

华中科技大学

图神经网络实验报告

专 业： 计算机科学与技术
班 级： CS2005
学 号： U202090063
姓 名： 董玲晶
电 话： 13067217235
邮 箱： 1355532189@qq.com

独创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签名：董玲晶

日期：2023 年 11 月 14 日

成 绩	
教师签名	

目 录

1	概述.....	1
2	背景知识	3
2.1	关于 DEEPWALK 和 NODE2VEC	3
2.2	关于 GNN.....	4
3	问题定义	6
4	模型概览	7
4.1	框架总览	7
4.2	复现思路	9
5	实验设计	12
5.1	源码分析	12
5.2	实验过程记录	16
5.3	实验结果分析	17
6	总结与展望	18
7	课程感想	19
	参考文献.....	20

1 概述

程序漏洞是软件安全的主要威胁之一，然而在开发阶段几乎不可能避免漏洞；甚至很难在生产阶段发现漏洞。源代码缺陷检测是判别程序代码中是否存在非预期行为的过程，广泛应用于软件测试、软件维护等软件工程任务，对软件的功能保障与应用安全方面具有至关重要的作用。安全专家通常利用动态模糊测试、符号执行或静态代码审计来发现漏洞。然而，这些技术都不能提供一个很好的解决方案：动态模糊测试存在代码覆盖问题和初始种子问题；由于路径爆炸和约束解决问题，符号执行不能很好地扩展到现实世界的程序中；静态代码审计通常需要人类的专业知识，当程序复杂性增加时，它不能进行很好地扩展^[1]。

由于代码具备各个维度的特征，例如局部的文本共现特征与长程的数据、控制依赖特征，因此，如何对代码进行表征成为了设计深度学习模型需首要考虑的问题。当前的方法主要通过序列、树和图对代码进行表征，其常用的对应网络结构^[2]如图 1-1 所示。

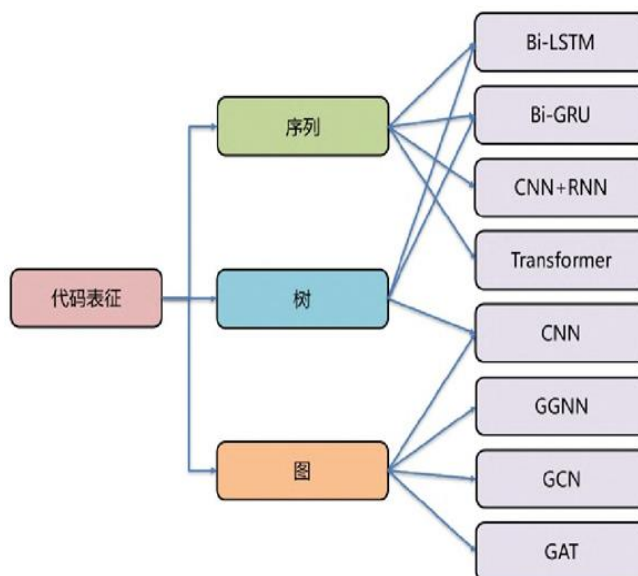


图 1-1. 代码表征的分类与应用模型

现有的方法在学习综合程序语义以表征高多样性、脆弱性以及真实世界的

代码的复杂性方面都有很大的局限^[3]。可以将现有方法的缺点总结为以下三点：

(1) **对源代码结构和逻辑的简化：**传统的学习方法将源代码视为平面序列或仅使用部分信息来表示代码，忽略了源代码的结构性和逻辑性。源代码通常具有抽象语法树、数据流、控制流等复杂的表示方式，这些信息对于准确理解代码含义是至关重要的。

(2) **缺乏综合检测漏洞的能力：**源代码中的漏洞有时是微妙的缺陷，需要从语义的多个维度进行综合调查才能发现。然而，现有方法在设计上存在缺陷，限制了其发现各种漏洞的潜力。

(3) **训练数据的问题：**部分数据是由静态分析器标注的，这导致了高比例的误分类，即被标记为漏洞但实际上并不是真正的漏洞。另一部分数据是人工编写的简单代码，相比于真实世界的代码过于简单。

本文提出了一种基于图神经网络的新型模型，该模型采用复合编程表示法来处理事实漏洞数据，能够编码一整套经典的编程代码语义，以捕捉各种脆弱性特征。本文的创新点如下：

(1) 首次使用图的数据结构，并将不同层次的程序控制和数据依赖关系编码成一个由异质边¹组成的联合图，每条异质边上的类型表示与相应表征的连接，这样有利于捕获尽可能广泛的漏洞类型和模式，并能够通过图神经网络学习更好的节点表示；

(2) 提出了一个新的带卷积模块的门控图神经网络模型，用于图级别的分类，有利于捕捉更高层次的图特征；

(3) 收集了许多手工标记的数据集并准确、有效验证了模型的性能，实验结果显示：本模型在缺陷检测任务的效果上显著高于之前方法；同时，在 112 个真实项目的缺陷函数中检测准确率达到 74%。

¹ 异质边 (Heterogeneous edges) 是指在图结构中连接不同类型节点之间的边。在异质图

(Heterogeneous graph) 中，节点可以表示不同的实体或概念，而异质边则表示这些实体或概念之间的关系或连接。异质边的存在使得我们能够在图结构中更好地捕捉不同实体之间的复杂关系和连接模式，从而提供更准确和全面的图分析和挖掘能力。

2 背景知识

个人对 GNN 的理解与总结如果用一句话表示就是：**信息传播+信息聚合**，**不断拉取和汇聚邻居节点的信息**。剩下的就是在信息传播和聚合的方法里做不同的文章，以下展开一些我在学习过程中认为比较核心或印象比较深的部分，尤其是我花了比较多时间理解的公式。

2.1 关于 deepwalk 和 node2vec

deepwalk^[9]都假设走一个固定长度的步且每一步都是随机选择而没有偏好，但这种方法对于来自附近同一网络社区的节点以及具有相似结构角色的节点的关注不够，如果我们更关心同一个社区里类似的节点以及远方类似结构的社区，这种方法就不够好，比如下图^[8]中的节点 1、2、3 构成的社区和 11、12、13 构成的社区。



图 2-1. 具有两个类似结构子网络的图示意图

但 node2vec^[10]就很巧妙地对这个方法进行了改进，它可以有意识地决定我们要走的方向并结合了经典的 BFS 和 DFS 图遍历算法。如下图^[8]，如果我今天想走 local 的就采用 BFS 策略，把 p 的值调小， $1/p$ 机率就更大；但如果我们今天想去远方，我们就采用 DFS 策略，把 q 的值调小， $1/q$ 的概率就更大，更容易往远方走。

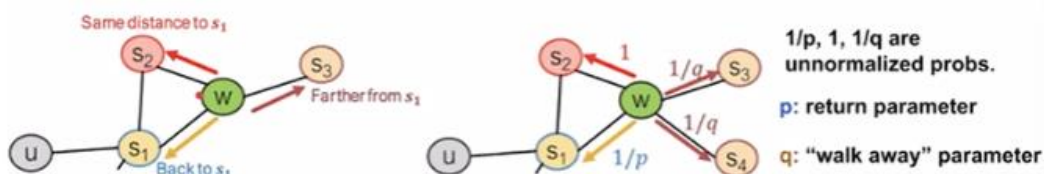


图 2-2. node2vec 算法示意图

2.2 关于 GNN

2.2.1 理论总结

(1) 信息传播，对邻接节点 u 的嵌入特征做转换：

$$m_u^{(l)} = MSG^{(l)}(h_u^{(l-1)})$$

这里的 MSG 可以是任何形式的变换，可以是线性的也可以是非线性的，甚至可以是多层网络的。

(2) 收集完邻居的信息后再做聚合：

$$h_v^{(l)} = AGG^{(l)}(\{m_u^{(l)}, u \in N(v)\})$$

这里的聚合可以是 $Sum(\cdot)$, $Mean(\cdot)$, $Max(\cdot)$ 任意函数，视情况而定。

(3) 对于当前节点 v ，若要加上自己本身的信息，则：

$$h_v^{(l)} = CONCAT(AGG(\{m_u^{(l)}, u \in N(v)\}), m_v^{(l)})$$

2.2.2 GCN

GCN^[6]在做的事情就是对邻居的信息做卷积操作，即 MSG 为卷积层，并且进行 $normalize$ ， AGG 取 $Sum(\cdot)$ 。即：

$$h_v^{(l)} = \sigma \left(\sum_{u \in N(v)} W^{(l)} \frac{h_u^{(l-1)}}{|N(v)|} \right)$$

变换后就是

$$h_v^{(l)} = \sigma \left(Sum(\{m_u^{(l-1)}, u \in N(v)\}) \right), \quad m_u^{(l)} = \frac{1}{|N(v)|} W^{(l)} h_u^{(l-1)}$$

如果写成矩阵计算的形式就是：

$$A' = A + I, \quad \hat{A} = D^{-\frac{1}{2}} A' D^{-\frac{1}{2}}$$

$$H_v^{(l)} = \sigma(\hat{A} H_v^{(l-1)} W^{(l)})$$

2.2.3 GAT

公式如下：

$$h_v^{(l)} = \sigma \left(\sum_{u \in N(v)} \alpha_{vu} W^{(l)} h_u^{(l-1)} \right)$$

这里的 α_{vu} 是注意力权重，也是通过学习得到的。在 GCN 里， $\alpha_{vu} = \frac{1}{|N(v)|}$ 即对于某个 node 所有的邻居都是一个固定的权重，所有的邻居都是同等重要。但在 GAT 里可以通过另外一组权重的学习来为不同的邻居分配不同的重要性。

假设 e_{vu} 代表了 node-u 和 node-v 之间的重要性， $a(\cdot)$ 是一个可以由任何计算方法叠加的算子，输入的是两个节点的嵌入特征，输出的是两个节点之间的重要性，计算公式为：

$$e_{vu} = a(W^{(l)} h_u^{(l-1)}, W^{(l)} h_v^{(l-1)})$$

然后再用 softmax 函数做 normalize，使得所有邻居的 e_{vu} 相加的和为 1：

$$e_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

同时，我们可以对单个节点学习多组 α ，再对这些结果做一个聚合。

$$h_v^{(l)}[1] = \sigma \left(\sum_{u \in N(v)} \alpha_{vu}^1 W^{(l)} h_u^{(l-1)} \right) \dots h_v^{(l)}[n] = \sigma \left(\sum_{u \in N(v)} \alpha_{vu}^n W^{(l)} h_u^{(l-1)} \right)$$

$$h_v^{(l)} = AGG(h_v^{(l)}[1], h_v^{(l)}[2], \dots, h_v^{(l)}[n])$$

最经典的图就是这张^[4]：

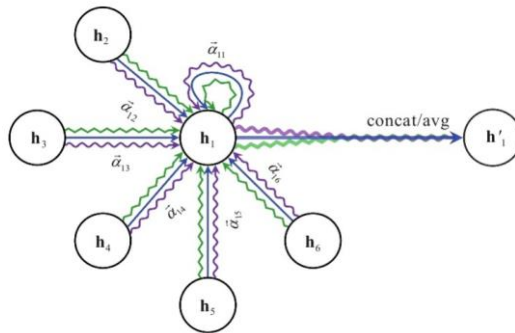


图 2-3. Multi-head Attention 机制示意图

3 问题定义

可以将代码漏洞检测问题转换为一个二元分类问题，即给定一个源代码，判定其是否存在函数级别的代码漏洞。假设代码数据样本定义为：

$$((c_i, y_i) \mid c_i \in \mathcal{C}, y_i \in \mathcal{Y}), i \in \{1, 2, \dots, n\}$$

其中 \mathcal{C} 表示代码中的函数集； $\mathcal{Y} = \{0, 1\}^n$ 中 1 表示有代码漏洞易受攻击，0 表示无； n 是实例个数。

- 把每个函数 c_i 编码成一个多边图 $g_i(V, X, A) \in \mathcal{G}$;
- m 是 V 的节点个数;
- $X \in \mathbb{R}^{m \times d}$ 是初始节点特征矩阵，每个节点 v_j 被表示为一个 d 维的向量 $x_j \in \mathbb{R}^d$ （即每个节点的特征是 $1 \times d$ ， V 中一共有 m 个节点，因此 \mathbb{R} 的大小是 $m \times d$ ）;
- 邻接矩阵 $A \in \{0, 1\}^{k \times m \times m}$ ，其中 k 是边类型的总数， $e_{s,t}^p \in A$ 等于 1 表示节点 v_s 和 v_t 通过类型 p 的边相连，否则为 0（每个节点都有可能与其它节点有 p 种类型的边相连，因此大小为 $k \times m \times m$ ）

所以模型的目标是学习从 \mathcal{G} 到 \mathcal{Y} 的映射 $f: \mathcal{G} \mapsto \mathcal{Y}$ 来预测一个函数里是否存在漏洞，使用交叉熵损失函数，并使用可调权重 λ 进行正则化 $\omega(\cdot)$ ，我们需要做的就是最小化：

$$\sum_{i=1}^n \mathcal{L}(f(g_i(V, X, A), y_i | c_i)) + \lambda \omega(f)$$

4 模型概览

本章节将从两个方面展开，一方面是回顾论文的模型设计，另一方面是阐述复现思路以及展示代码。

4.1 框架总览

模型包含三个部分：（1）复合代码语义的图嵌入层；（2）门控图循环层；（3）卷积层模块。各自功能如下：

- 将源代码编码成具有综合程序语义的联合图结构；
- 通过消息传递和消息聚合邻居节点的信息以学习每个节点特征；
- 提取有用的节点表征用于图层面的预测。

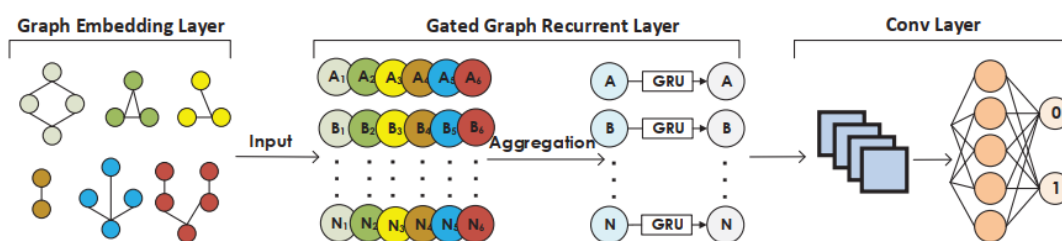


图 4-1. Design 模型框架示意图

4.1.1 代码语义的图嵌入

常见的代码程序表示方法有抽象语法树、控制流和数据流图（编译原理学过的那些）。比较优秀的漏洞检测方法需要结合这三种表示方法形成一个联合数据结构，同时本模型还考虑了源代码的自然序列，即一共四种表示方法。具体做法为：

将函数 c_i 用联合图 g_i 表示，其中四类子图共享同一组节点 $V = V^{ast}$ ，这里的每个节点 $v \in V$ 都有两个属性：*Code* 和 *Type*。*Code* 属性通过使用预训练的 word2vec 模型对源代码进行编码来表示，而 *Type* 属性则表示节点的类型；然后再将 *Code* 和 *Type* 的编码连接在一起，形成初始节点表示 x_v ，以便在后续的图神经网络或图表示学习中使用。这样，每个函数 c_i 都可以通

过一个联合图 g 来表示，并且可以利用这个联合图来进行进一步的分析和学习。

4.1.2 门控图循环网络

给定一个嵌入图 $g_i(V, X, A)$ ，对于每个节点 $v_j \in V$ ，状态向量 $h_j^{(1)}$ 被初始化为 $[x_j^T, 0]^T$ ，这里因为 $h_j^{(1)} \in \mathbb{R}^z, z \geq d$ ，所以多余的位置被用 0 来填充。

在每个时刻 $t \leq T$ 中，节点之间通过相应的边类型和方向进行信息传递。邻接矩阵 A_p 描述了从节点 v_j 到其他节点的连接情况，通过在每个时间步中执行这样的邻居聚合操作，可以在整个图中传播和聚合信息，从而使每个节点能够获取来自邻居节点的上下文信息，并将其整合到自身的状态中。

具体，到达 t 时刻，对于某个节点 j 的某种类型 p 拉邻接结点进行信息传播后的结果为：

$$a_{j,p}^{(t-1)} = A_p^T \left(W_p \left[h_1^{(t-1)^T}, \dots, h_m^{(t-1)^T} \right] + b \right)$$

我对以上的公式进行了改造以助于理解：

$$a_{j,p}^{(t-1)} = \sum_{u \in N(v)} W_p \frac{h_u^{(t-1)}}{|N(v)|} + b$$

这里 $N(v)$ 表示节点 u 的邻接结点，乘上 A_p^T 就是为了过滤掉非邻接结点。

然后对传递的信息进行聚合，这里 $AGG(\cdot)$ 可以是任何函数，如 $Sum(\cdot)$ ， $Mean(\cdot)$ ， $Max(\cdot)$ ，CS224W 的课程里有提到一般情况下 $Sum(\cdot)$ 性能是最佳的，此论文也用了该函数。

因此当前时刻节点的隐藏状态就可以通过前一时刻的隐藏状态和聚合后的邻居信息表现为：

$$h_j^{(t)} = GRU \left(h_j^{(t-1)}, AGG \left(\{a_{j,p}^{(t-1)}\}_{p=1}^k \right) \right)$$

在经历了 T 时刻这样的更新后， $H_i^{(T)} = \{h_j^{(T)}\}_{j=1}^m$ （即所有节点最后一个隐藏层表示的并集）就是最后集合 V 的节点表示。

4.1.3 卷积模块

应用一维卷积和全连接层来学习与图级任务相关的特征，以获得更有效的预测。定义 $\sigma(\cdot)$ 算子如下：

$$\sigma(\cdot) = \text{MAXPOOL}(\text{Relu}(\text{CONV}(\cdot)))$$

它将输入的特征进行卷积操作，并通过 ReLU 激活函数进行非线性变换，然后使用最大池化对卷积结果进行下采样。

同时为了利用每个节点本身的信息，devign 分别训练了两个卷积模块，得到特征：

$$Z_i^{(1)} = \sigma\left(\left[H_i^{(T)}, x_i\right]\right), \dots, Z_i^{(l)} = \sigma\left(Z_i^{(l-1)}\right)$$
$$Y_i^{(1)} = \sigma\left(H_i^{(T)}\right), \dots, Y_i^{(l)} = \sigma\left(Y_i^{(l-1)}\right)$$

最后再将两个特征输入到 MLP 中，然后对乘法结果进行平均聚合，最后使用 Sigmoid 激活函数将其映射到[0, 1]的范围内，得到节点的预测结果。

$$\tilde{y}_i = \text{Sigmoid}\left(\text{AVG}\left(\text{MLP}\left(Z_i^{(l)}\right) \odot \text{MLP}\left(Y_i^{(l)}\right)\right)\right)$$

4.2 复现思路

根据论文里的内容，利用每个节点本身的信息，分别训练了两个卷积模块，最后再将两个特征输入到 MLP 中，然后对乘法结果进行平均聚合，最后使用 Sigmoid 激活函数将其映射到[0, 1]的范围内，得到节点的预测结果。按照公式摸出来就行。这里我使用两层卷积模块和对应的池化层，参数如下：

```
# 设置卷积层和池化层参数
self.conv1_size = {
    "in_channels": self.max_nodes, "out_channels": 64,
    "kernel_size": 3, "padding": 1
}
self.conv2_size = {
    "in_channels": 64, "out_channels": 16,
    "kernel_size": 2, "padding": 1
}
self.maxp1_size = {
```

```
"kernel_size": 3, "stride": 2
}
self.maxp2_size = {
    "kernel_size": 2, "stride": 2
}

self.feature1 = nn.Conv1d(**self.conv1_size)
self.maxpool1 = nn.MaxPool1d(**self.maxp1_size)
self.feature2 = nn.Conv1d(**self.conv2_size)
self.maxpool2 = nn.MaxPool1d(**self.maxp2_size)
```

对于两个特征Z和Y对应的卷积层后面都要接全连接层，全连接层的第一个参数由老师给的函数 `get_conv_mp_out_size()` 计算（但是在后面实验的过程中其实遇到了一些小问题）调用传入的第一个参数需要由前面的图神经网络的参数计算，所以需要在初始化函数里添加图神经网络的 `out_channels` 和 `emb_size`，根据公式：

$$Z_i^{(1)} = \sigma \left(\left[H_i^{(T)}, x_i \right] \right)$$

$$Y_i^{(1)} = \sigma \left(H_i^{(T)} \right)$$

Z的h和w应该是 `graph_out_chs + emb_size`，而Y的h和w就只有 `graph_out_chs`。

```
# 根据conv和maxpool参数计算mlp尺寸
self.mlp1_size = get_conv_mp_out_size(
    graph_out_chs + emb_size,
    self.conv2_size,
    [self.maxp1_size, self.maxp2_size]
)
self.mlp2_size = get_conv_mp_out_size(
    graph_out_chs,
    self.conv2_size,
    [self.maxp1_size, self.maxp2_size]
)
self.mlp1 = nn.Linear(1200, 1)
self.mlp2 = nn.Linear(self.mlp2_size, 1)
```

在前向传播过程中，对于Z，首先需要将输入的 `h` 和 `x` 在特征维度上进行拼接，再进行 `reshape` 操作，接着通过两个 $\sigma(\cdot)$ 算子（即我们前面定义好的两层卷积层、激活层和对应的最大池化层），然后将得到的张量展平，输入到第一个全连接层中。Y的操作同理，唯一的区别就是不进行 `h` 和 `x` 的拼接。最后，将两

个结果相乘并展平为一维张量，并经过 Sigmoid 激活函数得到最终的输出结果。

```
def forward(self, h, x):
    z_f = torch.cat([h, x], 1)
    z_f = z_f.view(-1, self.max_nodes, h.shape[1] + x.shape[1])
    out_z = self.maxpool1(F.relu(self.feature1(z_f)))
    out_z = self.maxpool2(F.relu(self.feature2(out_z)))
    out_z = out_z.view(-1, int(out_z.shape[1] * out_z.shape[-1]))
    out_z = self.mlp1(out_z)
    y_f = h.view(-1, self.max_nodes, h.shape[1])
    out_y = self.maxpool1(F.relu(self.feature1(y_f)))
    out_y = self.maxpool2(F.relu(self.feature2(out_y)))
    out_y = out_y.view(-1, int(out_y.shape[1] * out_y.shape[-1]))
    out_y = self.mlp2(out_y)
    out = out_z * out_y
    out = torch.sigmoid(torch.flatten(out))
    return out
```

以上代码的框架图如图 4-2 所示。

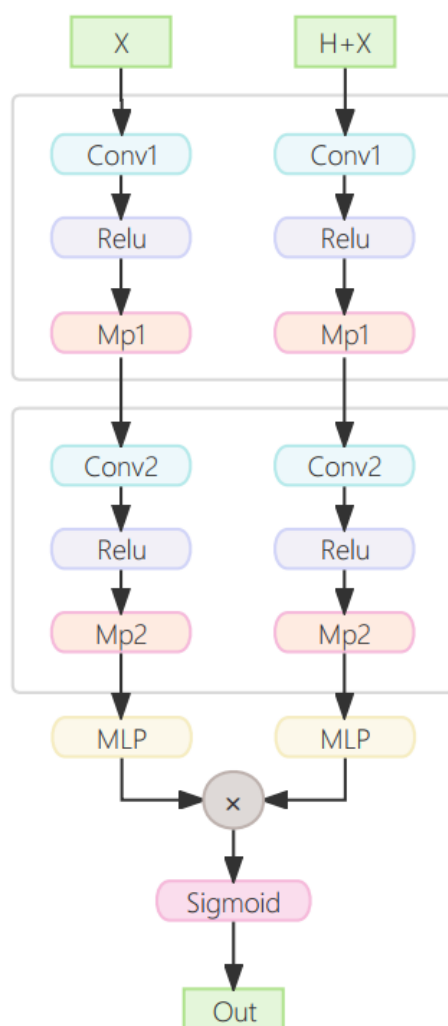


图 4-2. 复现代码框架示意图

5 实验设计

本人实验仓库：https://github.com/Elubrazione/gnn_lab_hust. 本章节将从系统源码分析、实验过程记录及实验结果分析三个部分展开。

5.1 源码分析

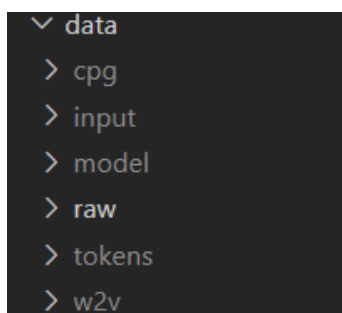
从 main.py 文件入手，实验代码最上层调用了三个函数 `setup()`、`embed_task()` 以及 `process_task()`。

5.1.1 `setup()`

创建 `Path()` 里定义的文件夹，我们到 `configs.json` 里可以看到，路径为 `data/*`。

```
"paths": {  
  "cpg": "data/cpg/",  
  "joern": "data/joern/",  
  "raw": "data/raw/",  
  "input": "data/input/",  
  "model": "data/model/",  
  "tokens": "data/tokens/",  
  "w2v": "data/w2v/"  
},
```

`data/raw/` 存放着我们在第一步里探索的原始代码数据集，`data/cpg/` 存放着我们从根目录的压缩包里解压出来的用 `joern` 工具生成的 CPG 图。执行完第二步的 `main.setup()` 就会创建其它文件夹以供后续使用。



```
▼ data  
  > cpg  
  > input  
  > model  
  > raw  
  > tokens  
  > w2v
```

图 5-1. 执行 `main.setup()` 后 `./data` 目录下内容

5.1.2 embed_task()

该函数训练了用于编码节点中文本的 word2vec 模型，并对节点的文本进行编码，保存预处理后的数据集。这里我画了个流程图。

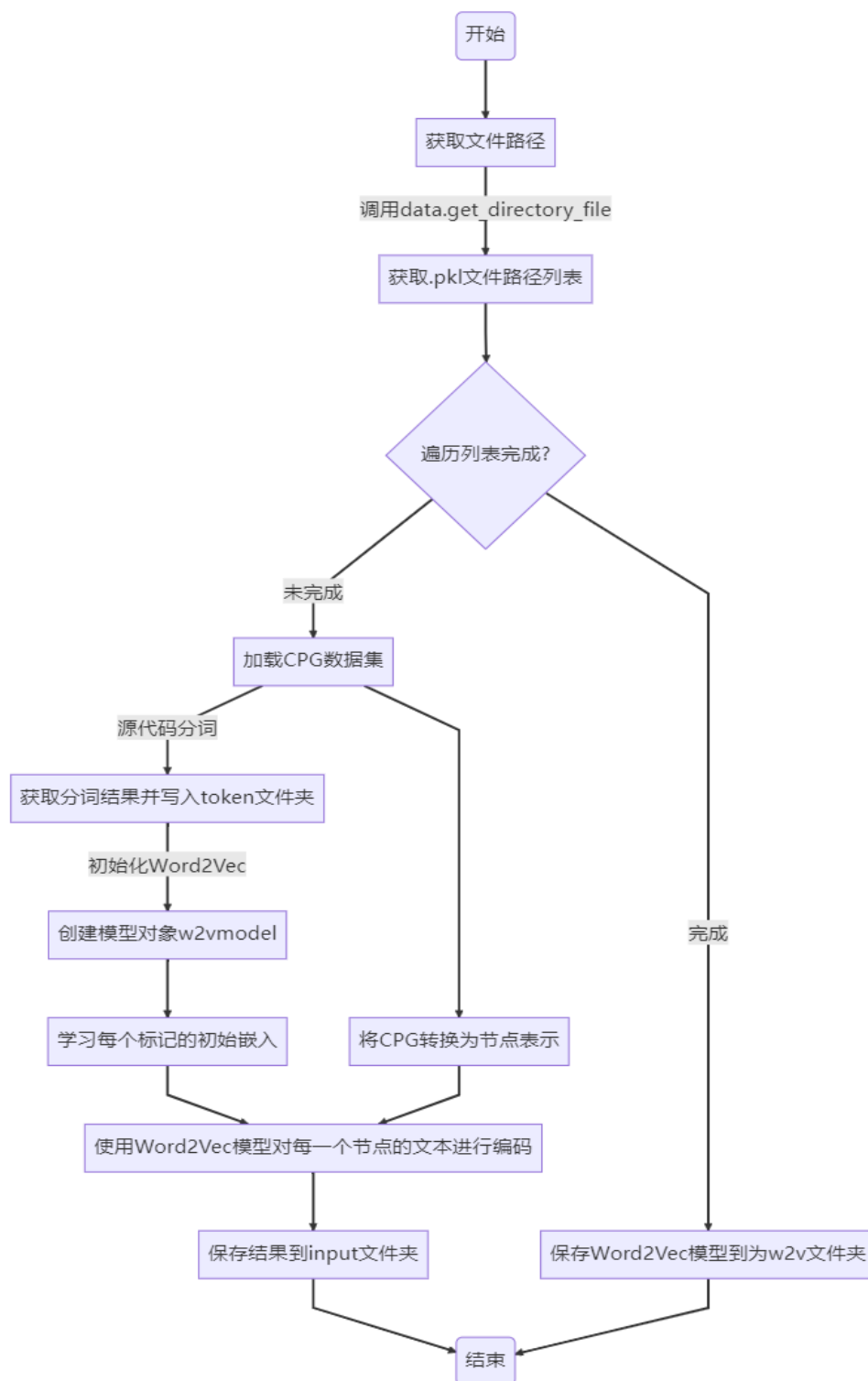


图 5-2. main.embed_task ()函数流程图

5.1.3 process_task()

同样画了个流程图，如图 5-4 所示。这里需要说明的是，根据 `./process/modeling.py` 的 `Train()` 函数，早停版本训练里计算验证集的 `loss` 后需要确认验证集的 `loss` 是否下降，如果下降的话就保存当前模型的参数到 `./data/model/` 目录下，命名为 `checkpoint.pt`，之后会从 `Net()` 类的 `load()` 方法里加载该参数以进行后续的测试集测试。而如果只是普通训练会根据 `configs.json` 里定义的 200 轮训练并保存模型。

`Train` 的调用方法里数据集迭代器用的不是我们平常用的 `DataLoader`，而是自己封装的一个类似功能的 `LoaderStep` 类，里面又嵌套调用了 `Stats` 类来计算损失和准确率。同时 `Train` 里调用了一个封装的 `History` 类来进行控制台打印和 `log` 文件写入。

值得一提的是，在跑完一个结果后我又仔细看了一下代码，早停训练原来并不是固定的 10 轮后就停止，根据 `./process/stopping.py` 里的 `EarlyStopping()` 的代码，应该是验证集的 `loss` 没有下降后默认十轮（也是由 `configs.py` 文件定义）停止。而我第一次跑的过程中，`Epoch1` 里 `loss` 从正无穷大变化到 0.693208，下降，所以保存了一个测试点模型，而 `Epoch2` 开始后面验证集 `loss` 都没有下降（如图 5-3 所示），因此又跑了十轮停止。而如果在 `Epoch2` 之后出现了 `loss` 下降则会重新保存模型并触发新的一套十轮 `EarlyStop` 计数。

```
Epoch 1; - Train Loss: 0.693; Acc: 0.0; - Validation Loss: 0.6932; Acc: 0.0; - Time: 79.72949171066284  
Validation loss decreased (inf --> 0.693208). Saving model ...  
  
Epoch 2; - Train Loss: 0.6926; Acc: 0.0; - Validation Loss: 0.6935; Acc: 0.0; - Time: 165.7048897743225  
EarlyStopping counter: 1 out of 10  
  
Epoch 3; - Train Loss: 0.6912; Acc: 0.0; - Validation Loss: 0.6941; Acc: 0.0; - Time: 239.62007784843445  
EarlyStopping counter: 2 out of 10
```

图 5-3. 训练过程控制台打印信息

看这份代码我的感想是各个函数分装地非常清楚，很多我以前会写在一起的功能在这份代码里都进行了分装，在一开始我并不清楚某些功能具体是怎么实现的（比如 `EarlyStop` 和 `LoaderStep`），只是大概知道是干什么的，也完全不影响我跑实验，同时在阅读代码的时候确实感到比较清晰，层层递进。

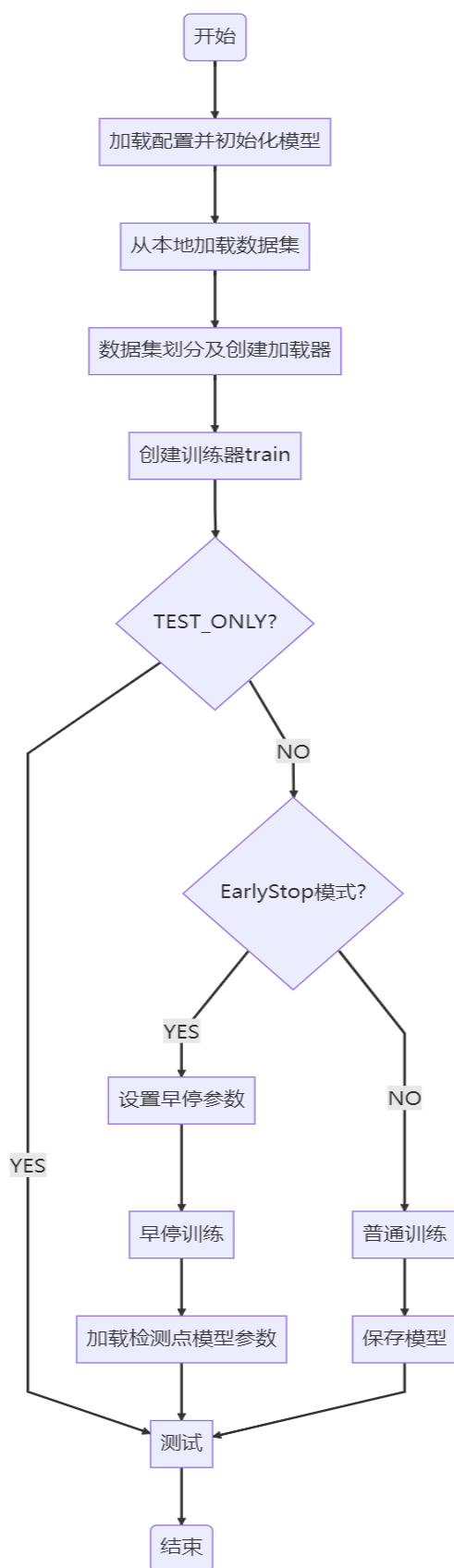


图 5-4. main.process_task ()函数流程图

5.2 实验过程记录

环境安装的过程中没有遇到太大的问题，只是在之前上其它课配置的环境的基础上按照 torch-geometric 官网的教程安装了这个包。基本情况如下：

```
(base) C:\Users\Jelly>nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Wed_Sep_21_10:41:10_Pacific_Daylight_Time_2022
Cuda compilation tools, release 11.8, V11.8.89
Build cuda_11.8.r11.8/compiler.31833905_0

(base) C:\Users\Jelly>conda activate pytorch

(pytorch) C:\Users\Jelly>python
Python 3.9.15 (main, Nov 24 2022, 14:39:17) [MSC v.1916 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> torch.cuda.is_available()
True
>>> print(torch.__version__)
1.13.0
```

图 5-5. 环境参数

开始跑训练的时候又遇到了 CUDA out of memory 的问题了，改用 CPU 跑，把配置文件的 process.use_gpu 改成 false 然后重启 ipynb 文件。

```
OutOfMemoryError: CUDA out of memory. Tried to allocate
242.00 MiB (GPU 0; 2.00 GiB total capacity; 1.49 GiB al
ready allocated; 0 bytes free; 1.51 GiB reserved in tot
al by PyTorch) If reserved memory is >> allocated memor
y try setting max_split_size_mb to avoid fragmentation.
See documentation for Memory Management and PYTORCH_CUD
A_ALLOC_CONF
```

图 5-6. GPU 容量不够的提示

遇到的第二个 bug 是，Z 特征通过卷积层接全连接层时调用老师给的函数 get_conv_mp_out_size() 算出来的尺寸是 1216，但是实际是 1200 所以对不上，暴力修改 Linear 层的第一个参数为 1200。

```
RuntimeError: mat1 and mat2 shapes cannot be multiplied (512x1200 and 1216x1)
```

图 5-7. 卷积层接全连接层参数对不上报错提示

跑完后的结果。

```
lab.ipynb (output) X main.py M lab.ipynb M
653 0 1
654 1 1
655 2 1
656 3 1
657 4 0
658 | | | ..
659 225 0
660 226 1
661 227 0
662 228 0
663 229 1
664 Length: 230, dtype: int64
665
666 Confusion matrix:
667 [[32 84]
668  [18 96]]
669 TP: 96, FP: 84, TN: 32, FN: 18
670 Accuracy: 0.5565217391304348
671 Precision: 0.5333333333333333
672 Recall: 0.8421052631578947
673 F-measure: 0.653061224489796
```

图 5-8. 控制台打印输出的测试结果

5.3 实验结果分析

一共有 230 个测试样本，被正确地分类为正例即存在代码漏洞的样本有 96 个，而错误地分类为有代码漏洞的样本高达 84 个。正确地分类成无代码漏洞的样本有 32 个，错误地分类为无代码漏洞地样本为 18 个。

从结果可以看出模型的准确率为 55.7%，F1 指数为 65.3%，论文里 Devign(AST)组对应的数据是 69.21%和 69.99%。与之相比准确率较低但 F1 指数较为接近，这是因为虽然精确率（53.3%）较低但召回率（84.2%）很高。

综合来看，模型在该实验中的准确率较低，在识别正例方面的表现相对较好，但对于负例的分类存在较多的错误，感觉可能主要的问题还是特征提取不够充分。

6 总结与展望

在本次实验中，我了解了代码漏洞检测领域相关的背景和研究现状，深刻认识到了代码漏洞检测领域的重要性和挑战性。

我还完成了 Devign 论文模型的复现。在论文复现过程中，我不仅巩固了在课堂上学习到的知识，还学习到了如何使用 Python 和 PyTorch 等工具来实现图神经网络模型，并对其进行训练和测试。通过在阅读作者的代码，我学习到了一些精炼、高效的写法和训练模型的一些小技巧，这些技能不仅在本次实验中有应用，也将会助力我未来的学习和研究。

7 课程感想

首先，对于这个课程我感到比较遗憾的是这个课实验部分的内容比较单薄，希望课题组如果有时间和意愿可以改善一下实验部分的内容，比如设计一些有趣的实验或者引进一些国外课程的实验内容，印象里《计算机视觉》课程的实验和结课作业就挺好的，还有《计算机系统基础》课程的实验是引自 CMU 很经典的 Bomb Lab。

但抛开以上部分不谈，我个人挺喜欢这个课的，在这个课上学习到了很多，而且抱着修了就要学点东西的想法以及在“完成不了作业就完蛋”想法的激励下课后我也花了点时间学习，收获很多。最值得欣喜的一点是以前乱七八糟、非常抽象的知识和公式现在终于变得和蔼可亲了许多。总的来说，这是门很不错的课，虽然大四有点忙，但我还是很庆幸修了这门课。

以上就是全部，完结撒花。

参考文献

- [1] Xu Z, Chen B, Chandramohan M, et al. Spain: security patch analysis for binaries towards understanding the pain and pills[C]//2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 2017: 462-472.
- [2] 邓梟, 叶蔚, 谢睿, 等. 基于深度学习的源代码缺陷检测研究综述[J]. 软件学报, 2023, 34(2): 625-654.
- [3] Zhou Y, Liu S, Siow J, et al. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks[J]. Advances in neural information processing systems, 2019, 32.
- [4] Veličković P, Cucurull G, Casanova A, et al. Graph attention networks[J]. arXiv preprint arXiv:1710.10903, 2017.
- [5] Scarselli F, Gori M, Tsoi A C, et al. The graph neural network model[J]. IEEE transactions on neural networks, 2008, 20(1): 61-80.
- [6] Kipf T N, Welling M. Semi-supervised classification with graph convolutional networks[J]. arXiv preprint arXiv:1609.02907, 2016.
- [7] Zhou J, Cui G, Hu S, et al. Graph neural networks: A review of methods and applications[J]. AI open, 2020, 1: 57-81.
- [8] <https://web.stanford.edu/class/cs224w/>
- [9] Perozzi B, Al-Rfou R, Skiena S. Deepwalk: Online learning of social representations[C]//Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. 2014: 701-710.
- [10] Grover A, Leskovec J. node2vec: Scalable feature learning for networks[C]//Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining. 2016: 855-864.