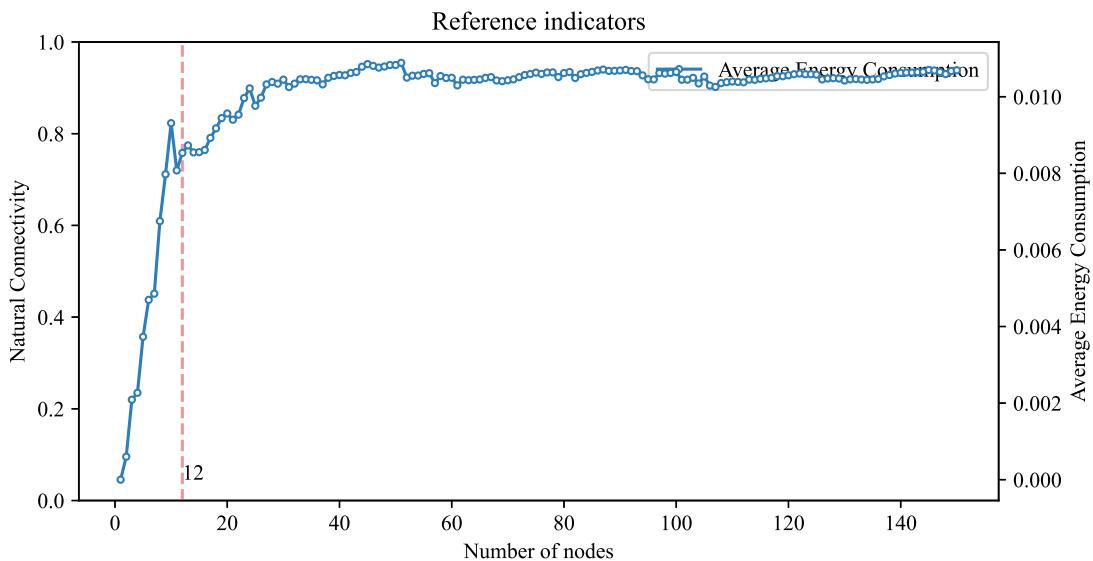
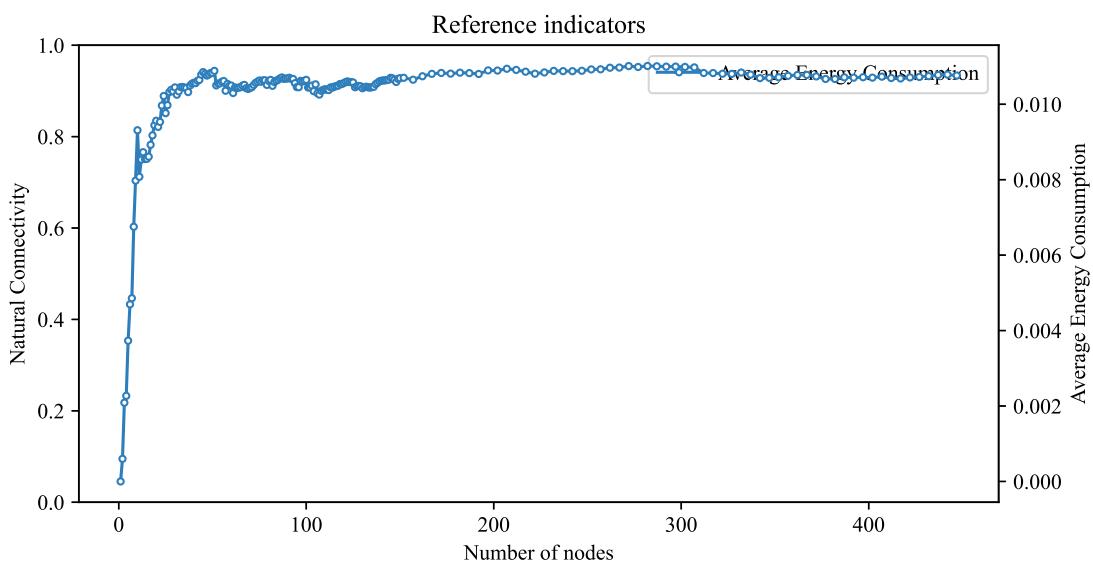
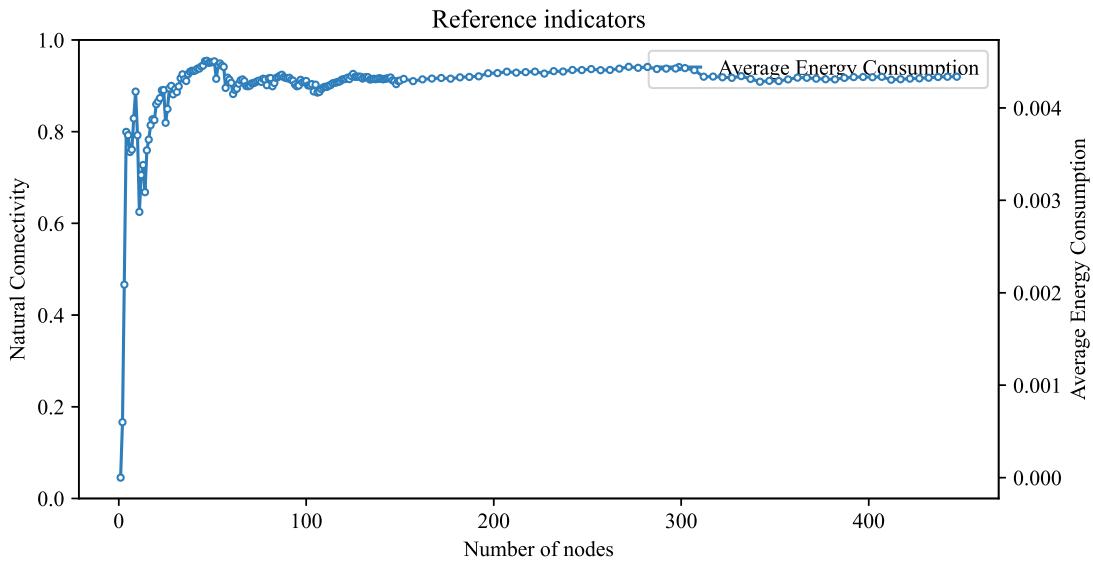


凑活画一个好看的图？？



4. ACE

实验：



5. DS

公式：

$$\gamma_{\text{value}} = \Gamma\left(\frac{m}{2} + 1\right)$$

$$R_{st} = \left(\frac{\gamma_{\text{value}}}{\pi^{m/2}} \cdot \frac{1}{n}\right)^{1/m}$$

$$Ds = \begin{cases} \frac{R_{st}}{r_{\max}} & \text{if } r_{\max} > 0 \\ \frac{R_{st}}{0.5} & \text{otherwise} \end{cases}$$

1. Method 1：不重新构图，偏心率计算空洞半径

连通子图： C_1, C_2, \dots, C_k ， $C_i \subseteq V$

$$\text{偏心率 } e_v^i = \max_{u \in V} d(v, u)$$

$$r_g^i = \min_{v \in V} e_v^i, i \in k$$

$$r_{\max} = \max(r_g^0, r_g^1, \dots, r_g^k)$$

2. Method 2：重新构图，

$\mathbf{P} \in \mathbb{R}_{n \times d}, \mathbf{D} \in \mathbb{R}_{n \times n}$

$$d(\mathbf{p}_i, \mathbf{p}_j) = \sqrt{\sum_{k=1}^d (p_{i,k} - p_{j,k})^2}, (i, j) \in n$$

$$\mathbf{D} = \begin{pmatrix} d(\mathbf{p}_1, \mathbf{p}_1) & d(\mathbf{p}_1, \mathbf{p}_2) & \dots & d(\mathbf{p}_1, \mathbf{p}_n) \\ d(\mathbf{p}_2, \mathbf{p}_1) & d(\mathbf{p}_2, \mathbf{p}_2) & \dots & d(\mathbf{p}_2, \mathbf{p}_n) \\ \dots & \dots & \dots & \dots \\ d(\mathbf{p}_n, \mathbf{p}_1) & d(\mathbf{p}_n, \mathbf{p}_2) & \dots & d(\mathbf{p}_n, \mathbf{p}_n) \end{pmatrix}$$

$$A_{ij} = \begin{cases} 1 & \text{if } d(\mathbf{p}_i, \mathbf{p}_j) < 1 \\ 0 & \text{otherwise} \end{cases}$$

3. Method 3：重新构图，考虑物理距离，考虑连接的邻居个数（这一过程与KNN类似，没有意义）

cKDTree :

$\mathbf{P} \in \mathbb{R}_{n \times d}, \mathbf{D} \in \mathbb{R}_{n \times n}$

$$d(\mathbf{p}_i, \mathbf{p}_j) = \sqrt{\sum_{k=1}^d (p_{i,k} - p_{j,k})^2}, (i, j) \in n$$

查询点 \mathbf{q} ，找到最近的点 \mathbf{p}_i : $\mathbf{p}_{\text{nearest}} = \arg \min_{\mathbf{p}_i \in \{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_n\}} d(\mathbf{q}, \mathbf{p}_i)$

$$A_{ij} = \begin{cases} 1 & \text{if } j \in \mathbf{p}_{\text{nearest}} \\ 0 & \text{otherwise} \end{cases}$$

连通子图： C_1, C_2, \dots, C_k ， $C_i \subseteq V$,

$$(v_m, v_n) \in C_i$$

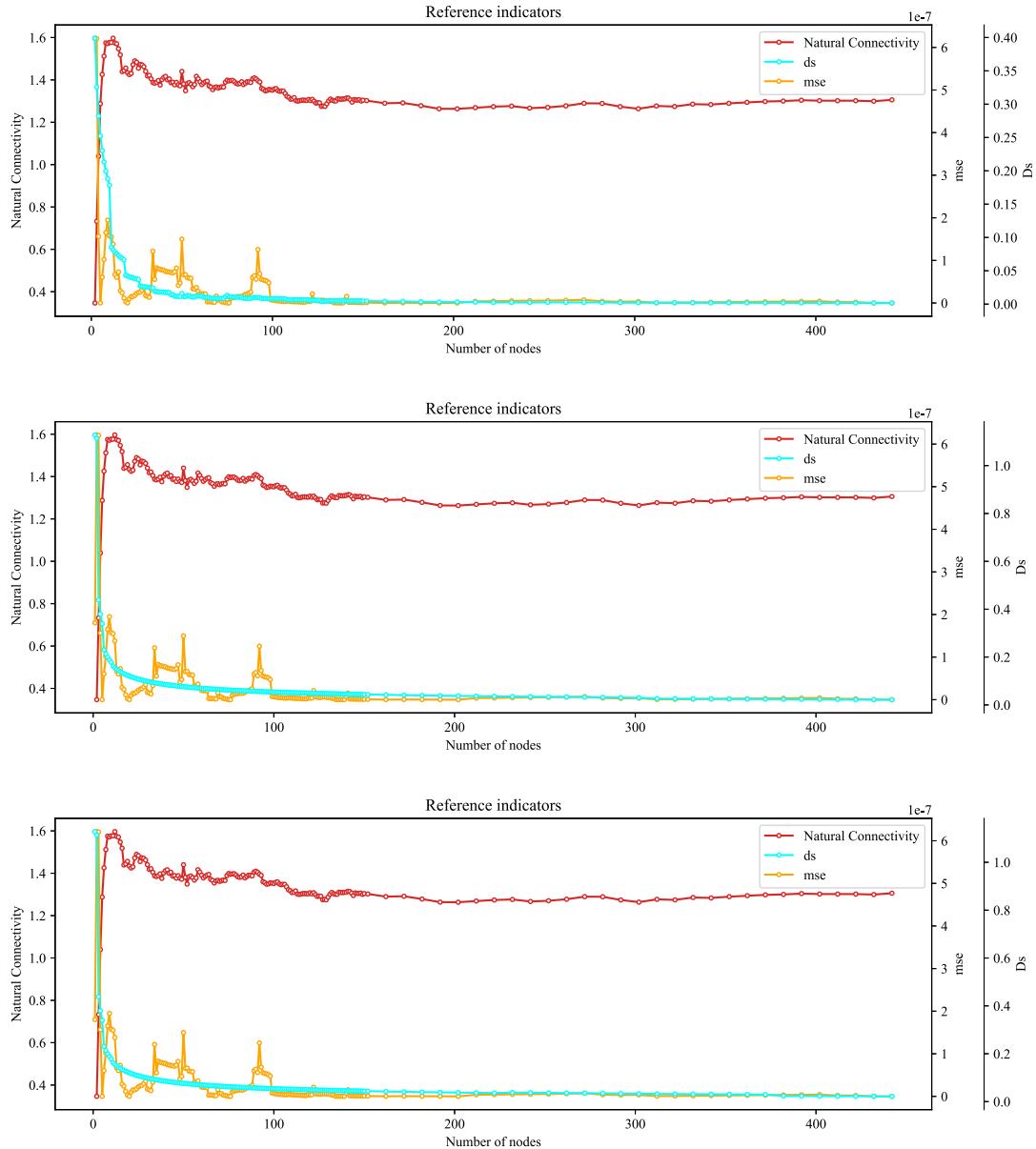
$$d(\mathbf{p}_m, \mathbf{p}_n) = \sqrt{\sum_{k=1}^d (p_{m,k} - p_{n,k})^2}, (m, n) \in |V(C)|$$

$$r_{c_i} = \frac{1}{2} \cdot \max_{(m, n) \in |V(C)|} d(\mathbf{p}_m, \mathbf{p}_n)$$

$$r_{max} = \max_{c_i \in [C_1, C_2, \dots, C_k]} r_{c_i}$$

实验：

重新构图了



趋势类似

Ds越小，

LOG

241120

1. eval效果很差

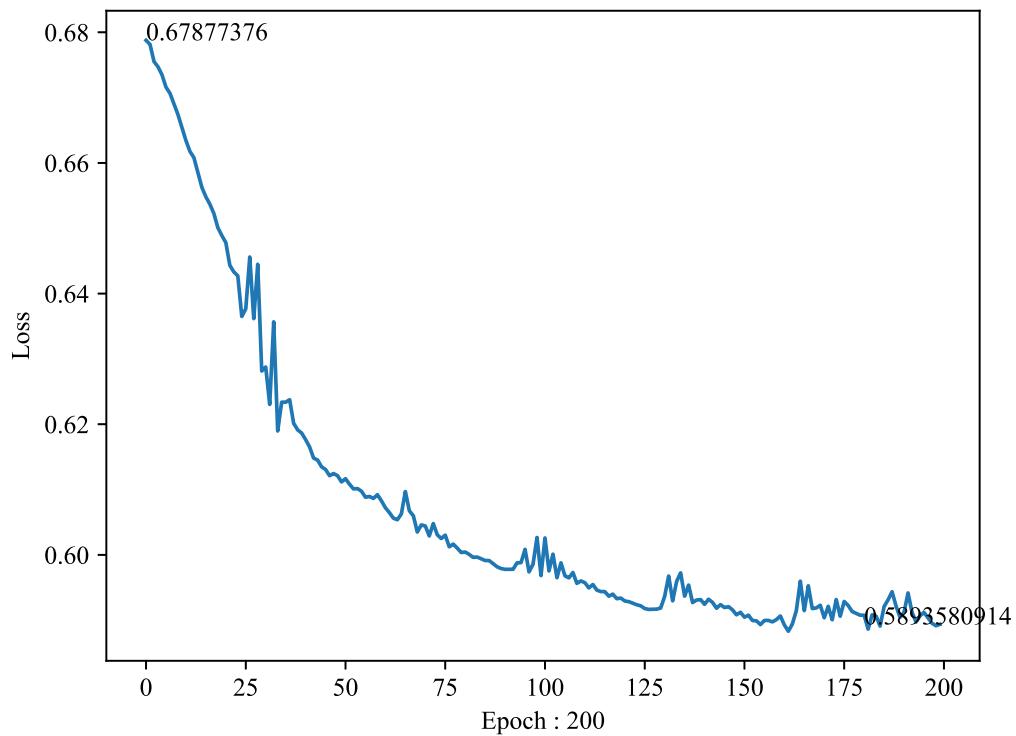
241119 梯度中断

1. cuda跑起来了，因为多个节点数据拼接过程中用的list导致梯度中断，所以之前没有梯度

修正后打印参数梯度依然为none，但是loss有在下降，应该是没有选对参数

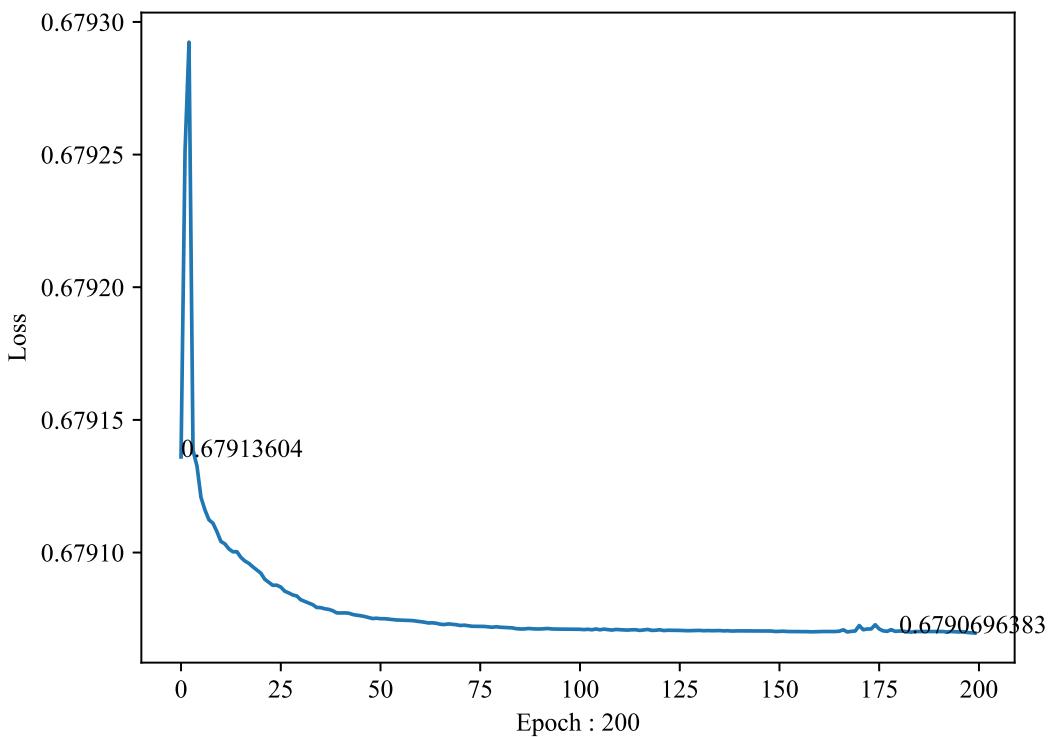
但是cuda的训练并没有明显的提速，依然很慢

2. 直接用R_g训练的损失好像不正常，不如用R_g的排序训练



那是不是也应该用scores的排序训练？？

梯度会消失



损失函数在训练过程中变得很大并且不再下降

- 当遇到损失函数不下降的情况，并且怀疑`loss.backward()`没有起作用时，可以考虑以下几个方面来排查和解决问题：

1. 损失函数定义

- 数值稳定性：在你的代码中使用了`sigmoid`函数，这可能导致梯度消失问题。特别是在计算`log(F.sigmoid(y_hat_ij))`时，如果`y_hat_ij`非常大或非常小，`F.sigmoid(y_hat_ij)`可能接近于1或0，导致`log`的结果为负无穷或非常大的负数，从而影响梯度的计算。
- 除以组合数：在返回损失值之前，你将总损失除以组合数`comb(len(scores), 2)`。这一步是合理的，但如果组合数非常大，可能会导致损失值变得非常小，进而影响梯度的大小。

2. 梯度检查

- 梯度是否为零：可以通过打印`scores_tensor.grad`来检查梯度是否为零。如果梯度始终为零，那么`loss.backward()`确实没有产生有效的梯度更新。
- 模型参数梯度：同样地，也可以检查模型参数的梯度，确保它们不是零或者非常小的值。

3. 优化器设置

- 学习率：过高的学习率可能导致模型权重的更新过大，使得模型无法收敛；过低的学习率则可能导致模型训练非常缓慢，甚至看起来像是没有变化。尝试调整学习率，看看是否有改善。
- 优化器类型：不同的优化器（如SGD、Adam等）对不同任务的表现可能有所不同。你可以尝试更换其他类型的优化器。

4. 模型架构

- 模型复杂度：模型过于简单可能无法捕捉到数据中的复杂模式，而过于复杂的模型又可能导致过拟合。确保模型的复杂度适合你的任务。
- 初始化：模型权重的初始化方式也会影响训练过程。确保你使用了一种合适的初始化方法。

5. 其他调试技巧

- 简化模型：尝试用一个更简单的模型来测试同样的数据和损失函数，看是否能正常工作。这有助于确定问题是在数据处理、损失函数还是模型本身。
- 单步调试：逐步执行代码，观察每一步的输出，特别是损失值的变化，以及梯度的计算。

如果你已经尝试了上述建议但仍然存在问题，可能需要进一步检查具体的实现细节，例如确保所有需要计算梯度的操作都在`autograd`机制下进行，确保张量的`requires_grad`属性被正确设置等。

- 这个警告信息表明你在访问一个非叶子节点（non-leaf Tensor）的`.grad`属性。在PyTorch中，只有叶子节点（leaf Tensor）才会默认保存梯度。如果你需要访问非叶子节点的梯度，可以使用`retain_grad()`方法。

具体来说，`scores_tensor`是通过模型的前向传播计算得到的，因此它是一个非叶子节点。为了能够访问它的梯度，你需要在调用`loss.backward()`之前调用`retain_grad()`方法。

以下是修改后的代码示例：

```
scores_tensor = torch.Tensor(scores).requires_grad_(True).to(device)
# 保留非叶子节点的梯度
scores_tensor,retain_grad()
```

解释

1. `scores_tensor,retain_grad()`: 这行代码确保 `scores_tensor` 的梯度会被保存，即使它不是一个叶子节点。
2. `loss.backward()`: 这行代码计算损失的梯度并将其存储在相应的张量的 `.grad` 属性中。

进一步调试

如果问题仍然存在，可以尝试以下步骤：

- 打印梯度：在 `loss.backward()` 之后，打印 `scores_tensor.grad` 和模型参数的梯度，确保它们不为零。
 - 检查损失值：确保损失值在每次迭代后有所变化，而不是一直保持不变。
 - 简化模型：尝试用一个更简单的模型来验证损失函数和优化器是否正常工作。
3. 如果 `scores_tensor,retain_grad()` 之后，`scores_tensor.grad` 仍然为 `None`，这通常意味着在反向传播过程中，`scores_tensor` 没有接收到任何梯度。以下是一些可能的原因和解决方法：

1. 确保 `scores_tensor` 是计算图的一部分

确保 `scores_tensor` 是由模型的前向传播计算得到的，并且在计算图中是可导的。如果 `scores_tensor` 是从某个常量或独立的张量派生出来的，它可能不会参与到梯度计算中。

2. 检查损失函数的计算

确保损失函数的计算没有断开计算图。例如，如果在损失函数中使用了某些操作，这些操作可能会导致梯度计算中断。常见的问题包括使用了 `.detach()` 或者 `.item()` 等方法。

3. 检查 `optimizer.zero_grad()`

确保 `optimizer.zero_grad()` 被正确调用，否则之前的梯度可能会累积，导致新的梯度无法正确计算。

4. 打印中间变量

在损失计算过程中，打印一些中间变量，确保它们的值和梯度都是正确的。

5. 确保 `requires_grad` 设置正确

确保所有需要计算梯度的张量都设置了 `requires_grad=True`。

6. 使用 `torch.autograd.grad` 进行调试

使用 `torch.autograd.grad` 可以手动计算梯度，帮助你定位问题。

示例代码

以下是一个更详细的示例，包含了上述建议的调试步骤：

```
# 检查模型参数的梯度
for name, param in ILGR_model.named_parameters():
    if param.grad is None:
        print(f"{name}.grad is None")
    else:
        print(f"{name}.grad:", param.grad)
```

进一步调试

如果上述步骤仍然不能解决问题，可以尝试以下方法：

1. **简化模型**：使用一个非常简单的模型（例如线性回归）来测试相同的损失函数和优化器，确保它们能正常工作。
2. **手动计算梯度**：使用 `torch.autograd.grad` 手动计算梯度，确保梯度计算是正确的。

```
# 手动计算梯度
gradients = torch.autograd.grad(loss, scores_tensor, retain_graph=True)
print("Manually computed gradients:", gradients)
```

4. 理解梯度的概念对于调试深度学习模型非常重要。梯度表示损失函数对模型参数的敏感程度，即损失函数关于参数的偏导数。在反向传播过程中，梯度被计算出来并用于更新模型的参数。

可能导致梯度中断的操作

1. `.detach()`：将张量从计算图中分离出来，使其不再跟踪梯度。
2. `.item()`：将张量转换为 Python 标量，通常用于获取单个元素的值。
3. `.numpy()`：将张量转换为 NumPy 数组，不再跟踪梯度。
4. `.clone()`：克隆张量，新张量默认不跟踪梯度。
5. **条件语句**：在条件语句中使用张量的值，可能导致梯度计算中断。
6. **(list 的拼接不对也有可能导致梯度中断，但是这里的梯度值应该是参与神经网络的变量的)**

如何打印其他中间变量的梯度

1. 确保中间变量是叶子节点或使用 `retain_grad()`：

```
x = torch.tensor([1.0], requires_grad=True)
y = x + 1
y,retain_grad() # 确保 y 保留梯度
z = y * 2
z.backward()
print("x.grad:", x.grad) # 输出 2.0
print("y.grad:", y.grad) # 输出 2.0
```

2. 使用 `torch.autograd.grad` 手动计算梯度：

```

x = torch.tensor([1.0], requires_grad=True)
y = x + 1
z = y * 2
gradients = torch.autograd.grad(z, [x, y], retain_graph=True)
print("x.grad:", gradients[0]) # 输出 2.0
print("y.grad:", gradients[1]) # 输出 2.0

```

变量的梯度到底是什么意思

梯度是一个向量，表示损失函数对模型参数的偏导数。在反向传播过程中，梯度被计算出来并用于更新模型的参数。具体来说：

- **损失函数**：假设损失函数为 (L)，模型参数为 (w)。
- **梯度**：梯度 ($\nabla_w L$) 表示损失函数 (L) 对参数 (w) 的偏导数。

在 PyTorch 中，每个张量（如果 `requires_grad=True`）都有一个 `.grad` 属性，用于存储该张量的梯度。

总结

1. 确保所有需要计算梯度的张量都设置了 `requires_grad=True`。
2. 使用 `retain_grad()` 保留非叶子节点的梯度。
3. 检查损失函数的计算，确保没有断开计算图的操作。
4. 使用 `torch.autograd.grad` 手动计算梯度，帮助调试。
5. 好的，让我们详细解释一下张量、梯度以及它们在深度学习中的作用。

张量 (Tensor)

张量是一种多维数组，可以是一维（向量）、二维（矩阵）或多维。在 PyTorch 中，张量是基本的数据结构，用于表示数据和模型参数。张量可以包含任意维度的数据，不仅仅是标量。

梯度 (Gradient)

梯度是一个向量，表示损失函数对模型参数的偏导数。在深度学习中，梯度用于指导模型参数的更新，以便最小化损失函数。具体来说：

- **损失函数**：假设损失函数为 (L)，模型参数为 (w)。
- **梯度**：梯度 ($\nabla_w L$) 表示损失函数 (L) 对参数 (w) 的偏导数。

梯度的计算和应用

在 PyTorch 中，梯度的计算和应用主要通过以下步骤完成：

1. **前向传播**：计算模型的输出。
2. **计算损失**：根据模型的输出和真实标签计算损失函数。
3. **反向传播**：通过调用 `loss.backward()` 计算损失函数对所有 **需要梯度的张量的梯度**。
4. **更新参数**：使用优化器（如 SGD、Adam 等）更新模型参数。

详细解释

1. **前向传播**：
 - `scores[i] = ILGR_model(fealist_tensor, Rg_tensor)`：通过模型前向传播计算模型的输出。
2. **计算损失**：

- `loss = ranking_loss(scores_tensor, criticality_scores)`: 根据模型的输出和真实标签计算损失函数。

3. 反向传播:

- `loss.backward()`: 计算损失函数对所有需要梯度的张量的梯度。这一步会自动计算并存储梯度。

4. 更新参数:

- `optimizer.step()`: 使用优化器更新模型参数。

5. 打印梯度:

- `if scores_tensor.grad is None`: 检查 `scores_tensor` 的梯度是否为 `None`。
- `print("scores_tensor.grad:", scores_tensor.grad)`: 打印 `scores_tensor` 的梯度。
- `for name, param in ILGR_model.named_parameters()`: 遍历模型的所有参数，检查并打印它们的梯度。

调试建议

1. 确保 `requires_grad=True`:

- 确保所有需要计算梯度的张量都设置了 `requires_grad=True`。

2. 检查梯度是否为 `None`:

- 如果梯度为 `None`，检查是否有断开计算图的操作（如 `.detach()`、`.item()` 等）。

3. 手动计算梯度:

- 使用 `torch.autograd.grad` 手动计算梯度，帮助调试。

```
gradients = torch.autograd.grad(loss, [scores_tensor],  
retain_graph=True)  
print("Manually computed gradients:", gradients)
```

6. 在 PyTorch 中，“需要梯度的张量”是指那些在计算图中需要计算其梯度的张量。这些张量通常用于模型的参数更新。要使一个张量在计算图中需要计算梯度，需要将 `requires_grad=True` 设置为其属性之一。这样，当进行反向传播时，PyTorch 会自动计算并存储这些张量的梯度。

需要梯度的张量

1. **模型参数**: 模型的权重和偏置通常需要计算梯度，因为这些参数会在训练过程中被更新。
2. **中间变量**: 在某些情况下，你可能需要计算中间变量的梯度，以便进行调试或进一步的计算。

示例

模型参数

```
import torch  
import torch.nn as nn  
import torch.optim as optim  
  
# 定义一个简单的线性模型  
class SimpleModel(nn.Module):  
    def __init__(self):
```

```

        super(SimpleModel, self).__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        return self.linear(x)

    # 创建模型实例
model = SimpleModel()

    # 创建优化器
optimizer = optim.SGD(model.parameters(), lr=0.01)

    # 输入数据
x = torch.tensor([[1.0]], requires_grad=True)
y_true = torch.tensor([[2.0]])

    # 前向传播
y_pred = model(x)

    # 计算损失
loss = (y_pred - y_true) ** 2

    # 反向传播
loss.backward()

    # 打印模型参数的梯度
for name, param in model.named_parameters():
    print(f"{name}.grad:", param.grad)

```

在这个例子中，`model.linear.weight` 和 `model.linear.bias` 都是需要梯度的张量，因为它们是模型的参数，会在训练过程中被更新。

中间变量

```

import torch
import torch.nn.functional as F

    # 创建一个输入张量
x = torch.tensor([1.0], requires_grad=True)

    # 前向传播
y = x + 1
y,retain_grad()  # 保留 y 的梯度
z = y * 2

    # 计算损失
loss = z ** 2

    # 反向传播
loss.backward()

    # 打印 x 和 y 的梯度
print("x.grad:", x.grad)
print("y.grad:", y.grad)

```

在这个例子中，`x` 是一个需要梯度的张量，因为它需要计算梯度。`y` 也是一个需要梯度的张量，但默认情况下，非叶子节点不会保存梯度，所以我们需要调用 `y.retain_grad()` 来保留 `y` 的梯度。

梯度的作用

1. **参数更新**: 在优化过程中，梯度用于更新模型的参数。例如，使用梯度下降法更新参数：

$$w_{\text{new}} = w_{\text{old}} - \eta \cdot \nabla_w L$$

其中，(η)是学习率，($\nabla_w L$)是损失函数对参数(w)的梯度。

2. **调试**: 通过打印中间变量的梯度，可以帮助你调试模型，确保梯度计算正确。

总结

- **需要梯度的张量**: 这些张量在计算图中需要计算其梯度，通常包括模型参数和某些中间变量。
- **设置 `requires_grad=True`**: 确保这些张量在计算图中需要计算梯度。
- **使用 `retain_grad()`**: 对于非叶子节点，如果需要保留其梯度，需要调用 `retain_grad()`。

希望这些解释和示例能帮助你更好地理解“需要梯度的张量”及其在深度学习中的作用。如果有更多问题，请随时提问。

7. 损失函数在训练过程中变得很大并且不再下降

当损失函数在训练过程中变得很大并且不再下降时，这可能是由多种原因引起的。以下是一些常见的问题及其解决方法：

1. 学习率过高

- **问题**: 学习率过高会导致梯度更新过大，使得模型权重在参数空间中跳跃，无法收敛。
- **解决方法**: 降低学习率。可以尝试将学习率减半或减小一个数量级，观察损失函数的变化。

2. 梯度爆炸或梯度消失

- **问题**: 梯度爆炸（梯度过大）或梯度消失（梯度过小）都会导致模型无法有效学习。
- **解决方法**:
 - **梯度裁剪**: 使用梯度裁剪（gradient clipping）来防止梯度爆炸。
 - **归一化**: 使用批量归一化（Batch Normalization）或层归一化（Layer Normalization）来稳定梯度。
 - **激活函数**: 使用 ReLU 以外的激活函数，如 LeakyReLU 或 SELU，这些激活函数可以缓解梯度消失问题。

3. 模型过拟合或欠拟合

- **问题**: 模型可能过拟合或欠拟合数据。
- **解决方法**:
 - **正则化**: 增加 L1 或 L2 正则化项，防止模型过拟合。
 - **增加模型复杂度**: 如果模型欠拟合，可以尝试增加模型的复杂度，如增加层数或神经元数量。
 - **数据增强**: 增加训练数据的多样性，使用数据增强技术。

4. 损失函数不合适

- **问题：**使用的损失函数可能不适合当前任务。
- **解决方法：**尝试使用其他损失函数，如均方误差（MSE）、交叉熵（Cross Entropy）等。

5. 数据问题

- **问题：**数据可能存在问题，如数据不均衡、噪声过多等。
- **解决方法：**
 - **数据预处理：**检查数据预处理步骤，确保数据清洗和标准化正确。
 - **数据平衡：**如果数据不均衡，可以使用过采样或欠采样技术。

6. 优化器问题

- **问题：**使用的优化器可能不适合当前任务。
- **解决方法：**尝试更换其他优化器，如 Adam、RMSprop 等。

7. 初始化问题

- **问题：**模型参数的初始化可能不合适。
- **解决方法：**尝试使用不同的初始化方法，如 Xavier 初始化或 Kaiming 初始化。

8. 模型结构问题

- **问题：**模型结构可能不适合当前任务。
- **解决方法：**重新设计模型结构，参考相关领域的最佳实践。

示例代码

以下是一个示例，展示了如何调整学习率和使用梯度裁剪：

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from scipy.special import comb

# 假设其他部分的代码已经定义好
# ...

# 定义损失函数
def ranking_loss(scores, true_ranks):
    loss = 0
    for i in range(len(scores)):
        for j in range(i + 1, len(scores)):
            r_ij = true_ranks[i] - true_ranks[j]
            y_hat_ij = scores[i] - scores[j]
            f_r_ij = F.sigmoid(r_ij.float())
            loss += -f_r_ij * torch.log(F.sigmoid(y_hat_ij)) - (1 - f_r_ij) * torch.log(1 - F.sigmoid(y_hat_ij))
    return loss / int(comb(len(scores), 2))

# 假设模型和数据已经定义好
ILGR_model = ... # 你的模型
optimizer = optim.Adam(ILGR_model.parameters(), lr=0.001) # 尝试降低学习率
```

```

un_location_list = ... # 你的位置列表
un_fea_list = ... # 你的特征列表
R_g = ... # 你的 R_g 列表
criticality_scores = ... # 你的真实排名
args = ... # 你的训练参数
select_node = ... # 选择的节点
location_list = ... # 位置列表
fea_list = ... # 特征列表
fea_list_tensor = ... # 特征列表的张量
scores = [] # 存储模型输出的列表
loss_history = [] # 存储损失的历史记录

for epoch in range(args.max_epoch): # 假设训练100个epoch
    for i, (location, fea) in enumerate(zip(un_location_list,
un_fea_list)):
        location_list[select_node] = location
        fea_list[select_node] = fea
        ILGR_model.train()
        R_g_tensor = R_g[i]
        optimizer.zero_grad()
        scores[i] = ILGR_model(fea_list_tensor, R_g_tensor)

        scores_tensor = torch.stack(scores).requires_grad_(True).to(device)
        # 保留非叶子节点的梯度
        scores_tensor,retain_grad_()
        loss = ranking_loss(scores_tensor, criticality_scores)

        # 打印损失值
        print('epoch:{}, loss:{}'.format(epoch, loss.item()))

        # 反向传播
        loss.backward()

        # 梯度裁剪
        torch.nn.utils.clip_grad_norm_(ILGR_model.parameters(),
max_norm=1.0)

        # 更新模型参数
        optimizer.step()
        loss_history.append(loss.item())

```

进一步调试

- 可视化损失曲线**: 绘制损失函数随训练轮次的变化曲线, 观察是否存在震荡或饱和现象。
- 检查梯度**: 在反向传播后, 检查模型参数的梯度, 确保它们不是零或非常小的值。
- 简化模型**: 尝试用一个更简单的模型来测试相同的损失函数和优化器, 确保它们能正常工作。

希望这些方法能帮助你解决损失函数不下降的问题! 如果有更多问题, 请随时提问。

- 修改RegressionModule的激活函数relu为Softmax

```
epoch:0, loss:0.6791266798973083
epoch:1, loss:0.6791266798973083
epoch:2, loss:0.6791263818740845
epoch:3, loss:0.6791262030601501
epoch:4, loss:0.679125964641571
epoch:5, loss:0.679125964641571
```

损失下降很慢

```
9. scores_tensor_scores =
    torch.tensor(np.argsort(np.array(tensors))).to(device)
```

对预测的分数进行排序，用排序进行学习

排序值过大过小很正常，但是变成概率后会有0值

```
r 30  y 35
f_y 1.0
f_r 1.0  1-f_r 0.0
log 0.0  1- log -inf
loss nan
```

y过大，导致 $F.sigmoid(y_{hat_ij}) = 1$, $\log = 0$, $\log 0 = -\infty$

r过大，导致 $F.sigmoid(r_{ij}) = 1$ ，则全看 \log 的值

Sigmoid 函数是一种常用的激活函数，广泛应用于神经网络中，特别是在二分类问题中。Sigmoid 函数将实数域映射到 (0, 1) 区间内，这使得它特别适合用于将模型的输出转换为概率值。

Sigmoid 函数的数学表达式

Sigmoid 函数的数学表达式为：

$$[\sigma(x) = \frac{1}{1 + e^{-x}}]$$

Sigmoid 函数的性质

- 范围**: Sigmoid 函数的输出范围是 (0, 1)，这使得它非常适合用于表示概率。
- 单调性**: Sigmoid 函数是单调递增的，这意味着随着输入的增加，输出也会增加。
- 平滑性**: Sigmoid 函数是连续且可微的，这使得它可以用于梯度下降等优化算法。
- 对称性**: Sigmoid 函数在 $x = 0$ 处是对称的，即 $(\sigma(0) = 0.5)$ 。

Sigmoid 函数的图形

Sigmoid 函数的图形是一个 S 形曲线，如下所示：

在 PyTorch 中使用 Sigmoid 函数

在 PyTorch 中，可以使用 `torch.sigmoid` 函数来计算 Sigmoid 函数的值。以下是一个示例代码：

```
import torch
import torch.nn.functional as F

# 创建一个示例张量
x = torch.tensor([-1.0, 0.0, 1.0, 2.0])
```

```
# 计算 Sigmoid 函数的值
sigmoid_values = torch.sigmoid(x)

# 或者使用 F.sigmoid
sigmoid_values = F.sigmoid(x)

# 打印结果
print("Input Tensor:")
print(x)
print("Sigmoid values:")
print(sigmoid_values)
```

输出

```
Input Tensor:
tensor([-1.0000,  0.0000,  1.0000,  2.0000])
Sigmoid values:
tensor([0.2689,  0.5000,  0.7311,  0.8808])
```

Sigmoid 函数的应用

1. **二分类问题**: Sigmoid 函数常用于二分类问题，将模型的输出转换为概率值。例如，在 logistic 回归中，Sigmoid 函数用于将线性组合的输出转换为概率。
2. **神经网络**: 在神经网络中，Sigmoid 函数可以用作隐藏层的激活函数，尽管在现代深度学习中，ReLU 和其变种（如 LeakyReLU、ELU 等）更常用，因为它们可以缓解梯度消失问题。
3. **损失函数**: 在二分类问题中，通常与 Sigmoid 函数结合使用的是二元交叉熵损失函数（Binary Cross-Entropy Loss）。

总结

- **Sigmoid 函数**: 将实数域映射到 (0, 1) 区间内，适用于表示概率。
- **PyTorch 实现**: 使用 `torch.sigmoid` 或 `F.sigmoid` 计算 Sigmoid 函数的值。
- **应用**: 常用于二分类问题、神经网络和损失函数。

希望这些解释和示例能帮助你更好地理解 Sigmoid 函数及其在 PyTorch 中的使用。如果有更多问题，请随时提问。

- **使用 `torch.log1p`**: 计算 $\log(1+x)\log(1+x)$ 以避免直接计算 $\log(0)$ 的问题。
- 10. 直接对预测分数排序，导致梯度中断，使用了软排序softsort

所谓梯度中断，是使用的运算不可微，导致无法反向传播（求偏导）

在深度学习中，排序操作通常会导致梯度中断，因为排序是一个非可导操作。为了在保持梯度传递的同时进行排序，可以考虑使用一些近似的可微方法。以下是几种常见的方法：

1. 软排序 (SoftSort)

软排序是一种近似排序的方法，可以保持梯度传递。它通过构建一个软排序矩阵来近似排序操作。

软排序通过引入平滑的近似方法，使得排序操作变得可微，从而可以在梯度下降等优化算法中使用。

```
def softsort(x, tau=0.1):
```

```

    ....
    输入向量 x 的形状为 [n]，则 x.unsqueeze(1) 的形状为 [n, 1]，  

x.unsqueeze(0) 的形状为 [1, n]。
    通过广播机制，x.unsqueeze(1) - x.unsqueeze(0) 会生成一个形状为 [n, n] 的  

    矩阵，  

    其中每个元素表示原向量中两个元素的差值。
    ....
    # 计算每对元素的差值
    pairwise_diff = x.unsqueeze(1) - x.unsqueeze(0)
    ....
    计算这些差值的绝对值，以确保相似度矩阵中的所有元素都是非负的。  

    将这些绝对差值转换为相似度，我们取负值  

    (将较大的差值转换为较小的相似度值。因为较大的差值表示两个元素相距较远，相似度较  

    低。)  

    并除以温度参数 tau。  

    (除以温度参数 tau 是为了调整相似度的平滑程度。较小的 tau 值会使相似度矩阵中的值  

    差异更大，  

    从而更接近于硬排序；较大的 tau 值会使相似度矩阵中的值更加平滑，从而更接近于软排  

    序。)  

    温度参数 tau 控制着相似度矩阵的平滑程度。  

    较小的 tau 值会使相似度矩阵更接近于硬排序（即 one-hot 编码），而较大的 tau 值  

    会使相似度矩阵更加平滑
    ....
    # 计算相似度矩阵
    similarity_matrix = -pairwise_diff.abs() / tau
    ....
    使用 softmax 函数将相似度矩阵转换为一个软排序矩阵。  

    这个矩阵的每一行表示一个元素在排序后的位置的概率分布。
    ....
    # 计算软排序矩阵
    soft_permutation_matrix = F.softmax(similarity_matrix, dim=1)
    # 计算软排序后的张量
    soft_sorted_x = soft_permutation_matrix @ x
    return soft_sorted_x

```

2. Top-K 松弛操作

Top-K 松弛操作通过引入温度参数来近似 Top-K 操作，使其可微。

3. Gumbel-Softmax 重参数化

Gumbel-Softmax 重参数化技巧可以用于近似离散操作，使其可微。

其他方法

1. Top-K 松弛操作：

- 使用温度参数 `tau` 来近似 Top-K 操作。
- 例如，可以使用 `F.softmax` 和 `topk` 函数来实现。

2. Gumbel-Softmax 重参数化：

- 使用 Gumbel-Softmax 技巧来近似离散操作。
- 适用于多分类问题中的离散选择。

希望这些方法能帮助你在保持梯度传递的同时进行排序操作。如果有更多问题，请随时提
问。

11. Softmax

`softmax` 函数能够将一个向量转换为概率分布，并且在软排序中实现平滑的排序效果，其背后的原理可以从以下几个方面来理解：

1. Softmax 函数的定义

`softmax` 函数的定义如下：

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

其中， (\mathbf{x}) 是一个长度为 (n) 的向量， $\text{softmax}(\mathbf{x})_i$ 表示向量 (\mathbf{x}) 经过 `softmax` 函数变换后第 (i) 个元素的值。

2. 归一化

`softmax` 函数将每个元素 (x_i) 映射到一个范围在 $((0, 1))$ 之间的值，并且所有这些值的和为 1。这意味着 `softmax` 函数将向量 (\mathbf{x}) 转换为一个概率分布。

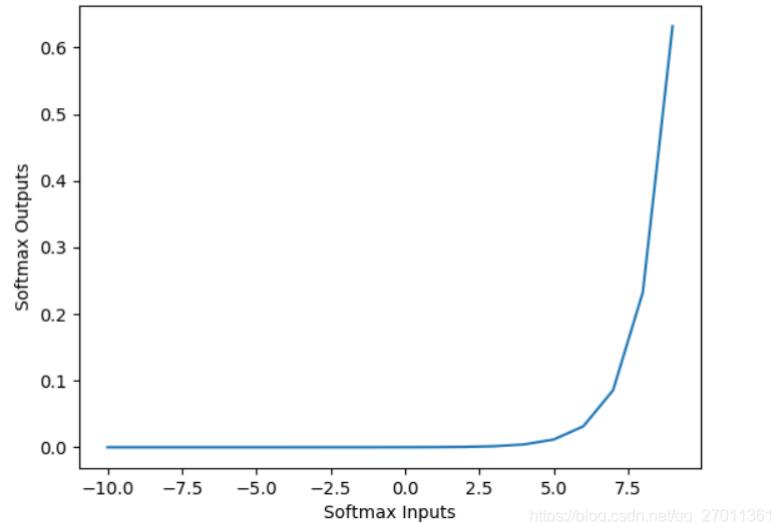
3. 温度参数 (τ)

在软排序中，通常会引入一个温度参数 (τ) ，使得 `softmax` 函数的定义变为：

$$\text{softmax}(\mathbf{x}/\tau)_i = \frac{e^{x_i/\tau}}{\sum_{j=1}^n e^{x_j/\tau}}$$

温度参数 (τ) 控制着 `softmax` 函数的平滑程度：

- **当 (τ) 较小时：** `softmax` 函数的结果更加尖锐，接近于 one-hot 编码。这是因为较小的 (τ) 会使指数项之间的差异放大，导致较大的值占据主导地位，而其他值接近于 0。
- **当 (τ) 较大时：** `softmax` 函数的结果更加平滑，每个值都比较接近。这是因为较大的 (τ) 会使指数项之间的差异缩小，从而使每个值都有一定的概率。



12. 对两种参数归一化

```
true_ranks = (true_ranks1 - torch.min(true_ranks1)) /  
(torch.max(true_ranks1) - torch.min(true_ranks1))
```

13. loss /int(comb(len(scores), 2))

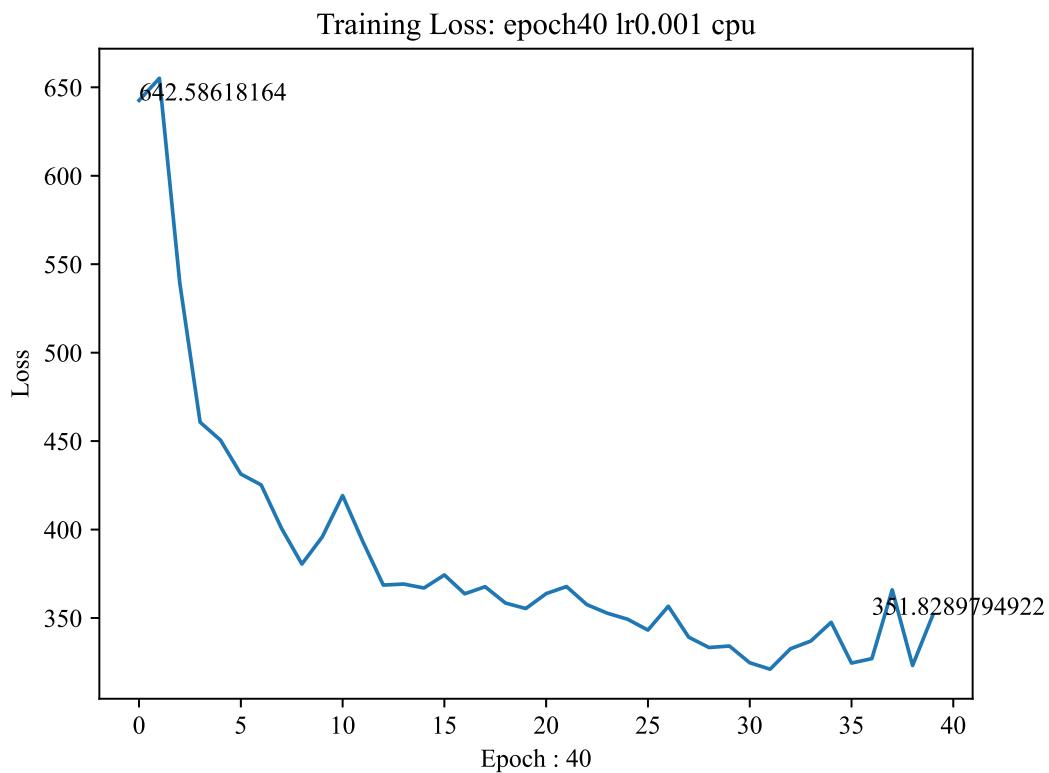
会导致损失值不贴和吧，，，

14. 直接用CrossEntropyLoss

```
epoch:0, loss:642.586181640625
epoch:1, loss:655.1332397460938
epoch:2, loss:539.5330200195312
epoch:3, loss:460.69256591796875
epoch:4, loss:450.5557861328125
epoch:5, loss:431.3824157714844
epoch:6, loss:425.2432556152344
epoch:7, loss:400.6612548828125
epoch:8, loss:380.50732421875
epoch:9, loss:395.83489990234375
epoch:10, loss:419.219482421875
epoch:11, loss:393.0357971191406
epoch:12, loss:368.60174560546875
epoch:13, loss:369.1983947753906
epoch:14, loss:366.9642028808594
epoch:15, loss:374.33819580078125
epoch:16, loss:363.66650390625
epoch:17, loss:367.6988830566406
epoch:18, loss:358.4563903808594
epoch:19, loss:355.3465576171875
epoch:20, loss:363.7801208496094
epoch:21, loss:367.78167724609375
epoch:22, loss:357.5622253417969
epoch:23, loss:352.73077392578125
epoch:24, loss:349.304931640625
epoch:25, loss:343.16790771484375
epoch:26, loss:356.6529541015625
epoch:27, loss:339.1769714355469
epoch:28, loss:333.3101806640625
epoch:29, loss:334.16046142578125
epoch:30, loss:324.69342041015625
```

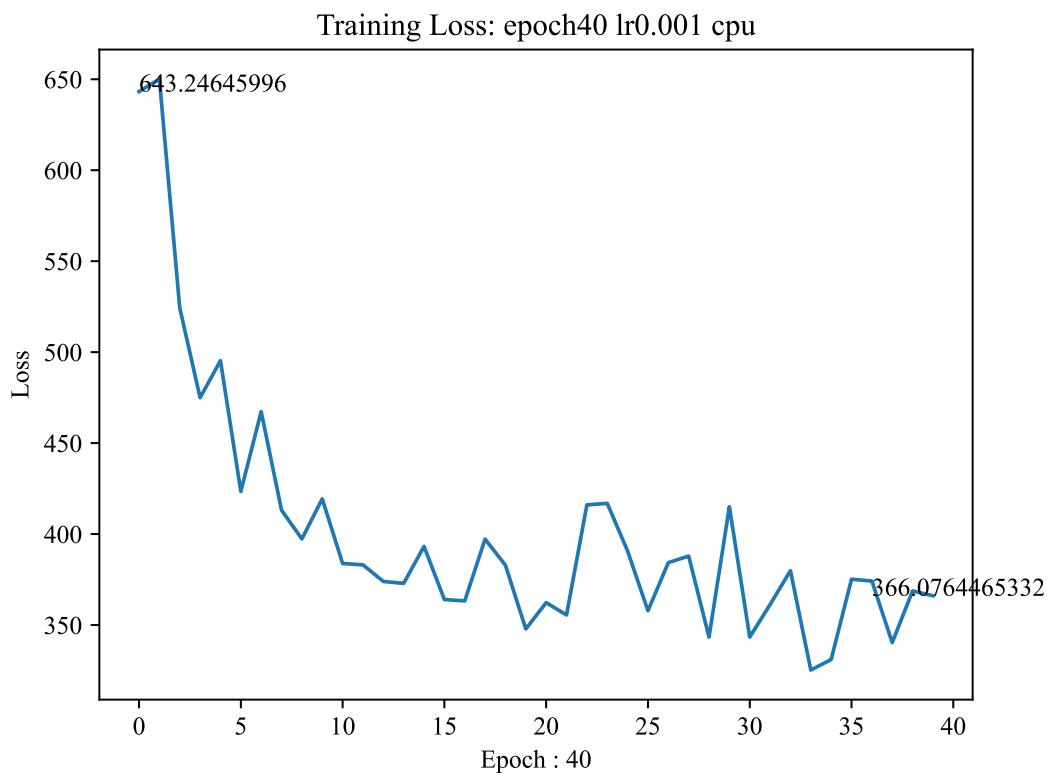
好像效果好一点：下降的更快

(排序, 排序) (relu) (CrossEntropyLoss)



15. 修改RegressionModule的激活函数relu为leaky_relu, 很异常

(排序, 排序) (leaky_relu) (CrossEntropyLoss)



16.

损失函数呈现齿状下降（即损失值在下降过程中有明显的波动）是训练过程中常见的现象，这可能是由多种因素引起的。以下是一些可能的原因及相应的解决方法：

1. 学习率问题

- **问题：**学习率可能设置得过高，导致梯度更新过大，从而使损失值在下降过程中出现波动。
- **解决方法：**
 - **降低学习率：**尝试逐步降低学习率，观察损失值的变化。
 - **使用学习率调度器：**使用学习率调度器（如 `torch.optim.lr_scheduler.ReduceLROnPlateau`）动态调整学习率，以便在训练过程中逐渐降低学习率。

2. 批量大小问题

- **问题：**批量大小（batch size）可能设置得过小，导致每次更新的梯度噪声较大。
- **解决方法：**
 - **增加批量大小：**尝试增加批量大小，以减少梯度噪声，使损失值更加平滑。

3. 数据预处理问题

- **问题：**数据可能没有很好地预处理，导致模型难以学习。
- **解决方法：**
 - **归一化或标准化：**确保输入数据经过归一化或标准化处理，使数据分布在合理范围内。
 - **检查数据分布：**确保数据分布合理，没有异常值或极端值。

4. 模型复杂度问题

- **问题：**模型可能过于复杂或过于简单。
- **解决方法：**
 - **增加模型复杂度：**如果模型过于简单，可能无法捕捉数据的复杂性。可以增加隐藏层或增加神经元数量。
 - **减少模型复杂度：**如果模型过于复杂，可能会过拟合。可以减少隐藏层或减少神经元数量。

5. 正则化问题

- **问题：**模型可能过拟合或欠拟合。
- **解决方法：**
 - **增加正则化：**使用 L1 或 L2 正则化来防止过拟合。
 - **减少正则化：**如果模型欠拟合，可以减少正则化强度。

6. 梯度问题

- **问题：**可能存在梯度消失或梯度爆炸问题。
- **解决方法：**
 - **梯度裁剪：**使用梯度裁剪（`torch.nn.utils.clip_grad_norm_`）来防止梯度爆炸。
 - **使用更稳定的激活函数：**例如，使用 ReLU 的变体（如 LeakyReLU 或 PReLU）来缓解梯度消失问题。

7. 数据加载问题

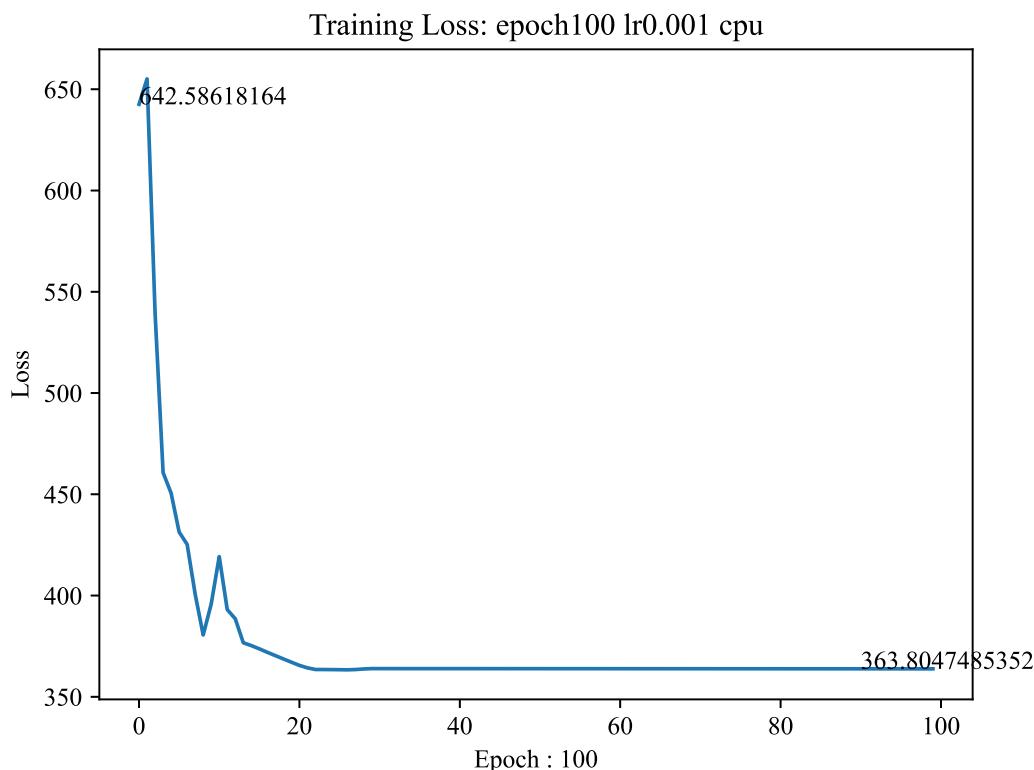
- **问题：**数据加载过程中可能存在随机性，导致每次迭代的数据顺序不同。
- **解决方法：**
 - **固定随机种子：**确保数据加载的随机性是固定的，可以使用 `torch.manual_seed` 和 `numpy.random.seed` 固定随机种子。
 - **数据增强：**如果使用数据增强，确保增强方式一致。

进一步调试

1. **检查数据分布：**确保数据分布合理，没有异常值或极端值。
2. **增加正则化：**如果模型过拟合，可以尝试增加 L2 正则化。
3. **调整模型结构：**尝试增加或减少隐藏层，调整神经元数量。
4. **使用更稳定的激活函数：**例如，使用 LeakyReLU 或 PReLU。

希望这些方法能帮助你减少损失值的波动，使损失曲线更加平滑。如果有更多问题，请随时提问。

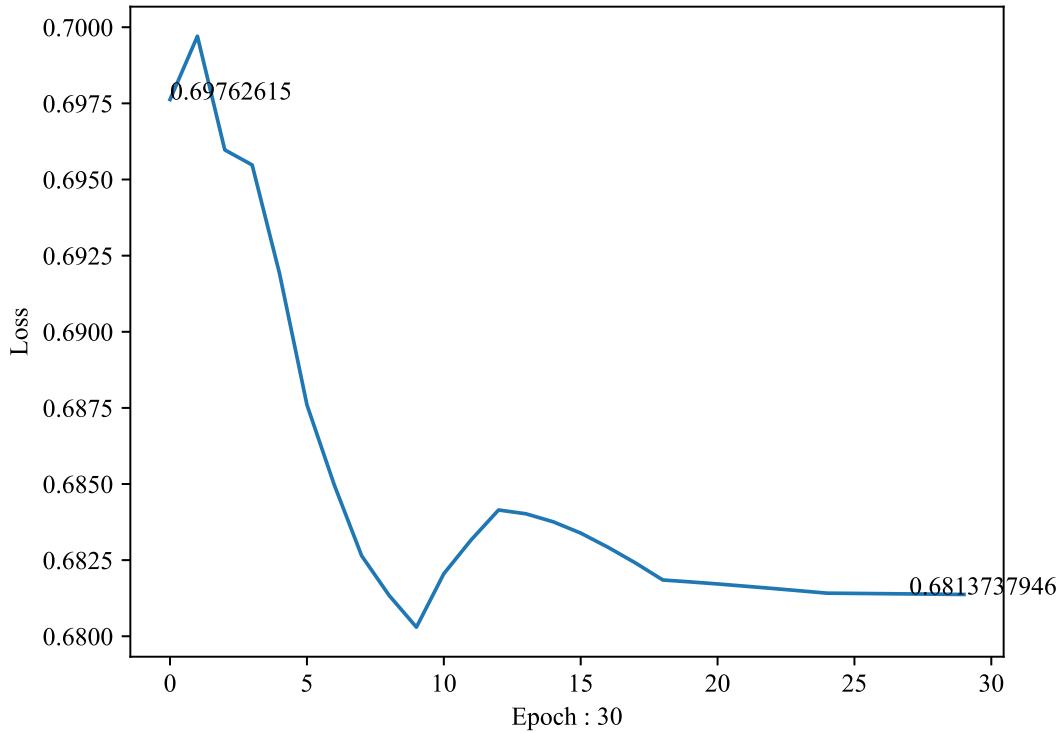
17. 学习率调度器尝试`torch.optim.lr_scheduler.ReduceLROnPlateau`
(排序, 排序) (relu) (CrossEntropyLoss) (学习率调度器`patience=4`)



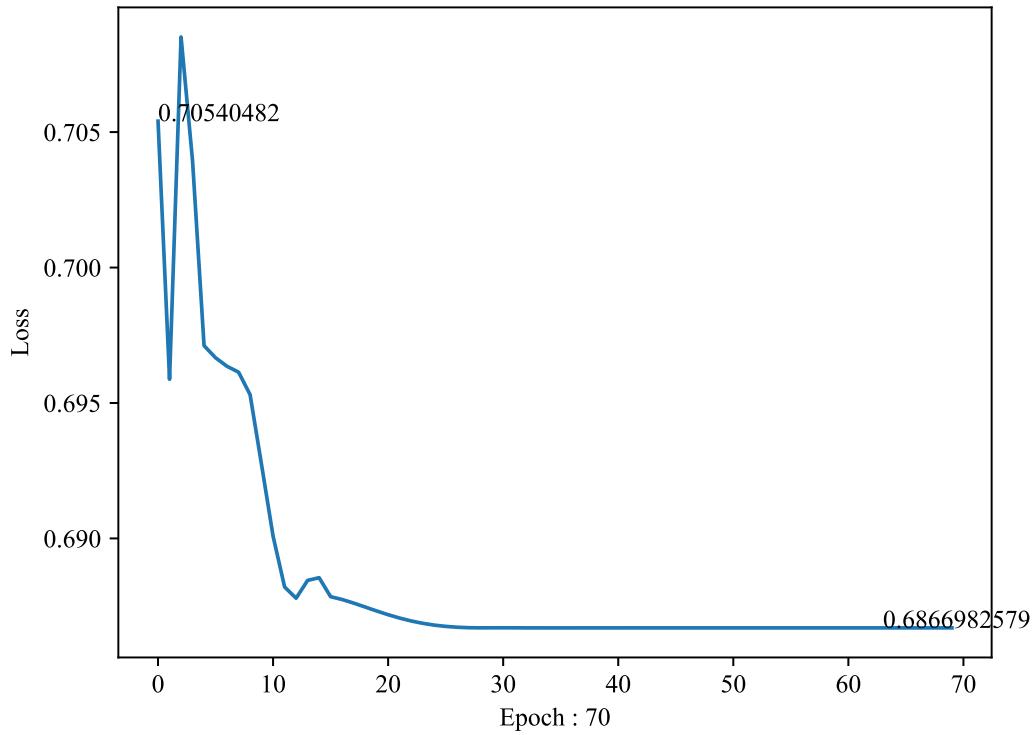
确实可以让损失函数平稳一些，但是下降很缓慢，(lr变小后不能变大吗)

(分数, 分数) (relu) (loss) (学习率调度器)

Training Loss: epoch30 lr0.01 cpu

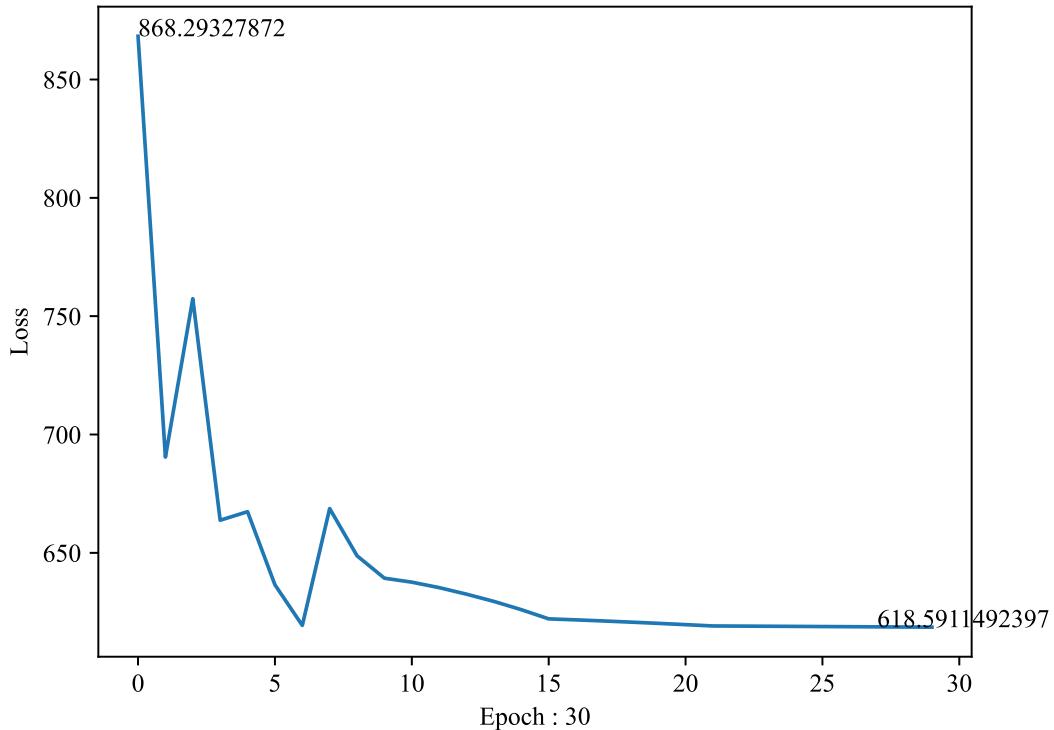


Training Loss: epoch70 lr0.01 cpu



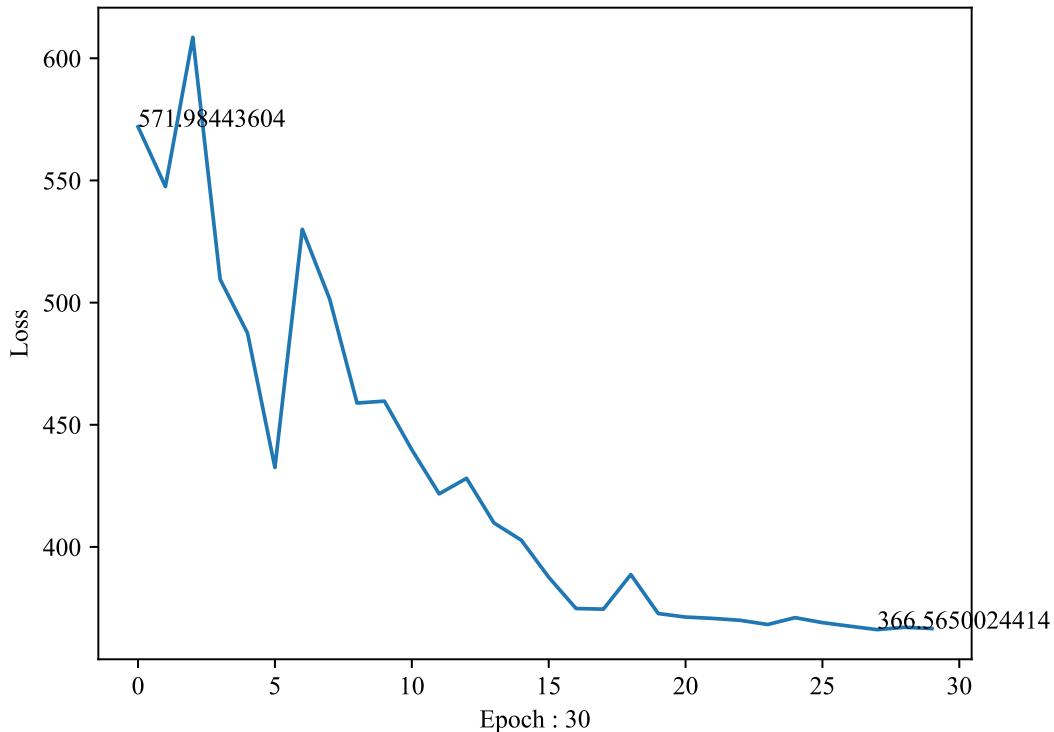
(排序, 排序) (leaky_relu) (CrossEntropyLoss) (学习率调度器)

Training Loss: epoch30 lr0.001 cpu



(排序, 排序) (leaky_relu) (CrossEntropyLoss) (无学习率调度器)

Training Loss: epoch30 lr0.001 cpu

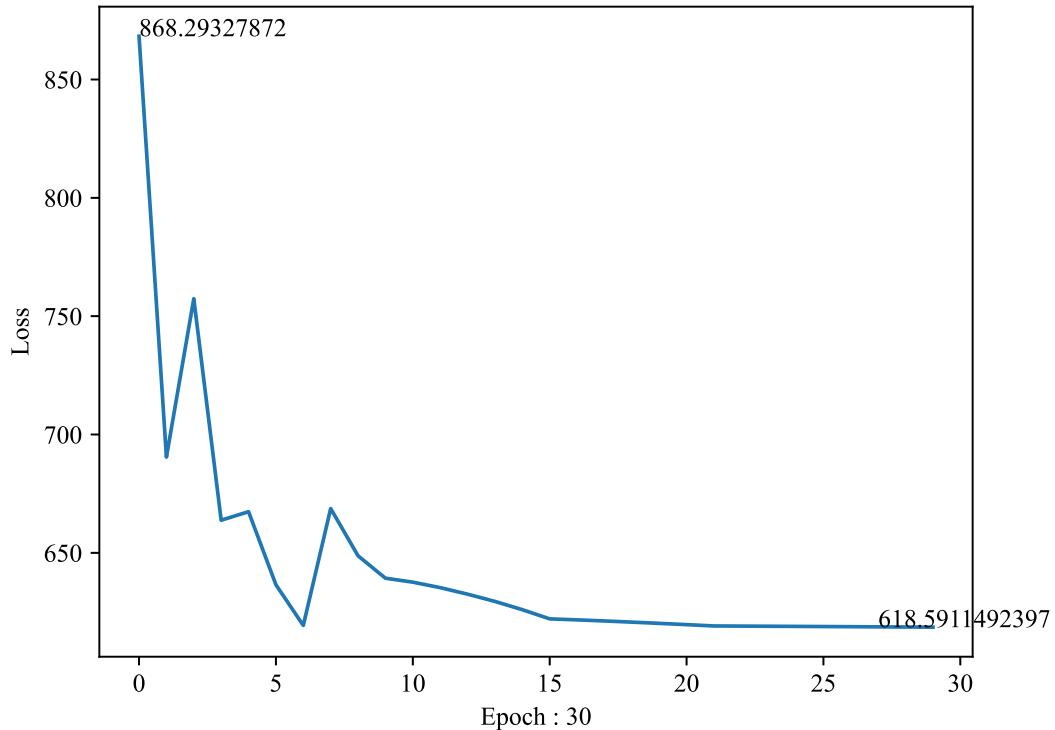


18. 反向传播在做什么

损失值大小本身无所谓，重要在于损失函数的公式本身，是对学习参数的求导，所以重点在于预测值在损失函数中的形态

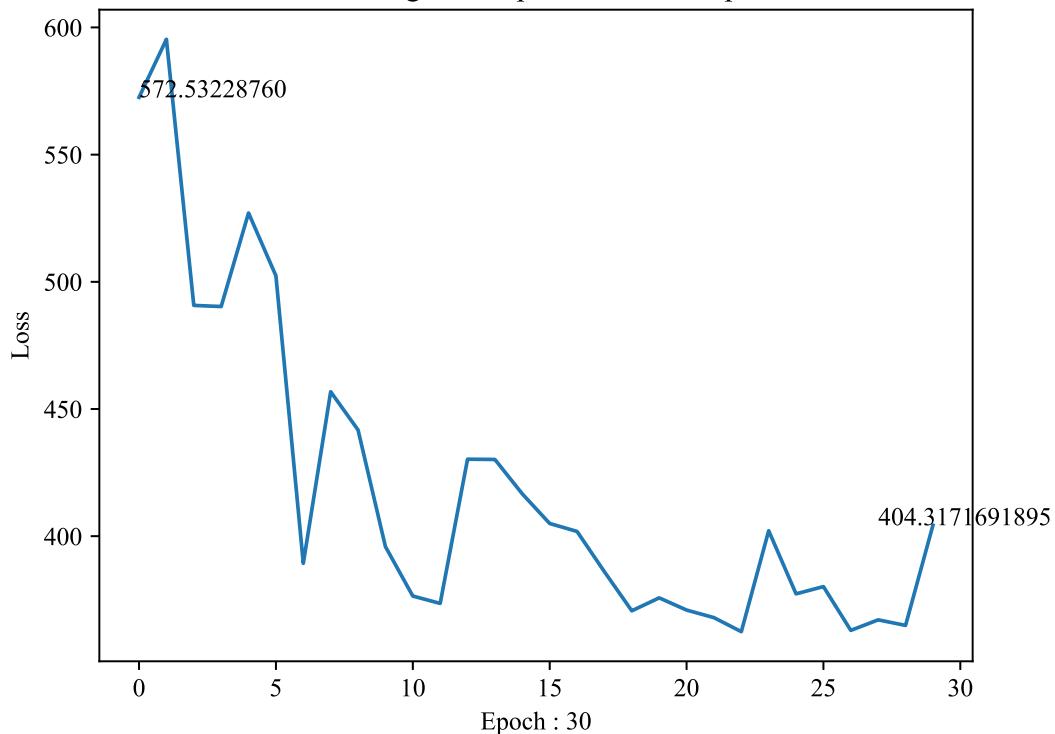
19. (分数, 分数) (relu) (CrossEntropyLoss) (学习率调度器)

Training Loss: epoch30 lr0.001 cpu

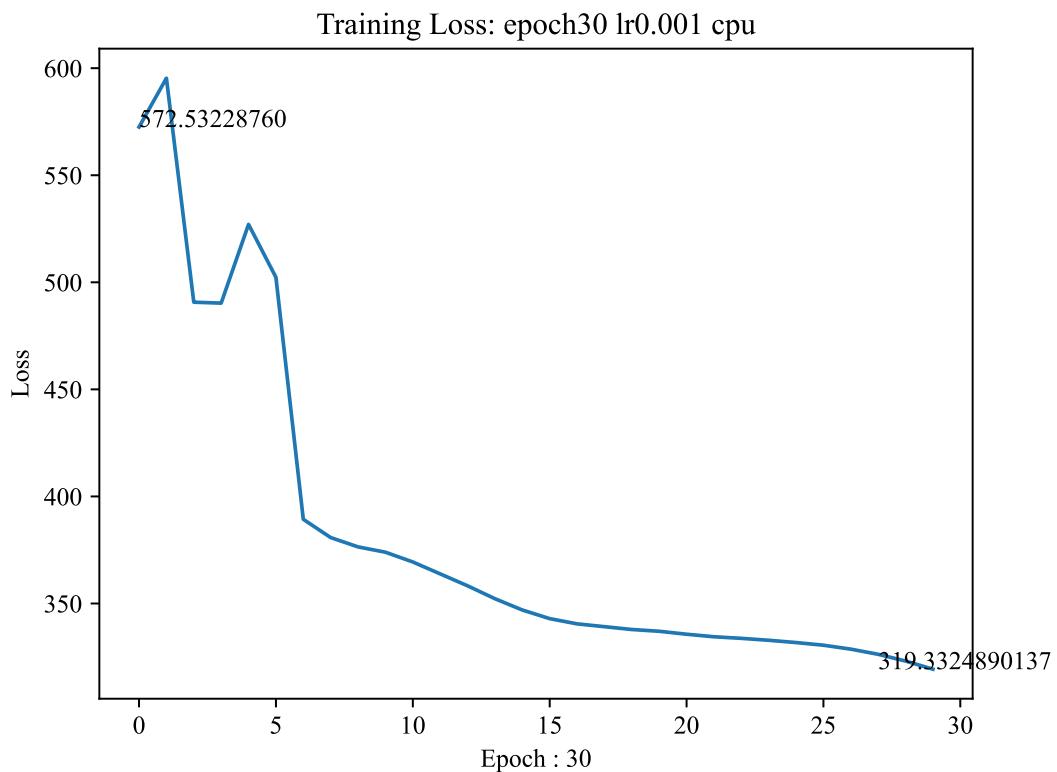


(排序, 排序) (relu) (CrossEntropyLoss) (无学习率调度器)

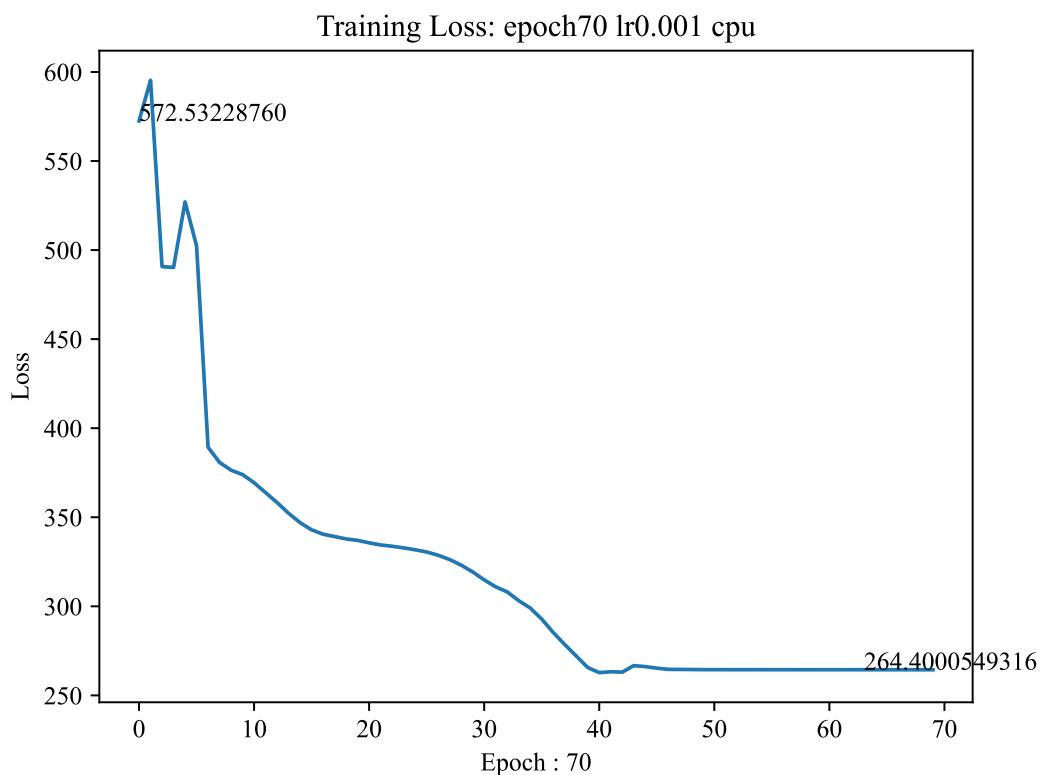
Training Loss: epoch30 lr0.001 cpu



(排序, 排序) (relu) (CrossEntropyLoss) (学习率调度器)

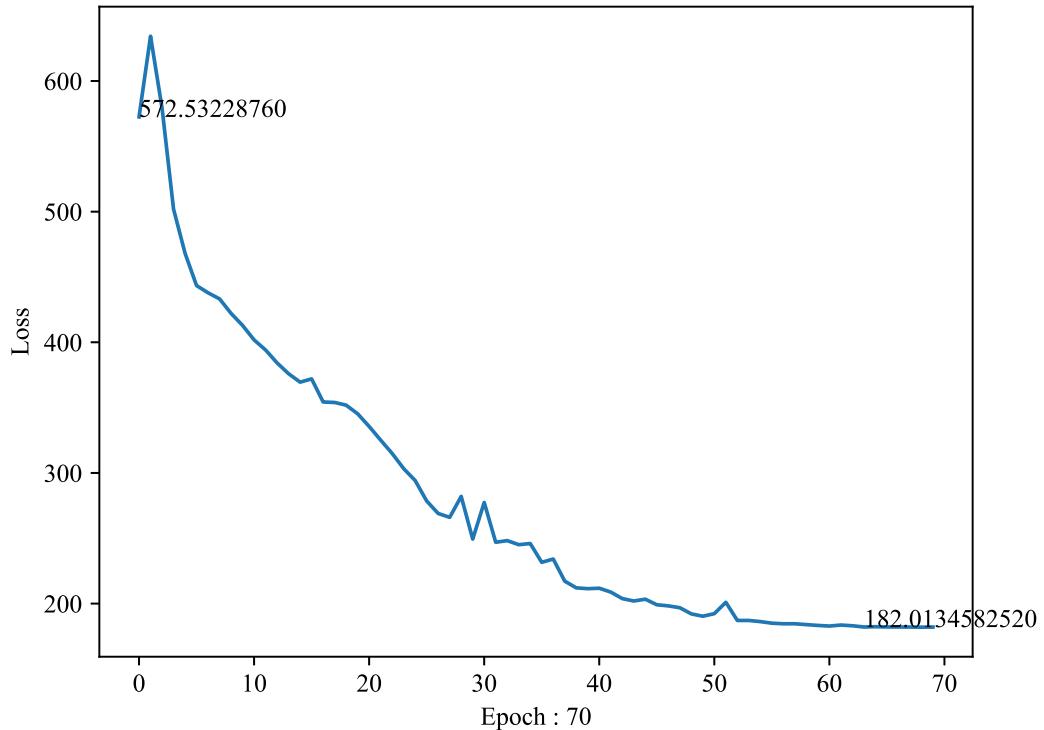


(排序, 排序) (relu) (CrossEntropyLoss) (学习率调度器)



(排序, 排序) (relu) (CrossEntropyLoss) (学习率调度器)

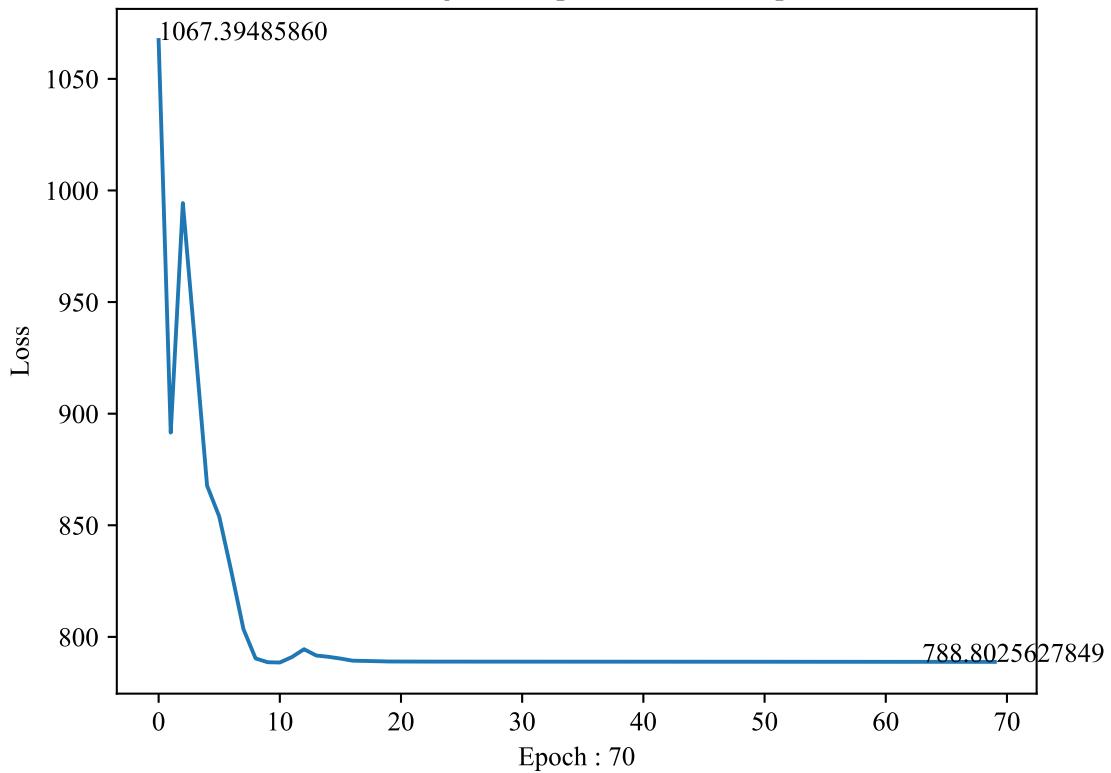
Training Loss: epoch70 lr0.0001 cpu



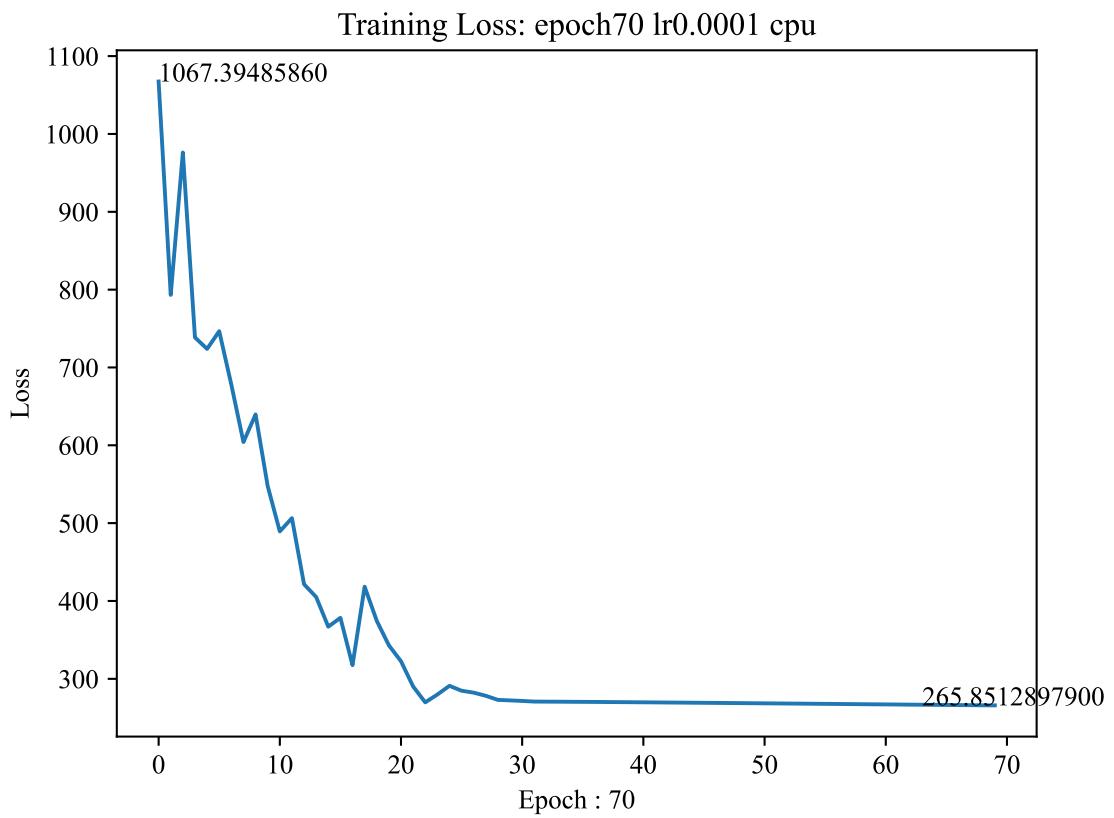
20. 都进行软排序，不对劲

(排序, 排序) (都软排序) (relu) (CrossEntropyLoss) (学习率调度器)

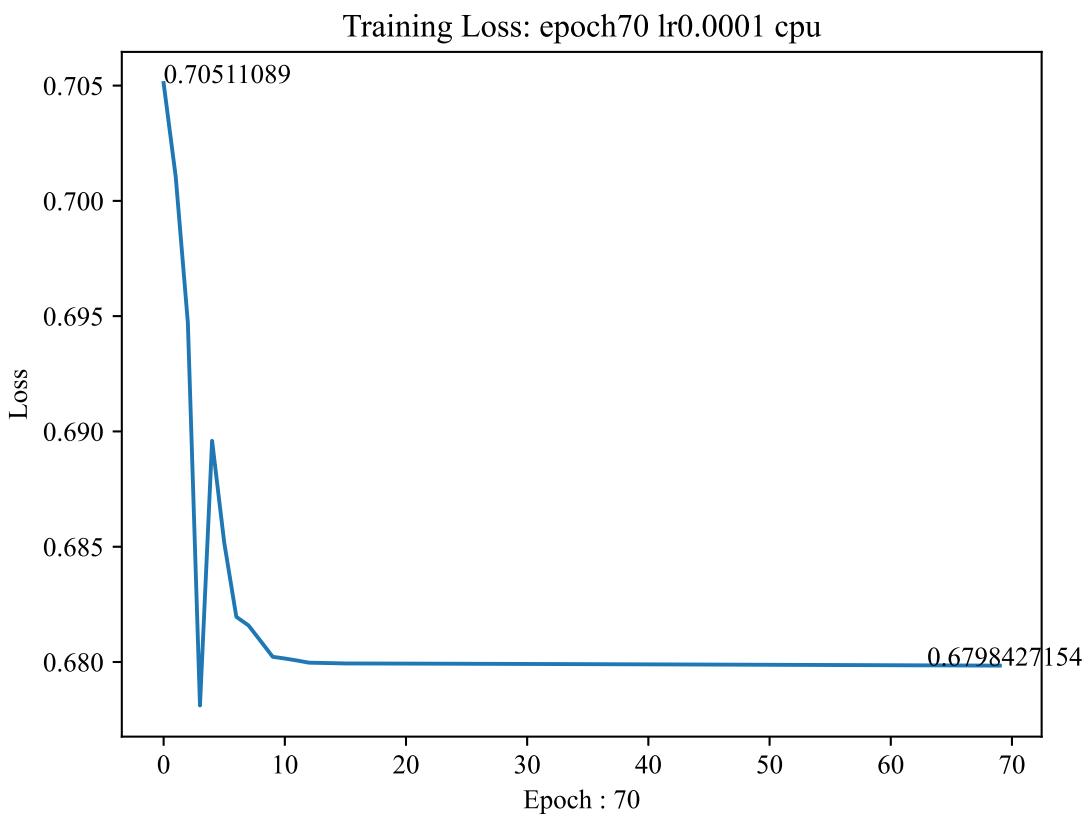
Training Loss: epoch70 lr0.001 cpu



(排序, 排序) (都软排序) (relu) (CrossEntropyLoss) (学习率调度器)

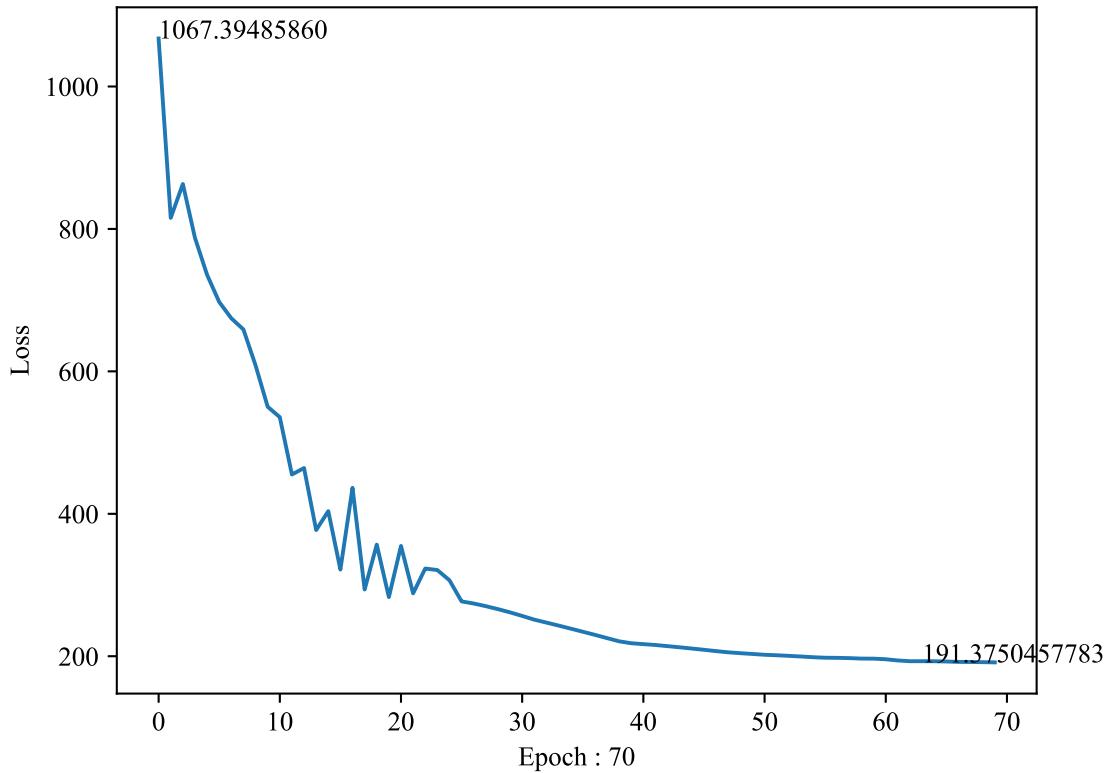


(排序, 排序) (都软排序) (relu) (loss) (学习率调度器)



(排序, 排序) (都软排序) (relu) (CrossEntropyLoss) (学习率调度器)

Training Loss: epoch70 lr1e-05 cpu

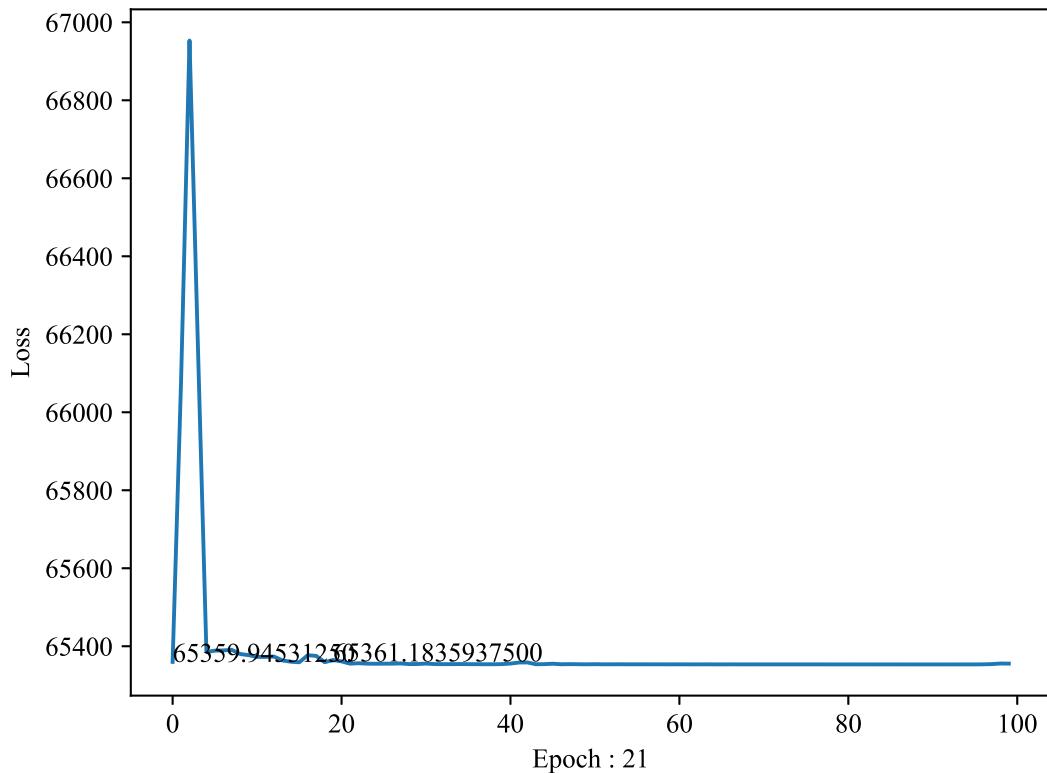


241118

1. 模型构筑好了但是不下降，有问题

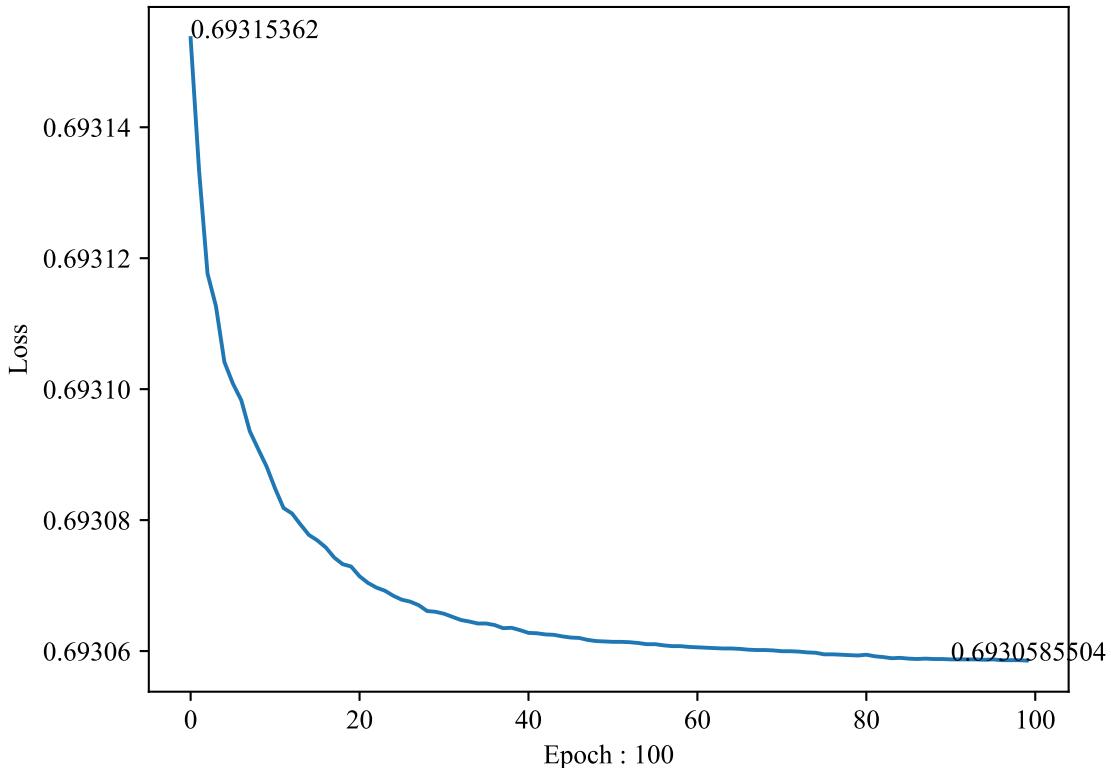
1. 损失函数
2. 单个点还是所有点
3. 优化是不是没有优化上
4. 线性层到底在做什么，注意力机制又在做什么
5. batch??
6. cuda的也有问题

epoch_21_lr_0.001



修改了lr, lr太大, 反而不能收敛

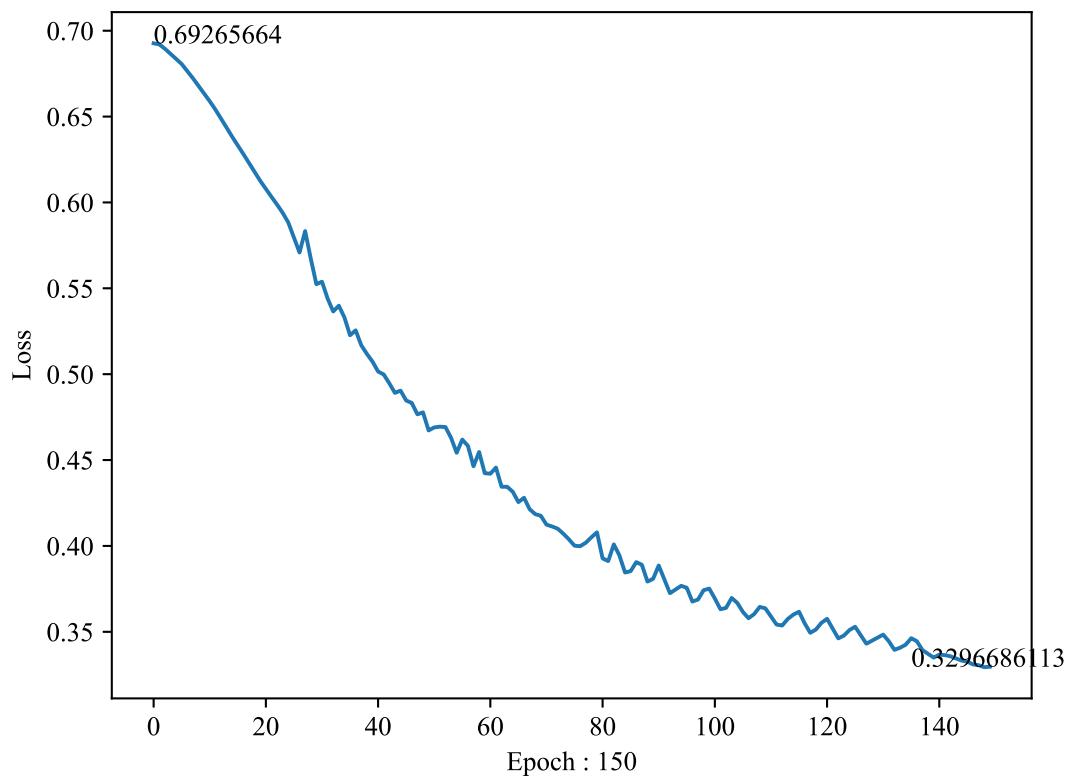
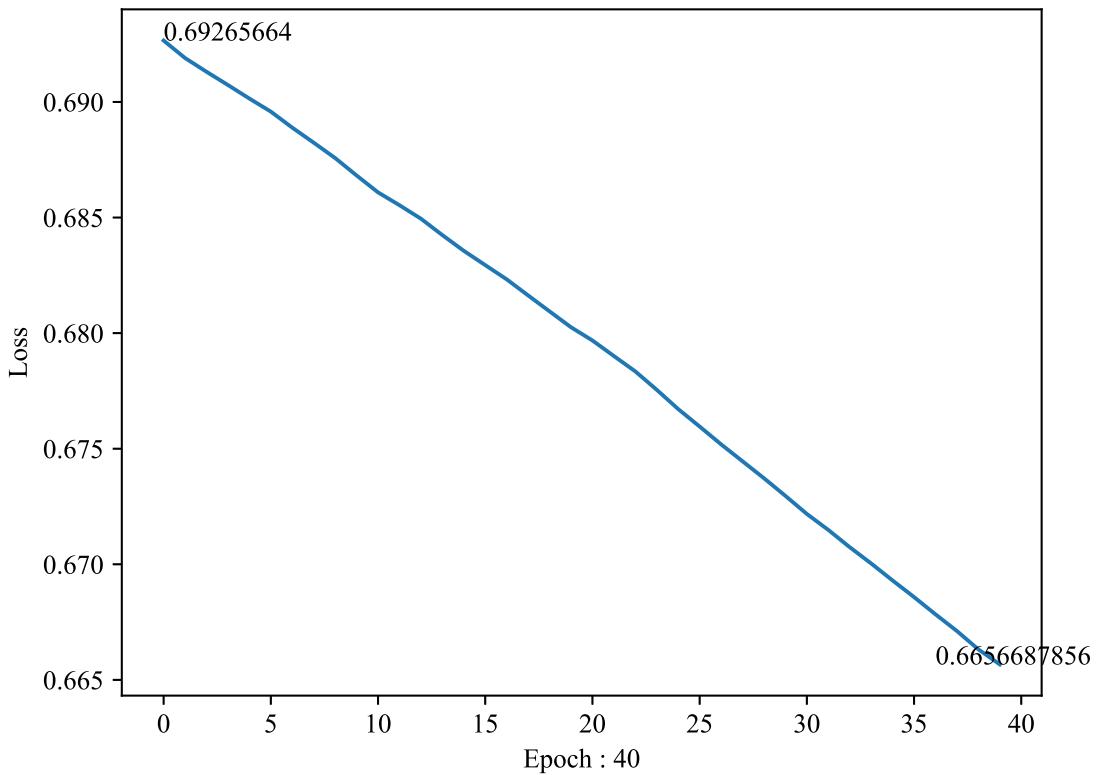
epoch_100_lr_1e-05



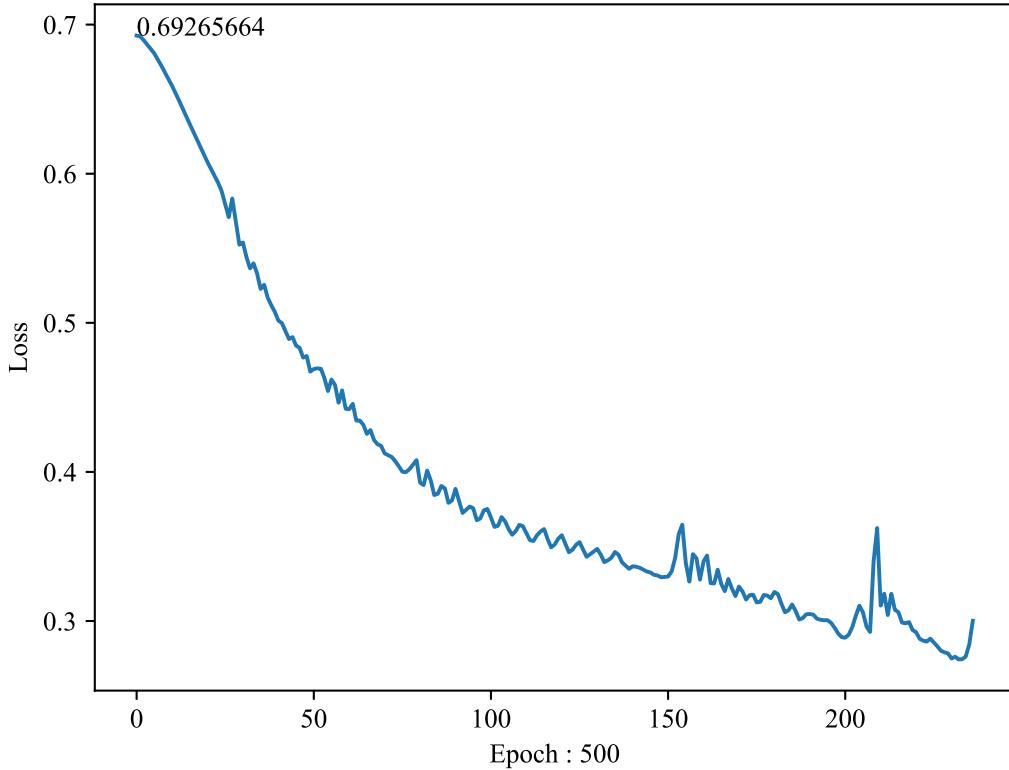
loss = ranking_loss(scores, criticality_scores)

需要大一点的epoch

epoch_40_lr_1e-05



比用图阻值计算的损失更小，但是下降的很慢



在训练深度学习模型时，如果损失值（loss）变为 `NaN`（Not a Number），这通常表示训练过程中出现了某种问题。`NaN` 值的出现可能是由多种原因引起的，以下是一些常见的原因及相应的解决方法：

1. 学习率过高

高学习率可能导致梯度爆炸，从而使损失值变为 `NaN`。

解决方案：

- **降低学习率：**尝试使用更小的学习率，例如从 0.01 降低到 0.001 或 0.0001。
- **学习率调度：**使用学习率调度器（如 `torch.optim.lr_scheduler.StepLR` 或 `ReduceLROnPlateau`）在训练过程中逐渐降低学习率。

2. 梯度爆炸

梯度爆炸是指梯度值变得非常大，导致参数更新过大，从而使损失值变为 `NaN`。

解决方案：

- **梯度裁剪：**使用梯度裁剪技术（如 `torch.nn.utils.clip_grad_norm_`）来限制梯度的大小。

```
import torch.nn.utils as utils

for epoch in range(num_epochs):
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()

        # 梯度裁剪
        utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

    optimizer.step()
```

3. 数据问题

训练数据中的异常值或 NaN 值可能导致损失值变为 NaN。

解决方案：

- **数据预处理**：对数据进行清洗，去除异常值和 NaN 值。
- **数据标准化**：对输入数据进行标准化处理，使其均值为 0，方差为 1。

```
import numpy as np

# 假设 input_data 和 target_data 是 numpy 数组
input_data = np.nan_to_num(input_data) # 将 NaN 值替换为 0
target_data = np.nan_to_num(target_data) # 将 NaN 值替换为 0
```

4. 模型初始化问题

模型的初始权重可能不合适，导致训练过程中出现 NaN。

解决方案：

- **权重初始化**：使用合适的权重初始化方法，如 Xavier 或 He 初始化。

```
import torch.nn.init as init

def init_weights(m):
    if isinstance(m, nn.Linear):
        init.xavier_uniform_(m.weight)
        if m.bias is not None:
            init.zeros_(m.bias)

model.apply(init_weights)
```

5. 数值稳定性问题

某些操作（如除以非常小的数或对非常小的数取对数）可能导致数值稳定性问题，从而产生 NaN。

解决方案：

- **数值稳定性**：在计算中添加一个很小的常数（如 1e-8）来避免除以零或取对数时出现问题。

```
# 例如，在计算对数时
epsilon = 1e-8
log_value = torch.log(value + epsilon)
```

6. 激活函数问题

某些激活函数（如 ReLU）可能导致梯度消失或梯度爆炸，从而产生 `Nan`。

解决方案：

- **选择合适的激活函数**：使用 LeakyReLU 或 PReLU 等变体来替代 ReLU，以避免梯度消失问题。

```
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(10, 10),
    nn.LeakyReLU(),
    nn.Linear(10, 1)
)
```

7. 正则化问题

过强的正则化（如 `weight_decay` 过大）可能导致损失值变为 `Nan`。

解决方案：

- **调整正则化强度**：适当减小 `weight_decay` 的值。

```
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=0.001)
```

8. 检查损失函数

损失函数的实现可能有误，导致计算出的损失值为 `Nan`。

解决方案：

- **检查损失函数**：确保损失函数的实现正确，没有除以零或取对数时出现问题。

9. 固定随机种子

训练过程中的随机性可能导致 `Nan` 的出现。

解决方案：

- **固定随机种子**：在训练开始时固定随机种子，以确保实验的可重复性。

```
import random
import numpy as np
import torch

seed = 42
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
```

总结

`Nan` 值的出现通常是多方面因素共同作用的结果。通过调整学习率、梯度裁剪、数据预处理、模型初始化、数值稳定性、激活函数、正则化强度等方法，可以有效地解决 `Nan` 问题。

241117

1. 发现mse_all的对比对象有问题

修正后效果还是不好，只对填充后的值做MSE，未对填充后的值进行汇聚（多0值和全满矩阵之间的差距还是太大，需要进一步变化）

2. `enumerate` 在遍历列表、元组或其他可迭代对象时，同时获取元素及其对应的索引。这对于需要在循环中同时处理索引和值的情况非常方便。

3. 权重参数在模型中如何体现self.W? ? ?

在数学公式中，权重参数通常表示模型中的可学习参数，这些参数在实际代码中是通过神经网络层中的权重矩阵来实现的。在 PyTorch 中，这些权重参数通常是在定义模型时自动初始化的，并在训练过程中通过反向传播和优化算法进行更新。

数学公式与代码的对应关系

假设我们有一个简单的线性模型 ($y = Wx + b$)，其中 (W) 是权重矩阵，(x) 是输入向量，(b) 是偏置项，(y) 是输出向量。在数学公式中，(W) 和 (b) 是显式表示的参数。

在 PyTorch 中，这个模型可以定义如下：

```
import torch
import torch.nn as nn

class LinearModel(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(LinearModel, self).__init__()
        self.linear = nn.Linear(input_dim, output_dim)

    def forward(self, x):
        return self.linear(x)

# 创建模型实例
input_dim = 10
output_dim = 1
model = LinearModel(input_dim, output_dim)

# 查看模型参数
for name, param in model.named_parameters():
    print(name, param)
```

详细解释

1. 定义模型：

- `nn.Linear(input_dim, output_dim)` 创建一个线性层，其中 `input_dim` 是输入特征的维度，`output_dim` 是输出特征的维度。
- 这个线性层内部自动初始化了权重矩阵 (W) 和偏置项 (b)。

2. 前向传播：

- `self.linear(x)` 在前向传播过程中计算 ($y = Wx + b$)。

3. 查看模型参数：

- `model.named_parameters()` 返回模型中所有可学习参数的名称和值。
- 通常，线性层的权重矩阵会被命名为 `linear.weight`，偏置项会被命名为 `linear.bias`。

例子：注意力机制

假设我们有一个简单的注意力机制，其数学公式如下：

$$\text{attention}(q, k, v) = \text{softmax}\left(\frac{qk^T}{\sqrt{d_k}}\right)v$$

其中 (q) 是查询向量，(k) 是键向量，(v) 是值向量，(d_k) 是键向量的维度。

在 PyTorch 中，这个注意力机制可以实现如下：

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class AttentionLayer(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(AttentionLayer, self).__init__()
        self.w_q = nn.Linear(input_dim, hidden_dim, bias=False)
        self.w_k = nn.Linear(input_dim, hidden_dim, bias=False)
        self.w_v = nn.Linear(input_dim, hidden_dim, bias=False)

    def forward(self, query, key, value):
        Q = self.w_q(query)
        K = self.w_k(key)
        V = self.w_v(value)

        scores = torch.matmul(Q, K.transpose(-2, -1)) / 
torch.sqrt(torch.tensor(K.size(-1), dtype=torch.float32))
        attention_weights = F.softmax(scores, dim=-1)
        weighted_sum = torch.matmul(attention_weights, V)

        return weighted_sum

# 创建模型实例
input_dim = 10
hidden_dim = 10
attention_layer = AttentionLayer(input_dim, hidden_dim)

# 创建输入张量
query = torch.randn(1, 10)
key = torch.randn(5, 10)
value = torch.randn(5, 10)

# 前向传播
output = attention_layer(query, key, value)

print("输出张量:", output)
print("输出张量形状:", output.shape)
```

详细解释

1. 定义注意力层:

- `self.w_q`、`self.w_k` 和 `self.w_v` 是三个线性层，分别用于计算查询向量 (q)、键向量 (k) 和值向量 (v) 的变换。
- 这些线性层内部自动初始化了权重矩阵。

2. 前向传播:

- `Q = self.w_q(query)` 计算查询向量的变换。
- `K = self.w_k(key)` 计算键向量的变换。
- `V = self.w_v(value)` 计算值向量的变换。
- `scores = torch.matmul(Q, K.transpose(-2, -1)) / torch.sqrt(torch.tensor(K.size(-1), dtype=torch.float32))` 计算注意力分数。
- `attention_weights = F.softmax(scores, dim=-1)` 计算注意力权重。
- `weighted_sum = torch.matmul(attention_weights, V)` 计算加权求和的值向量。

3. 查看模型参数:

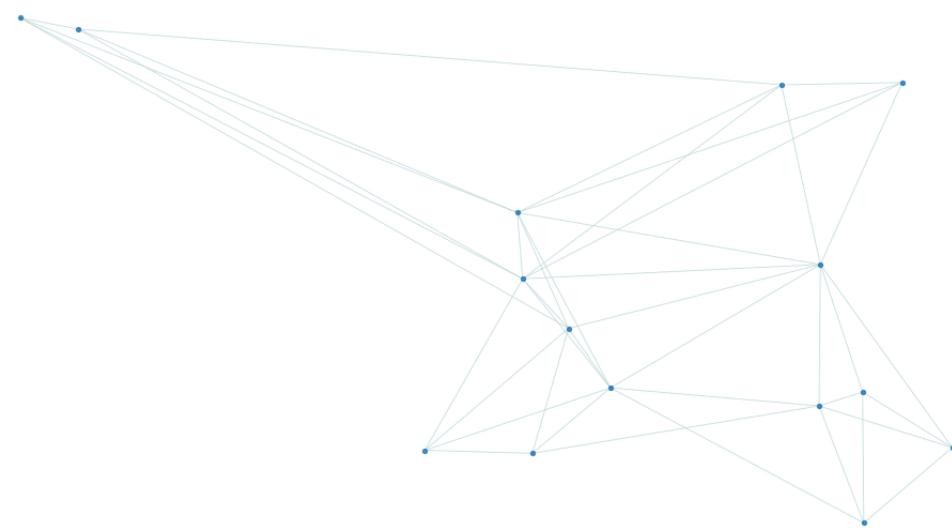
- `attention_layer.named_parameters()` 返回注意力层中所有可学习参数的名称和值。

总结

- **数学公式中的权重参数**: 在数学公式中，权重参数是显式表示的。
- **代码中的权重参数**: 在 PyTorch 中，这些权重参数通过神经网络层 (如 `nn.Linear`) 自动初始化，并在训练过程中通过反向传播和优化算法进行更新。
- **对应关系**: 数学公式中的权重参数在代码中通过神经网络层的权重矩阵来实现。

4. `torch.cat` 是 PyTorch 中的一个函数，用于在指定维度上拼接多个张量。通过设置 `dim` 参数，您可以控制在哪个维度上进行拼接。`dim` 参数指定了拼接操作的轴或维度。

5.



每个节点的邻居个数不同

邻居节点的特征汇聚，维度如何设定，根据哪个维度进行汇聚，应该2000汇聚，那么怎么处理多个邻居

241115

1. 沿途小师

和人工智能长期对话，会感觉他的语料缺乏创新性，或者说像是函数进入平稳期，会翻来覆去重复上下文，导致逐渐混乱、出错，需要重开一个对话。

对话的创新性和可靠性最高点存在于对话过程中的某一个阶段，比如一开始，新颖，但是不够懂我们（我们提供的语料不够），几个轮回后充分理解我们的意思，这个阶段最准确，但是再往后，就翻来覆去缺乏创新甚至逻辑有误了。

显示 latex 公式、代码不稳定

不能够复制 latex 公式

241114

计划, 梳理

1. 计算残差图 (G-n) 的鲁棒性度量

2.

241113 性能指标选点结果

计划

1. 梳理一下公式，和弹性训练的公式

2. 画线界定一下区间，做一下后面弹性的训练

3. 再回来处理性能指标的问题

1. 在 LaTeX 中表示乘法有几种常见的方法，取决于你希望乘法符号如何显示。以下是几种常用的方式：

1. 使用星号(*):

- 这是最简单的形式，但在数学公式中通常不推荐使用，因为它看起来像编程语言中的乘法操作符。
- 示例代码：`$a * b$`
- 显示结果：(a * b)

2. 使用点乘(\cdot):

- 这是数学文献中最常用的乘法表示方式。
- 示例代码：`$a \cdot b$`
- 显示结果：(a \cdot b)

3. 使用叉乘(\times):

- 当需要明确表示向量或矩阵的乘法时，通常使用叉乘。
- 示例代码：`$a \times b$`
- 显示结果：(a \times b)

4. 直接相邻：

- 在某些情况下，特别是在代数表达式中，两个变量或数字直接相邻就表示乘法。
- 示例代码：`ab`
- 显示结果：(ab)

5. 使用括号：

- 有时为了清晰起见，会用括号来表示乘法。
- 示例代码：`$(a)(b)$`
- 显示结果：((a)(b))

选择哪种方式取决于具体的上下文和个人偏好。在正式的数学文档中，使用 `\cdot` 或者直接相邻的形式最为常见。

2. 计算两两之间的距离：

- 使用 `pdist` 函数计算每对点之间的距离，得到一个一维数组。
- `pdist` 函数默认使用欧几里得距离（Euclidean distance），但也可以指定其他距离度量。

3. 将一维数组转换为二维矩阵：

- 使用 `squareform` 函数将 `pdist` 返回的一维数组转换为一个 $n \times n$ 的对称矩阵。

4. ◦ `min`：返回的是最小值本身。

- `argmin`：返回的是使函数取得最小值的输入参数。

1. `min` (最小值)

`min` 表示一个集合或函数的最小值。具体来说，`min` 返回的是最小的那个值本身。

2. `argmin` (最小值点)

`argmin` 表示使某个函数取得最小值的输入参数。具体来说，`argmin` 返回的是使函数值最小的那个输入值或输入值的集合。

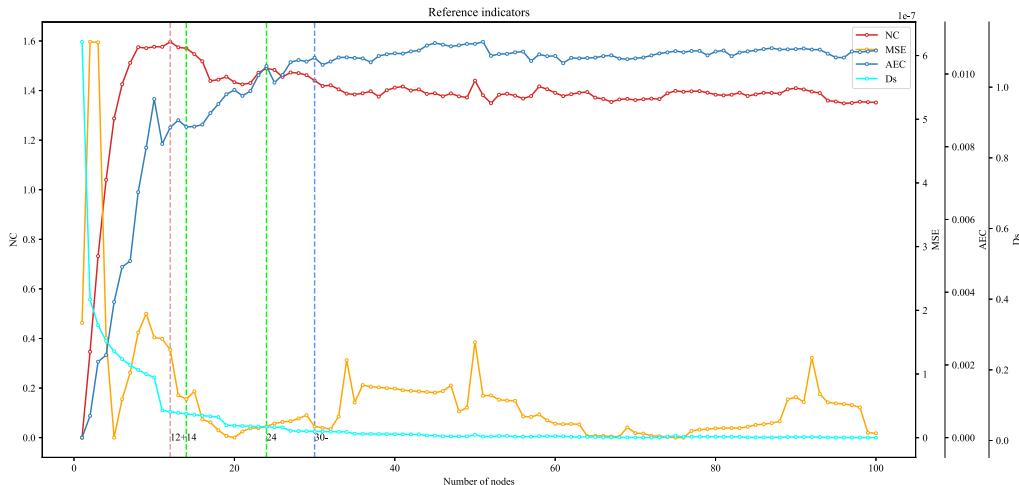
对于一个函数 ($f(x)$)，`argmin` 表示使函数取得最小值的输入参数：

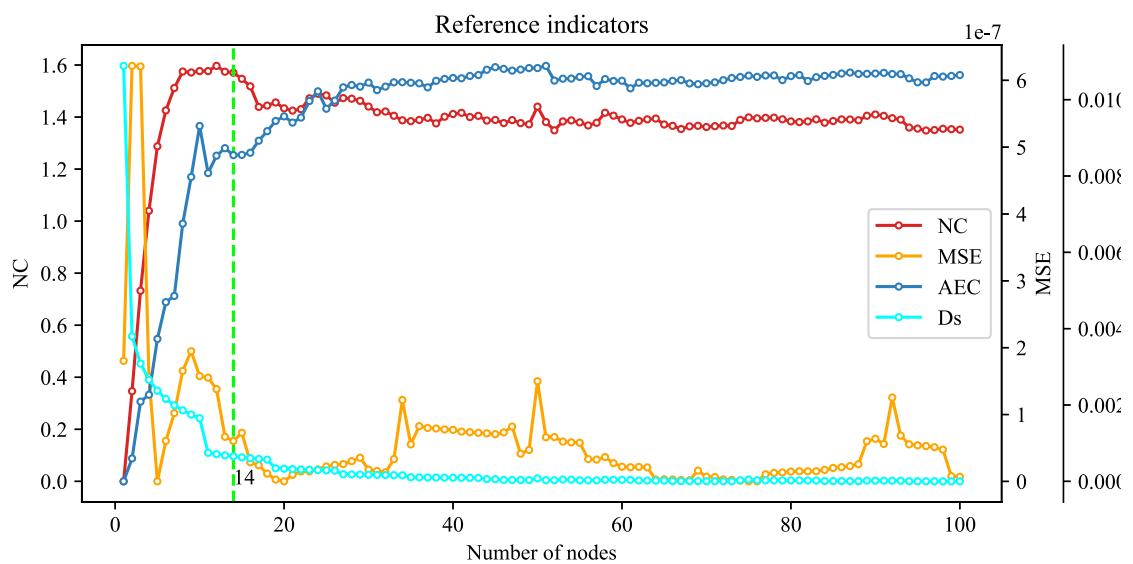
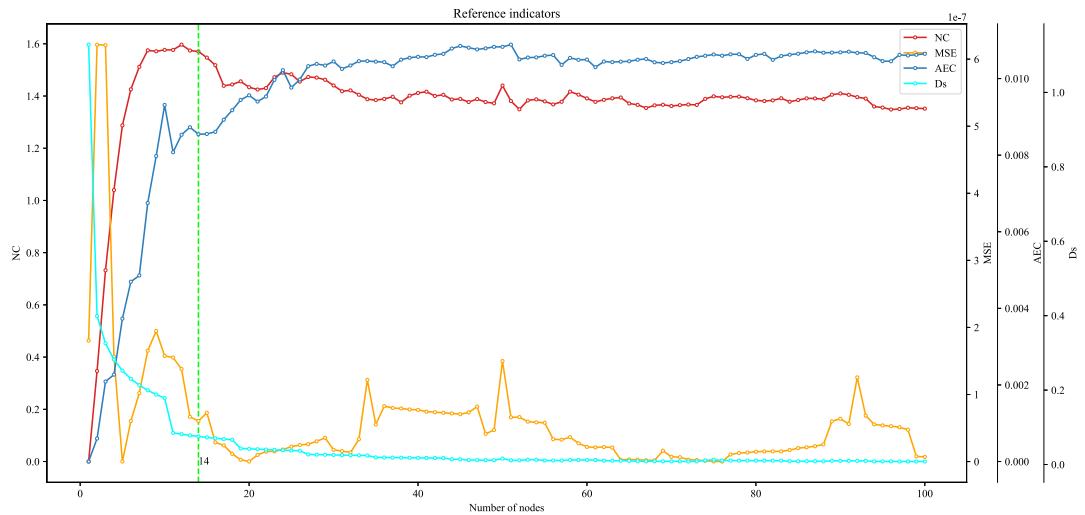
$$\arg \min_{x \in D} f(x)$$

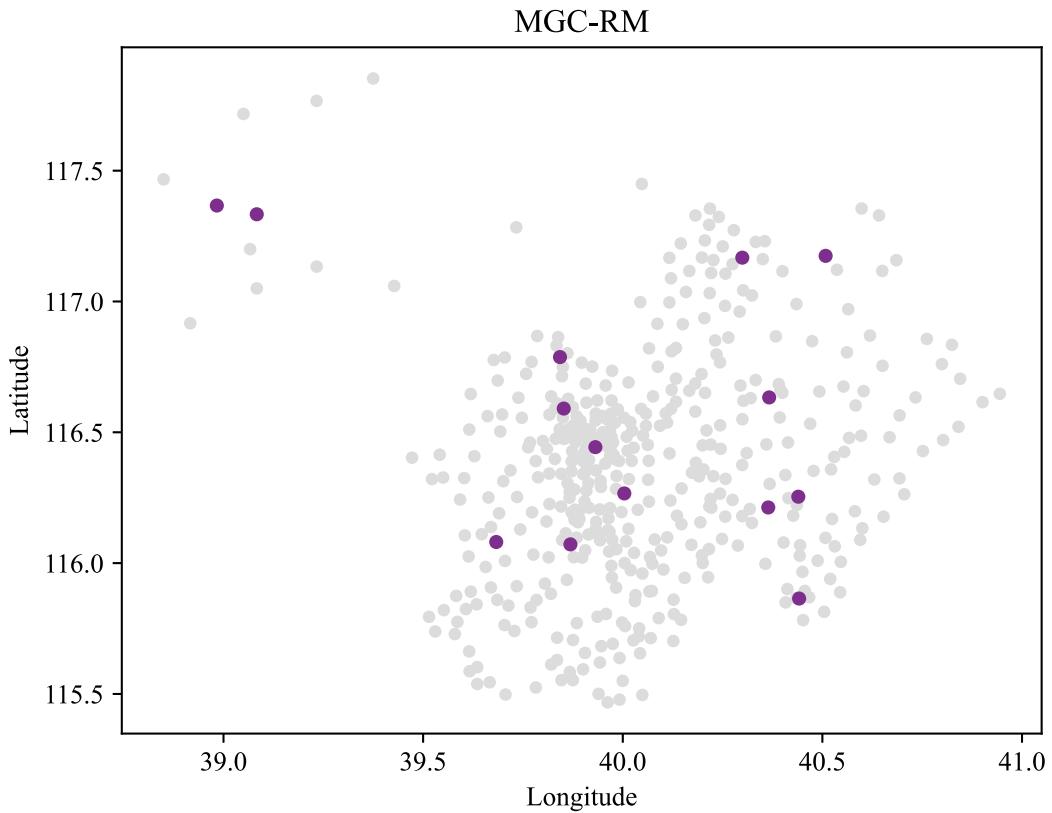
其中 (D) 是函数 ($f(x)$) 的定义域。

5. 剩余平均能量逻辑有误

完全重构AEC,效果还凑合







感觉有点夸张，减少的是不是太小了点，没有测试图级的特征

241112

1. readout

- `self.weight`: 一个可学习的权重矩阵，形状为 `[output_size, output_size]`。这个权重矩阵用于转换全局特征。
- `nn.init.xavier_uniform_`: 使用 Xavier 均匀分布初始化权重矩阵，这是一种常用的权重初始化方法，有助于加速训练过程。
- **输入**:
 - `h`: 节点特征矩阵，形状为 `[num_nodes, feature_dim]`，其中 `num_nodes` 是节点的数量，`feature_dim` 是每个节点的特征维度。
- **步骤解析**:

1. 计算全局特征的初始值:

```
global_h = torch.matmul(torch.mean(h, dim=0), self.weight)
```

- `torch.mean(h, dim=0)`: 计算所有节点特征的平均值，结果是一个形状为 `[feature_dim]` 的向量。
- `torch.matmul(..., self.weight)`: 将平均特征与权重矩阵相乘，得到一个形状为 `[output_size]` 的向量 `global_h`。

2. 转换全局特征:

```
transformed_global = torch.tanh(global_h)
```

- `torch.tanh`：使用双曲正切函数对 `global_h` 进行非线性变换，结果仍然是一个形状为 `[output_size]` 的向量 `transformed_global`。

3. 计算注意力得分：

```
sigmoid_scores = torch.sigmoid(torch.mm(h, transformed_global.view(-1, 1)).sum(dim=1))
```

- `transformed_global.view(-1, 1)`：将 `transformed_global` 转换为形状为 `[output_size, 1]` 的列向量。
- `torch.mm(h, ...)`：将节点特征矩阵 `h` 与 `transformed_global` 相乘，结果是一个形状为 `[num_nodes, 1]` 的矩阵。
 - `.sum(dim=1)`：对每个节点的乘积结果求和，得到一个形状为 `[num_nodes]` 的向量。
 - `torch.sigmoid(...)`：使用 Sigmoid 函数将每个节点的得分转换为 `[0, 1]` 范围内的注意力权重 `sigmoid_scores`。

4. 加权节点特征：

```
h_global = sigmoid_scores.unsqueeze(-1) * h
```

- `sigmoid_scores.unsqueeze(-1)`：将 `sigmoid_scores` 转换为形状为 `[num_nodes, 1]` 的列向量。
- `* h`：将注意力权重与节点特征矩阵相乘，结果是一个形状为 `[num_nodes, feature_dim]` 的矩阵 `h_global`，其中每个节点的特征都被相应的注意力权重加权。

5. 生成图级别的特征：

```
return h_global.sum(dim=0).unsqueeze(0)
```

- `h_global.sum(dim=0)`：对所有节点的加权特征进行求和，结果是一个形状为 `[feature_dim]` 的向量。
 - `.unsqueeze(0)`：在第0维增加一个新的维度，将结果转换为形状为 `[1, feature_dim]` 的张量，表示图级别的特征。
- 使用了 `zip()` 函数来将这两个列表的对应元素配对，然后在每次循环中将这些配对的元素分别赋值给变量 `low_dim_vector` 和 `node_list`
 - `enumerate(node_list)` 是一个非常有用的Python内置函数，它允许你在遍历列表（或其他可迭代对象）的同时获取每个元素的索引。这对于需要同时使用元素及其位置的情况特别有用。
 - 统一不同维度图的特征：全图节点特征构成图的特征，根据节点索引填充0同一纬度，在进行汇聚，但是效果不明显。

在处理不同维度的向量时，直接计算均方误差（MSE）可能会遇到维度不匹配的问题。为了解决这个问题，可以采取以下几种方法之一：

1. **Padding (填充)**：将所有向量填充到相同的维度。
2. **Truncation (截断)**：将所有向量截断到相同的维度。
3. **Projection (投影)**：将所有向量投影到一个共同的低维空间。
4. **Weighted MSE (加权MSE)**：根据向量的实际长度调整MSE的权重。

方法 1: Padding (填充)

将所有向量填充到相同的最大维度，通常用零填充。

```
import torch
import torch.nn.functional as F

def pad_tensors(tensors, max_len):
    padded_tensors = []
    for tensor in tensors:
        padded_tensor = F.pad(tensor, (0, max_len - tensor.size(0)),
'constant', 0)
        padded_tensors.append(padded_tensor)
    return torch.stack(padded_tensors)

def mse_with_padding(pred, target):
    max_len = max(pred.size(0), target.size(0))
    pred_padded = pad_tensors([pred], max_len)[0]
    target_padded = pad_tensors([target], max_len)[0]
    return F.mse_loss(pred_padded, target_padded)

# 示例
pred = torch.tensor([1.0, 2.0, 3.0])
target = torch.tensor([1.0, 2.0, 3.0, 4.0])
mse = mse_with_padding(pred, target)
print(f'MSE with padding: {mse.item()}')
```

方法 2: Truncation (截断)

将所有向量截断到相同的最小维度。

```
def truncate_tensors(tensors, min_len):
    truncated_tensors = []
    for tensor in tensors:
        truncated_tensor = tensor[:min_len]
        truncated_tensors.append(truncated_tensor)
    return torch.stack(truncated_tensors)

def mse_with_truncation(pred, target):
    min_len = min(pred.size(0), target.size(0))
    pred_truncated = truncate_tensors([pred], min_len)[0]
    target_truncated = truncate_tensors([target], min_len)[0]
    return F.mse_loss(pred_truncated, target_truncated)

# 示例
pred = torch.tensor([1.0, 2.0, 3.0, 4.0])
target = torch.tensor([1.0, 2.0, 3.0])
mse = mse_with_truncation(pred, target)
print(f'MSE with truncation: {mse.item()}')
```

方法 3: Projection (投影)

将所有向量投影到一个共同的低维空间。

```
import torch
import torch.nn as nn

class ProjectionLayer(nn.Module):
    def __init__(self, input_size, output_size):
        super(ProjectionLayer, self).__init__()
        self.fc = nn.Linear(input_size, output_size)

    def forward(self, x):
        return self.fc(x)

def mse_with_projection(pred, target, proj_layer):
    pred_proj = proj_layer(pred.unsqueeze(0)).squeeze(0)
    target_proj = proj_layer(target.unsqueeze(0)).squeeze(0)
    return F.mse_loss(pred_proj, target_proj)

# 示例
input_size = 4
output_size = 3
proj_layer = ProjectionLayer(input_size, output_size)

pred = torch.tensor([1.0, 2.0, 3.0, 4.0])
target = torch.tensor([1.0, 2.0, 3.0])
mse = mse_with_projection(pred, target, proj_layer)
print(f'MSE with projection: {mse.item()}')
```

方法 4: Weighted MSE (加权MSE)

根据向量的实际长度调整MSE的权重。

```
def weighted_mse(pred, target):
    min_len = min(pred.size(0), target.size(0))
    pred_truncated = pred[:min_len]
    target_truncated = target[:min_len]
    mse = F.mse_loss(pred_truncated, target_truncated)

    # 计算剩余部分的MSE
    if pred.size(0) > min_len:
        mse += F.mse_loss(pred[min_len:], torch.zeros(pred.size(0) - min_len))
    if target.size(0) > min_len:
        mse += F.mse_loss(target[min_len:], torch.zeros(target.size(0) - min_len))

    return mse

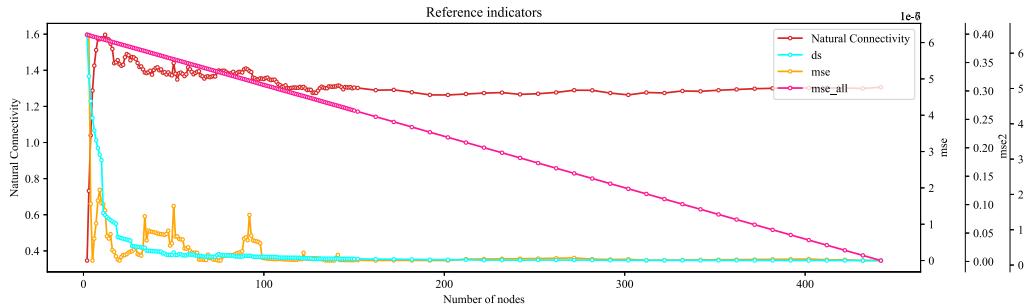
# 示例
pred = torch.tensor([1.0, 2.0, 3.0, 4.0])
target = torch.tensor([1.0, 2.0, 3.0])
mse = weighted_mse(pred, target)
print(f'Weighted MSE: {mse.item()}')
```

总结

以上方法都可以解决不同维度向量之间的MSE计算问题。选择哪种方法取决于具体的应用场景和数据特性：

- **Padding**: 适用于需要保留所有信息的情况。
- **Truncation**: 适用于只需要比较共同部分的情况。
- **Projection**: 适用于需要将高维数据降维的情况。
- **Weighted MSE**: 适用于需要平衡不同长度向量的情况。

5. mse all , 填充0进行全图比较, 太 , , , ,



还是得汇聚图级特征，或者自编码器

但是还是得重新训练

6. 有没有一种可能性就是，不可能真的只用2、3个传感器，所以前几个传感器的值很奇怪不用很纠结

7. 画图随时记录

241111

1. Ds尝试了很多种算法

Ds: 原始算法，没有细看f_distance和g_distance的原理，算法太慢了跑不出来

Ds2: 修改算法对Rst和rmax的计算，未考虑物理距离

Ds3: 考虑物理距离，尝试了考虑连接的点数和没有考虑连接点数，区别不大

先这样吧，有一个大概的趋势

2. 考虑汇总成图级特征进行比较：MSE MI

1. MSE:

提炼一个图级特征

加性注意力机制

加性注意力机制 (*Additive Attention Mechanism*)，也被称为Bahdanau注意力或基于内容的注意力模型，是深度学习中一种重要的机制，用于在**处理序列数据时给予不同部分不同的权重**。这种机制最初是在神经机器翻译任务中提出并应用的，由Dzmitry Bahdanau等人在2015年的论文《Neural Machine Translation by Jointly Learning to Align and Translate》中介绍。

工作原理

加性注意力机制的核心思想是通过计算查询向量 (query vector) 与每个键向量 (key vector) 之间的相似度得分来决定给定序列中每个元素的重要性。这里的查询向量可以理解为解码器的状态，而键向量则是编码器产生的每个时间步的状态。具体来说，该机制通过一个前馈神经网络来计算这些得分，这个过程可以形式化为以下步骤：

- 计算得分**: 对于解码器状态(s_t)和编码器每个时间步的状态(h_i), 使用一个前馈神经网络(通常是一个单层的全连接网络)来计算得分(e_{ij})。这个得分反映了(s_t)对(h_i)的关注程度。数学上, 可以表示为:

$$e_{ij} = v^T \tanh(W_s s_t + W_h h_i)$$
 其中, (v)、(W_s)和(W_h)是可学习的参数, (\tanh)是非线性激活函数。
- 归一化得分**: 将得分转换成概率分布, 这一步通常是通过softmax函数实现的, 以确保所有得分加起来等于1。这样就得到了注意力权重(a_{ij}):

$$a_{ij} = \frac{\exp(e_{ij})}{\sum_k \exp(e_{ik})}$$
- 计算上下文向量**: 根据上述得到的注意力权重, 对编码器的所有隐藏状态进行加权求和, 从而获得上下文向量(c_t):

$$c_t = \sum_i a_{ij} h_i$$
- 使用上下文向量**: 最后, 将上下文向量(c_t)与解码器的状态结合, 用于生成下一个时间步的输出。

应用场景

加性注意力机制广泛应用于自然语言处理领域, 包括但不限于:

- 神经机器翻译**: 帮助模型更好地理解源语言句子中的各个部分, 并准确地将其翻译成目标语言。
- 文本摘要**: 从长篇文章中提取关键信息, 生成简洁的摘要。
- 问答系统**: 提高模型对问题的理解能力, 从而提供更准确的答案。

加性注意力机制通过允许模型动态地关注输入序列的不同部分, 极大地提高了模型处理复杂任务的能力。然而, 值得注意的是, 随着输入序列长度的增长, 计算注意力权重的成本也会相应增加, 这在实际应用中可能需要考虑优化策略来缓解这一问题。

多层感知机

多层感知机 (Multilayer Perceptron, MLP) 是一种前馈型人工神经网络模型, 它将多个神经元组织成多层结构, 包括一个输入层、一个或多个隐藏层以及一个输出层。每一层中的神经元都与下一层中的所有神经元相连, 但同一层内的神经元之间没有连接。MLP能够学习非线性函数映射, 因此在解决复杂的分类和回归问题上表现出色。

构成

- 输入层**: 接收原始数据输入。每个神经元对应输入特征的一个维度。
- 隐藏层**: 位于输入层和输出层之间, 可以有一个或多个。隐藏层中的每个神经元都会对来自前一层的输入进行**加权求和**, 并通过一个激活函数来产生输出。
- 输出层**: 根据任务需求设计, 例如在二分类问题中, 输出层可能只有一个神经元; 而在多分类问题中, 则可能有多个神经元, 每个神经元对应一个类别。

激活函数

激活函数用于引入非线性, 使得网络能够拟合复杂的函数关系。常见的激活函数包括:

- Sigmoid**: ($\sigma(x) = \frac{1}{1 + e^{-x}}$), 输出范围(0,1), 适合用于二分类问题的输出层。
- ReLU (Rectified Linear Unit)**: ($f(x) = \max(0, x)$), 广泛用于隐藏层, 因为它有助于缓解梯度消失问题。
- Tanh (Hyperbolic Tangent)**: ($\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$), 输出范围(-1,1), 也是一种常用的非线性激活函数。
- Softmax**: 常用于多分类问题的输出层, 可以将输出转化为概率分布。

训练过程

MLP的训练通常采用反向传播算法来最小化损失函数。这个过程涉及以下几个步骤：

1. **前向传播**: 从输入层开始, 数据通过每一层的神经元传递到输出层, 每经过一个神经元, 都会进行加权求和并通过激活函数。
2. **计算损失**: 在输出层, 根据预测值与真实值之间的差异计算损失函数的值。
3. **反向传播**: 从输出层向输入层方向, 逐层计算每个神经元对最终损失的影响 (即梯度), 并据此调整各层的权重和偏置。
4. **权重更新**: 利用梯度下降等优化算法, 根据计算出的梯度更新网络中的权重和偏置, 以减小损失函数的值。

应用

MLP广泛应用于各种领域, 包括但不限于:

- **图像识别**: 通过训练MLP模型来识别图片中的对象。
- **语音识别**: 将音频信号转换为文本。
- **自然语言处理**: 如情感分析、文本分类等任务。
- **推荐系统**: 根据用户的历史行为预测其可能感兴趣的的商品或内容。

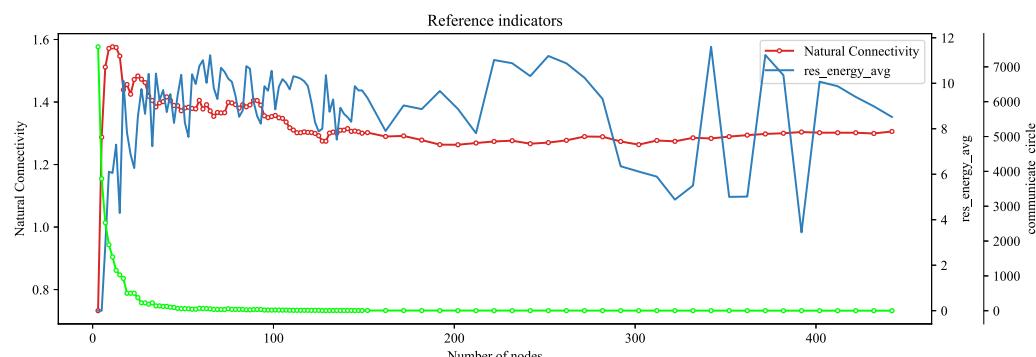
总之, 多层感知机作为一种基础且强大的神经网络模型, 在许多机器学习任务中都有着广泛的应用。

241110

1. 计算通信损失时没有考虑物理距离 (自然连通性也没有考虑物理距离), 考虑物理距离赋值为边的权重

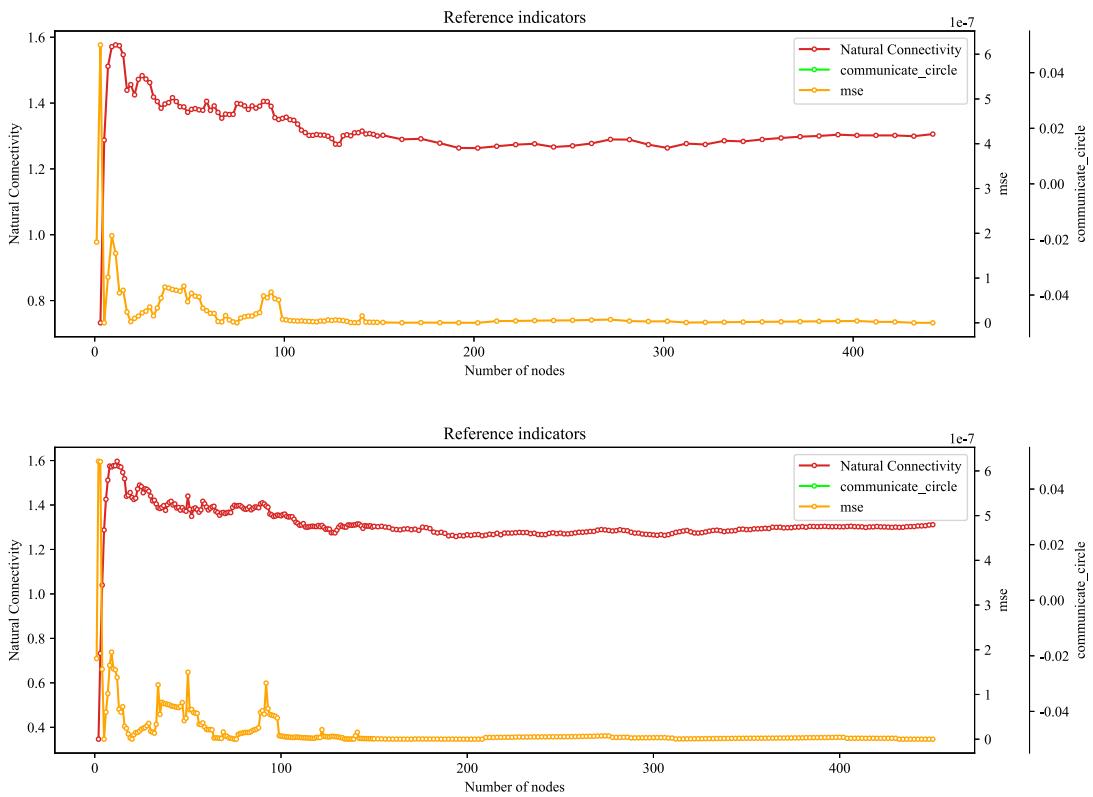
假设暂时不考虑物理距离

2. 网络生命会快速下降



剩余能量这个评估指标不好, 换一个

3. 图核
4. 图嵌入
5. 得到的一个单一节点的特征表示对比



6. Ds的max_radius 没有考虑物理距离

最新算法重新计算了 r_{max} , 相当于重新构图, 没有按照原来的方法构图,

计划:

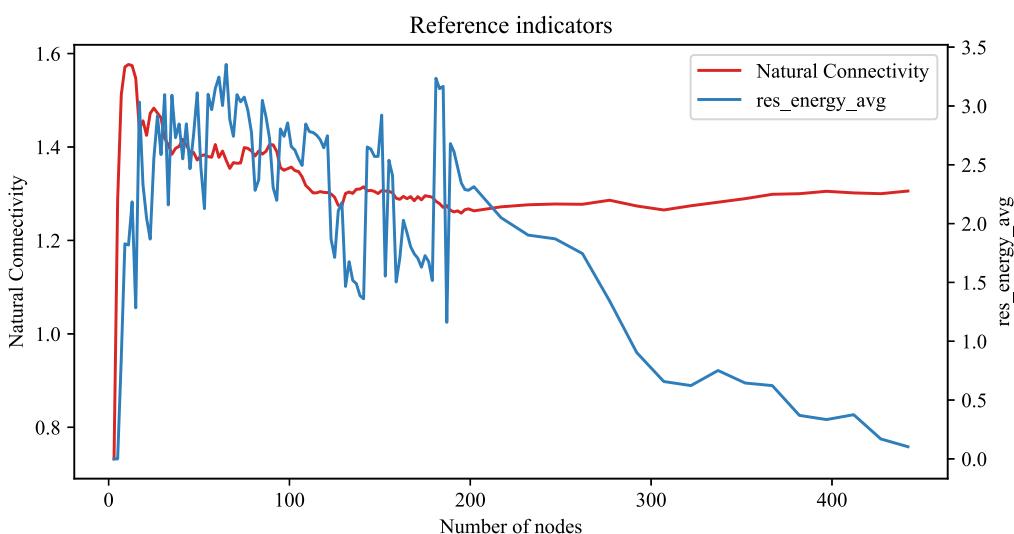
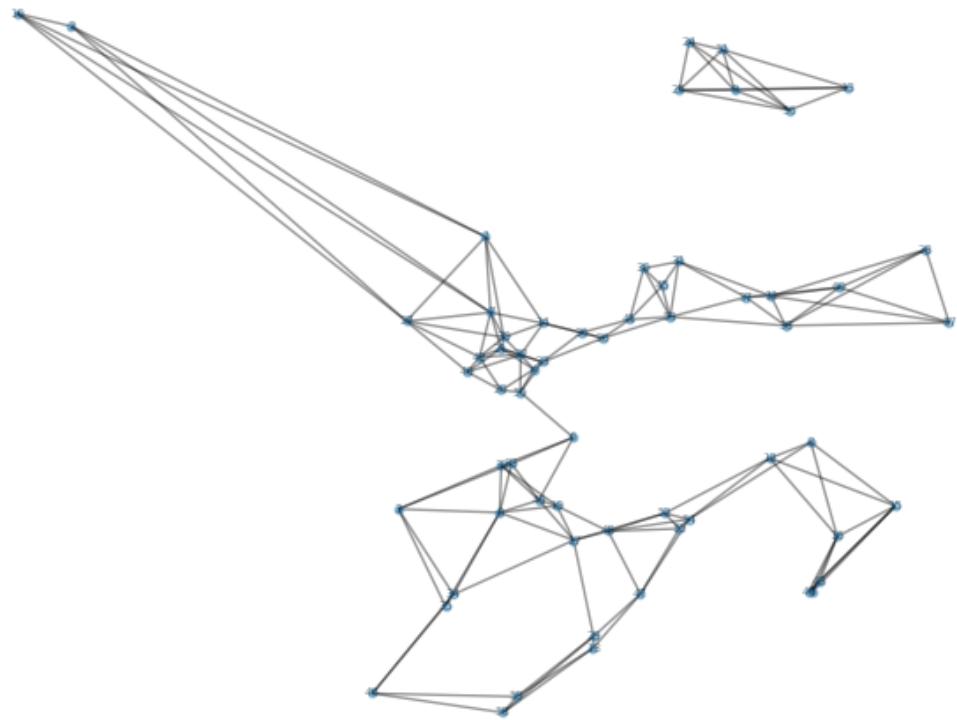
1. 考虑汇总成图级特征进行比较: MSE MI
2. 考虑一些弹性方向的度量

241106

1. 指标的选择: 多个不同维度的图进行比较

1. 节点特征聚合为图特征进行衡量: 图特征聚合器训练
2. 比较过程中是否固定图的维度。 (选出的节点重新构图/未选节点从原图中移除边、掩蔽特征)

1. 使用 NetworkX 库计算图的半径时, 如果图不是连通的, `nx.radius` 函数会抛出 `NetworkXError`, 因为半径的定义要求图必须是连通的。对于非连通图, 你需要分别计算每个连通子图的半径。



2. 平均剩余能量是什么含义,代码的含义

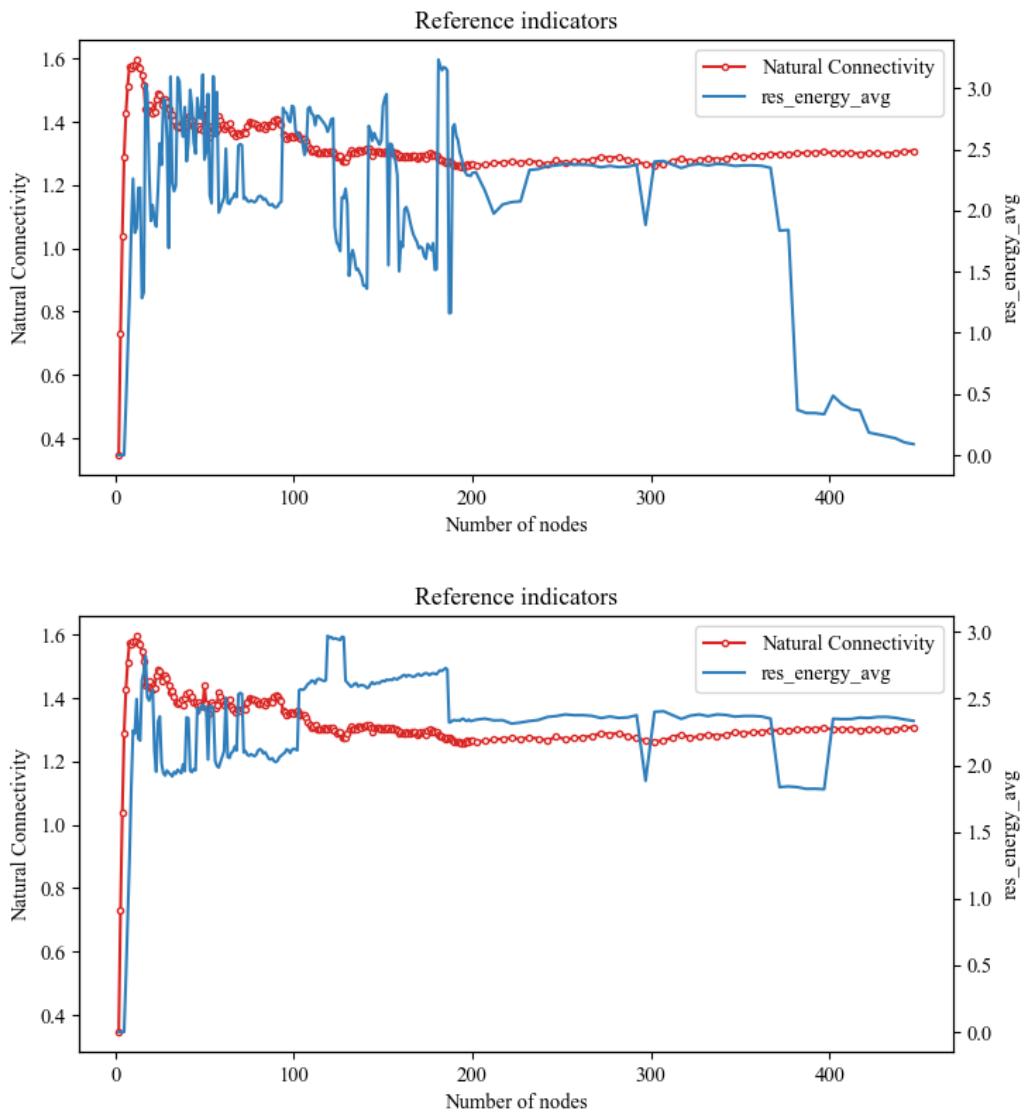
1. **通信能耗模型**: 定义了 `communication_energy_loss` 函数, 用于计算单个节点与它的邻居进行一轮通信后的能量损失。这个函数考虑了两个主要因素:

- `E_elec`: 电子设备消耗的能量。
- `beta * dis ** alpha`: 信号传输的能量损耗, 其中 `dis` 是两节点之间的距离, `alpha` 和 `beta` 是与传播介质相关的系数。

2. **网络生命周期计算**: 在 `network_life` 函数中, 通过循环模拟多轮通信过程。对于每一轮, 每个节点都会与其所有在通信范围内的邻居进行通信 (那图上的连接关系是什么含义? ? ? 连接如果是通信的含义, 那就不应该是图半径内的点通信, 而是相连的点通信), 并根据 `communication_energy_loss` 函数更新其能量值。当任意节点的能量降至0或以下时, 该轮次结束, 网络被认为失效。此时, 计算剩余节点的平均能量值, 并返回网络的总生存轮次和剩余平均能量。

这个值主要和传感器数量和位置有关吧 (在不在“通信范围内”)

改变通信范围后图像变化了



计划:

1. 平均剩余能量,
 1. 非连通的图半径能那样求吗
2. MSE等

241105

1. (1) **均方误差 (MSE):** MSE 用于比较选择出的节点经重构后的特征和原始特征之间的差异。MSE 越低，说明通过所选节点重构原始信息的能力越强。

$$MSE = \frac{1}{N} \sum_{i=1}^N (x_i - \hat{x}_i)^2 \quad (3-41)$$

其中， x_i 为节点 v_i 的原始特征， \hat{x}_i 为重构特征。

MSE 衡量某个节点的预测特征与原始特征的均方误差？

怎么用来评估整个图？

对于全图而言的特征是？？拓扑特征？节点特征？

什么意思，选出来的点通过特征预测后的特征与原特征的差距？

数据重建

2. (2) 平均能量消耗 (Average Energy Consumption, AEC): AEC 度量了传感网的总体能量消耗。AEC 越低, 说明网络中的能量效率越高, 有利于网络的可持续性。

$$AEC = \frac{1}{N} \sum_{i=1}^N \left(E_0 - L \sum_{N_i} E_T - L \sum_{N_i} E_R \right) \quad (3-42)$$

其中, E_0 为传感器节点的初始能量, L 为网络寿命, E_T 为发送能耗, E_R 为接收能耗。根据无线传感网的无线传输能耗模型 (Radio Energy Dissipation Model, REDM) [93], 每个传感器节点在单次通信中的发送能耗和接收能耗分别为:

$$E_T = k(E_{elec} + \beta d^\alpha) \quad (3-43)$$

$$E_R = kE_{elec} \quad (3-44)$$

其中, E_{elec} 为单位信息的电子损耗, k 为信息量, β 为单位信息的功放损失, α 为传播衰减指数, d 为通信距离。

3. 评价指标在第三章的三个指标的基础上, 本章从信息角度增加两个指标来评估原始网络的特征和优化后网络的特征所包含语义信息的差异: 互信息 (Mutual Information, MI) 和 KL 散度。MI 用于衡量变量之间共享信息的数量, 而 KL 则衡量概率分布之间的相似性。较高的 MI 和较低的 KL 表明优化方法在过程中有效地保留了相关信息。

$$MI = \sum_x \sum_{x'} p(X, X') \log \frac{p(X, X')}{p(X)p(X')} \quad (4-40)$$

$$KL = \sum_x p(X) \log \frac{p(X)}{p(X')} \quad (4-41)$$

其中, X 为原始网络特征, X' 为优化后网络特征, $p(\square)$ 表示特征分布密度。

4. `.twinx()` 方法用于在同一张图中绘制两个具有不同 y 轴刻度的曲线。`.twinx()` 方法会创建一个新的 Axes 对象, 共享同一个 x 轴, 但有两个不同的 y 轴。
5. `.get_label()` 方法用于获取由 `.plot()` 或其他绘图方法设置的图例标签。这些标签通常用于图例 (legend) 来标识图表中的不同数据系列。
6. `axhline()` 方法用于在图表中绘制一条水平线。这条水平线可以用来标记特定的 y 值, 例如平均值、阈值等。
7. `.legend()` 方法用于添加图例
8. 传感器覆盖面积作为衡量?
9. KL散度里的概率分布要怎么获得:
连续特征: 对于连续特征, 可以使用直方图或核密度估计 (KDE) 来估计概率分布。

嵌入向量：使用图嵌入方法（如 Node2Vec、GraphSAGE 等）生成节点的嵌入向量，可以计算嵌入向量的分布。

```
k1_divergence = entropy(degree_freq_G1, degree_freq_G2)
```

10. 自然连通性（Natural Connectivity）是一种衡量图的鲁棒性的指标，它反映了图在面对节点或边的删除时保持连通性的能力。自然连通性越高，图的鲁棒性越好。
11. `find_value_according_index_list(aim_list, index_list):` # 索引转换 从排序索引找对应节点

但是aim_list 是？ 节点特征， position

12. 图鲁棒性归纳学习器Inductive Learner for Graph Robustness (ILGR)
13. 需要把不同数量节点的图构建出来
14. 按照重要性排序删点及其边，但是维度不变？这样会影响比较吧，在一些孤立点上会有影响吧
15. 确保不会出现“list assignment index out of range”错误。特别是，`selected_node` 列表的初始化应该在读取文件并填充 `node_list` 之后进行，以确保 `selected_node` 有足够的空间来存储每个 `select_node` 的值。

计划

1. 指标的分析：

1. 自然连通性具体代表什么含义(√)
2. 还有哪些可用的指标，针对优化和鲁棒性

2. 指标的代码构建

3. 弹性部分的展开逻辑

241104

1. gat权值共享怎么体现

2. 马尔科夫链？

3. 1. `np.sum(A, axis=1):`

- 这个函数计算矩阵 AA 每一行的元素之和。结果是一个一维数组，其中每个元素对应于 AA 中相应行的元素总和。

2. `np.power(..., -1):`

- 这个函数将上一步得到的一维数组中的每个元素取倒数（即，对每个元素应用幂 -1）。如果某行的和为0，则该操作会导致一个除以零的错误，因此在实际使用时需要确保没有行和为0，或者适当地处理这种异常情况。

3. `np.diag(...):`

- 最后，`np.diag` 函数用于从上述步骤的结果创建一个对角矩阵。这意味着它会将输入的一维数组转换成一个二维数组，其中输入数组的元素位于矩阵的主对角线上，而所有其他位置的值都为0。

4. `np.dot(A, D)` 可以计算这两个矩阵的点积（矩阵乘法）。具体来说，这个操作将矩阵 A 与对角矩阵 D 相乘，其中 DD 的对角线元素是 AA 每一行元素之和的倒数。

5. `np.ones(N)` 是 NumPy 库中的一个函数，用于创建一个包含全1的数组。具体来说，这个函数接受一个整数 N 作为参数，并返回一个长度为 N 的一维数组，其中所有元素都是1。

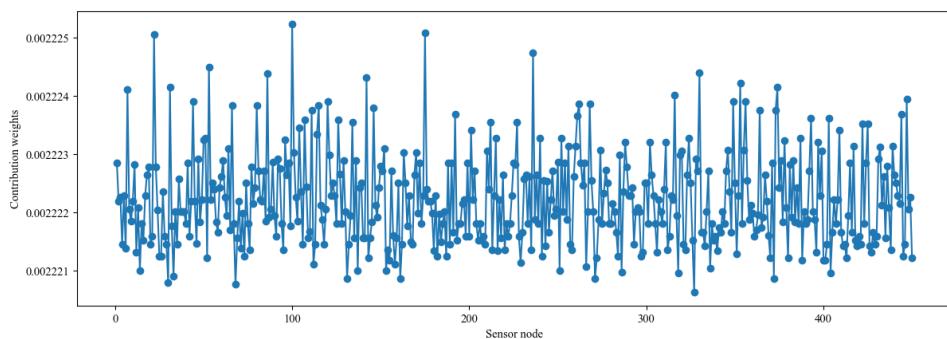
6. `np.argsort` 是 NumPy 库中的一个函数，用于返回数组元素排序后的索引。具体来说，它返回一个索引数组，这些索引按照原数组中元素的排序顺序排列。

4. 相似度分数的异常值不影响排序
5. 整理师兄的评估指标代码，绘制评估图的性能

241103

早上一来，有图没数据，机器好像重启了，果然不能寄希望于没有被看着的机子
好在我电脑上的数据保住了，我的宝贝电脑此刻价值得到了极大的发挥
服务器的GPU价值好还是很大的

1. 看起来prediction 误差很小，但是similarity score之间的误差也很小，这样的误差足以变换次序了
2. 预测误差为什么是一致的，在特定点上有一致的误差
3. 328异常 1.464528177166357636e-04 (收集数据)
但是要用data[329]的数据修正data[328]
4. colorbar?
5. 预测损失的规律性，与重要性得分之间存在什么关联？似乎是因此而影响的



计划

1. 细看论文，解析代码

prediction loss 的分布情况，与节点的重要性相关：更为重要的节点在全部节点中占少数，模型预测对这些节点或其邻居节点表现出更大的误差；重要性较低的节点在全部节点中占比较大，在模型预测中表现出较小的误差。

241102

GPU确实训练部分很快！！

但是模型评估部分运行特别慢

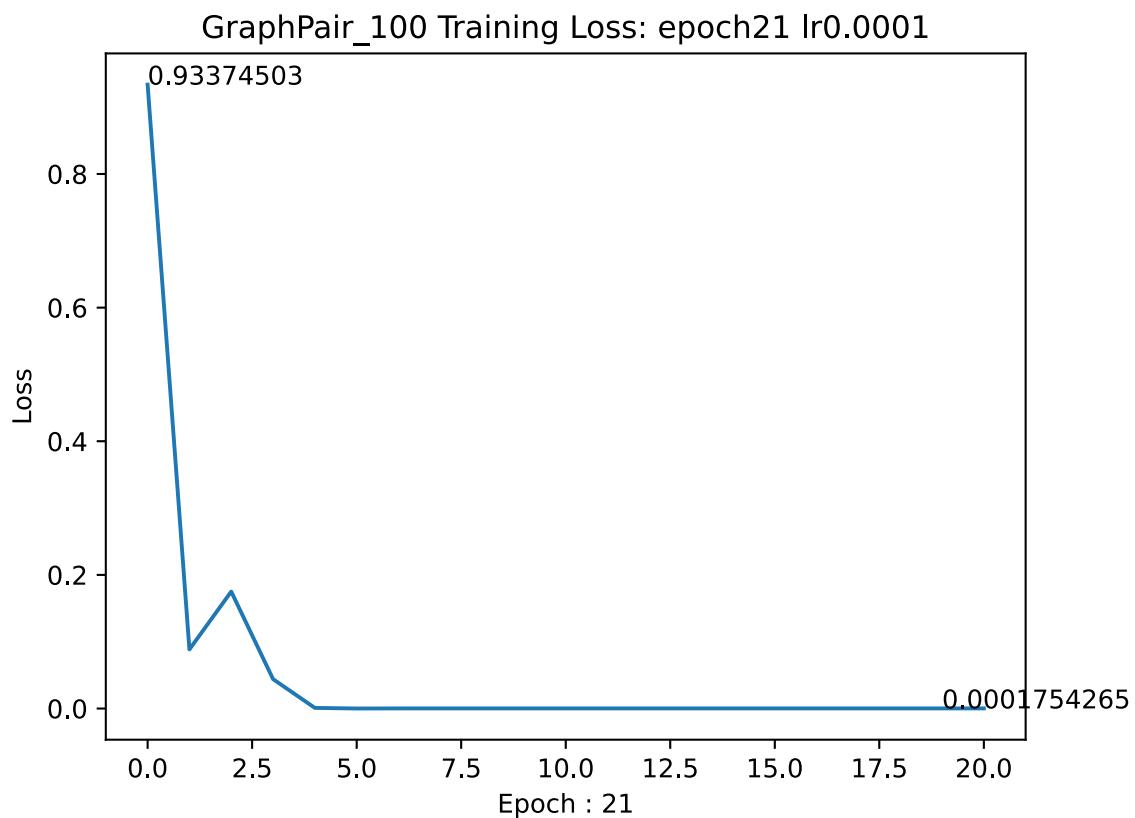
1. cuGraph缺少wheel，安装了Docker，但不知道方法是否正确，还缺少什么环境
2. 最慢的部分不是math.exp，是NX.graph_edit_distance, 但是没有找到NX在cuda的办法
3. **cuda的编程使用是个待研究的课题**
4. 每次获得全图的数据都很慢，一个小失误，一天白干。**注意小数据测试**

```
for i in range(perturbed_a.shape[0]-445):
```

5. 还是我的电脑跑的快一点
6. GPU同一个机器跑出来还会不同？**GPU是不是需要单独设置种子**，越跑越平滑了，但是预测分数依然总是1

prediction loss 较低的点会不会训练的模型更好? 230 75

prediction loss 较大的点拿来训练似乎会出现尖刺。但是每次会有不同。预测分数也总是1



```
epoch:17, loss:0.00017542651039548218
z: 1.0
epoch:18, loss:0.00017542651039548218
z: 1.0
epoch:19, loss:0.00017542651039548218
z: 1.0
epoch:20, loss:0.00017542651039548218
0.000175426510395
GP: 0 =====
loss: 0.00017542651039548218
score: 1.0
GP: 1 =====
loss: 0.00012209427950438112
score: 1.0
GP: 2 =====
loss: 0.00012209427950438112
score: 1.0
GP: 3 =====
```

计划：

1. 对分数做处理，画出分数的彩色图
2. pagerank

241101

长时间休眠和睡眠，期间打开主机进行打印，然后系统崩溃，修复系统时文件损坏，不能修复，于是重装了系统。

1. 写readme文件

在服务器重新安装了pytorch，可以运行cuda，但是好像还是很慢

先用效果好的小图导出模型，计算数值往后做

241025

1. 仅用一个图对训练出模型

1. 不需要训练的数据怎么快速通过模型获得结果
2. 他的模型没有考虑每个扰动图之间的关系
3. 图对能够把所有扰动图考虑进去吗？
4. 不同重要性节点的图对训练的模型有什么不同
5. loss有尖刺，（存在离群点干扰？），是否跟设备有关？
6. 怎么调用已有的模型（√），需要forward吗，好像不用
7. 测试集对比（数据量太大，不易实现，先不管了）：
 1. 某扰动图模型的泛化能力，在其他扰动图上的损失效果： **代码变量调用可能有问题**
 2. 大图（效果不好，未知问题，放置），改变节点后模型的能力： 模型中间的层需要更改?? 或者只改变节点个数，不改变特征的维数
(节点个数会导致数据量变大，但似乎不影响层数。但是改变层数后模型行不收敛了)

如果损失函数开始时几乎不变（或者说变化很小），然后逐渐开始收敛，这通常意味着以下几种情况之一：

1. 初始阶段数据稀疏或权重初始化不佳：

- 在训练初期，如果数据集很大，或者权重初始化不当（比如所有权重都初始化为零），那么模型可能需要一些时间来“热身”，即开始有效地调整其参数以适应数据。

2. 学习率设置：

- 如果使用的是一个固定的学习率（learning rate），且这个值设置得过小，那么在训练初期模型的更新可能非常缓慢，导致损失函数的变化不大。随着训练的进行，模型参数逐渐调整，损失函数开始下降并收敛。

3. 梯度消失或爆炸：

- 如果模型结构较深或者激活函数选择不当（如使用了sigmoid或tanh作为深层网络中的激活函数），可能会遇到梯度消失或梯度爆炸的问题。这会导致在训练初期权重更新非常缓慢甚至不动。解决这个问题的方法包括使用ReLU这样的激活函数，或者采用梯度裁剪等技术。

4. 局部最小值：

- 损失函数可能有多个局部最小值，在某些情况下，模型可能会陷入一个局部最小值，在这里损失函数的变化很小。随着进一步的训练，它可能会跳出这个局部最小值，并继续向全局最小值靠近。

5. 正则化效应:

- 如果使用了正则化技术（如L1或L2正则化），这些技术可以帮助避免过拟合，并可能导致损失函数在某些阶段变化较小。但是，随着训练的继续，模型会学习到更合适的参数，损失函数会继续下降。

6. 数据预处理:

- 数据预处理（如归一化、标准化）可以影响损失函数的变化速率。如果数据未正确预处理，可能会导致训练初期损失函数变化不大。

在任何情况下，重要的是要监控训练过程中的损失函数和其他相关指标，以确保模型能够有效地学习并且不会过拟合或欠拟合。如果有疑问，可以尝试调整学习率、使用不同的优化算法、改变模型架构或者增加/减少正则化强度等方法来改善训练过程。

计划

先用效果好的小图导出模型，计算数值往后做

241024

1. GAT 需要edge index ,但是edge index 不规则，不能npz

2. 构建graph pair 类

3. `feature_1 = (feature_1 - mean_1) / std_1`

对每个特征进行标准化处理（也称为标准化或 z-score 标准化）是一种常用的数据预处理技术，目的是使每个特征具有零均值和单位方差。标准化处理可以改善数据的分布，从而提高机器学习模型的性能。以下是标准化处理的具体解释：

标准化处理公式

对于给定的一组数据 ($\{x_1, x_2, \dots, x_n\}$)，其标准化处理公式为：

$$[z_i = \frac{x_i - \mu}{\sigma}]$$

其中：

- (z_i) 是标准化后的数据；
- (x_i) 是原始数据；
- (μ) 是原始数据的平均值（均值）；
- (σ) 是原始数据的标准差。

标准化的目的

1. **消除量纲影响**：不同特征可能有不同的量纲或尺度，标准化后可以消除这种差异，使特征在同一尺度上进行比较。
2. **加速收敛**：在梯度下降等优化算法中，标准化后的数据可以使损失函数更快收敛。
3. **提高模型性能**：对于一些机器学习算法（如支持向量机、线性回归等），标准化后的数据可以提高模型的性能。

总结

通过上述标准化处理，每个特征的平均值变为 0，标准差变为 1。这样可以使得特征在相同的尺度上进行比较，有助于提高模型的性能和收敛速度。如果你有进一步的问题或需要更具体的解释，请随时告诉我。

4. `label_exp = torch.exp(-1 * label) # 无用`

您提到的将 `label` 转换为概率值的过程可以通过指数函数 (e^{-x}) 来实现。这里我们来详细解释一下为什么这样做可以将正数转换为接近 0 的值，负数转换为接近 1 的值。

指数函数的性质

指数函数 (e^{-x}) 有一些重要的性质：

1. 当 ($x > 0$):

- (e^{-x}) 的值会小于 1，并且随着 (x) 的增大，(e^{-x}) 的值迅速趋近于 0。

2. 当 ($x < 0$):

- (e^{-x}) 的值会大于 1，并且随着 (x) 的减小，(e^{-x}) 的值迅速趋近于无穷大。
- 但是，如果我们关注的是 ($e^{-|x|}$)，那么当 ($x < 0$) 时，(e^{-x}) 会变成 (e^x)，此时 (e^x) 的值会大于 1，并且随着 (x) 的减小，(e^x) 的值会迅速趋近于 0。

3. 当 ($x = 0$):

- ($e^{-0} = e^0 = 1$)。

举例说明

假设 `label` 是一个表示得分或距离的标量或向量，我们可以通过 ($e^{-\{x\}}$) 来将其转换为概率值：

◦ 当 `label` 为正数:

- ($e^{-\text{positive number}}$) 会接近 0，表示该得分或距离较高时，转换后的值接近 0。

◦ 当 `label` 为负数:

- ($e^{-\text{negative number}}$) 会接近 1，表示该得分或距离较低时，转换后的值接近 1。

修正为概率

如果我们希望将 `label` 转换为介于 0 和 1 之间的概率值，可以使用 Sigmoid 函数 ($\sigma(x) = \frac{1}{1+e^{-x}}$)。Sigmoid 函数确保所有的值都在 0 和 1 之间：

总结

通过 (e^{-x}) 或 Sigmoid 函数 ($\frac{1}{1+e^{-x}}$)，可以将 `label` 转换为介于 0 和 1 之间的概率值。这种方法在很多场景中都非常有用，特别是在需要将得分或距离转换为概率值的情况下。如果你有具体的上下文或应用场景，请提供更多信息，以便我能给出更具体的解释和建议。

5. 粗心大意，输错矩阵，不是大小对不上GAT，是维度

6. 如果用一个图对训练出模型，需要都训练一遍吗？(√) 还是后面的其实不用

1. 不需要训练的数据怎么快速通过模型获得结果
2. 他的模型没有考虑每个扰动图之间的关系？？
3. 还是说图对能够把所有扰动图考虑进去

7. 测试集对比一下吧（先不管了）

8. 某个图对的loss有尖刺，什么原理，是否跟设备有关？？

241023※

1. 每个图对都分别计算相似度分数，计算N个？

与AI交流<https://lxblog.com/qianwen/share?shareId=d1f6fc34-953b-4241-941d-a260f037fbba>

1. N个扰动图怎么凑图对：尽管存在N个扰动图，但每个扰动图都是独立地与基准图形成图对并通过相同的流程进行处理。

2. GAT中未扰动的节点也要学习表示？：GAT（图注意力网络）来学习节点表示时，即使某些节点本身没有改变，它们也会参与到其他节点特征的学习过程中，因为GAT是基于节点之间的相互作用来进行特征提取的。

此外，扰动图的设计目的是为了评估每个节点在网络中的贡献。通过对比扰动前后网络的变化情况，可以更好地理解每个节点的重要性。即使有些节点没有被直接扰动，它们在扰动图中的表现也会受到周围被扰动节点的影响，从而间接反映出它们在整个网络中的地位和功能。

综上所述，即使是那些在扰动图中没有直接发生改变的节点，它们的处理也不是多余的。

3. "图间"是？：相似度系数 β_{mj} 和 β_{in} 是用来衡量基准图 G_c 中节点 v_m 与扰动图 G_o 中节点 v_m^c 之间的相似度，以及基准图 G_c 中节点 v_i 与扰动图 G_o 中节点 v_c^i 之间的相似度。相似度系数是通过计算节点间表示向量的余弦相似度来获得的。

4. "图间表示"的加权：图间表示是通过加权获得的原因在于它旨在捕捉两个图之间节点特征的关联性，并以此来反映节点在不同图中的相似性和重要性。这里所谓的“加权”，实际上是对节点在基准图和扰动图中表示的融合过程。

5. 加权的N：在相似度系数计算过程中，分母中的 NN 指的是图对中的另一个图中的节点数量。

6. 学习一个节点的图间表示，为什么 h_m^* 和 h_i^* 要计算2边：这种做法的目的在于评估节点在不同图结构下的表现一致性或差异性。

7. 全局读出：获得图的表示

基准图和扰动图的节点级图内（没有考虑彼此的相似度）和图间表示（考虑彼此的相似度）

8. 交叉匹配：图对相似，则图对的图内表示和图间表示之间的差异很小

图间表示：假如图对相似，他们的图内表示相似，求的余弦相似度是这个点和其它节点的，**如果笼统的看作是节点在图中的表现的话肯定是相似，但具体而言呢？** 相似度系数肯定笼统而言，是相近的，加权后的表示也是。

9. 越重要，相似度越低

2. pycharm更新导致代码不能运行？？？

退回版本后可以了

```
from . import _csparssetools
ImportError: DLL load failed while importing _csparssetools: 动态链接库(DLL)初始化例程失败。
```

3. 写成函数似乎运行速度会变慢？？（通过计时，发现不会）

4. MGC的数据读取部分，以及模型函数的调用部分需要重构

241022

最近一直在忙汇报和实验课

关于扰动图，因为要考虑每个节点的重要性，所以每个节点都有一个扰动图



扰动确实是只扰动一个节点

GAT

那我GAT只需要对这一个节点做？（对每个节点都做，虽然某些节点没有改变，但是会参与到其他节点的学习中去，扰动图设计的目的是为了评估每个节点在网络中的贡献，虽然某些节点没有直接被扰动，它们在扰动图中的表现也会受到周围被扰动节点的影响，从而间接反映出他们在整个网络中的地位和功能）

本章均采用均方误差作为损失函数

采用重构损失？常见的重构损失包括均方误差（MSE）或交叉熵损失

加权求和（相似度系数）

1. 用相似度系数加权，是在和谁加权，N是谁？？总节点？？邻居？

GAT中已经考虑了邻居，图间表示是哪一步抽象？

不同图之间加权，对N个扰动图进行加权？

~~每个节点的扰动特性来源于N个扰动图中分别计算出的每个节点的表示，~~

~~可是，这样的话，大部分节点的表示都是相同的，每个扰动图只有一个节点只差~~

原图和扰动图分别图间表示

2. 相似度系数，是谁和谁相似，该节点在每个扰动图之间的表示

(图间表示没有搞懂，从图级表示入手倒推理解一下)

241017

文件调用

内存不够了，内存映射，但是好像没有效果

只扰动了一个点，某个点都一个扰动图

图对是每个节点都有一对吗

两个图分别GAT学习图表示？ G_c 有很多个图，怎么学习图表示

用传感器测量值作为节点特征这合理吗？？一段时间的温度数据是这个点的气候变化特征，对这个特征进行抽象，有什么意义吗

数据的特征，是该数据的抽象，表征，类似label，但不是准确的label

241016

重新下载数据集

整理报告的论文

241015

写实验报告

241014

看了看课程汇报的素材。但是没有定下题目。

241013

完成了无线投屏，确实略有延迟，但是可用

241012

配置了SSH的基本连接

pycharm需要专业版才能ssh连接

无线显示器方案