ĐẠI HỌC QUỐC GIA TP.HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

BÁO CÁO ĐỒ ÁN

PHÂN LOẠI MỨC ĐỘ PHỨC TẠP CỦA MÃ NGUỒN PYTHON THÔNG QUA AST SỬ DỤNG GNN

Giáo viên hướng dẫn: ThS Hà Lê Hoài Trung

Sinh viên thực hiện:

Hoàng Bảo Long	22520807
Đặng Trần Long	22520805
Lương Tuấn Vỹ	22521711
Nguyễn Duy Phương	22521165

TP. Hồ Chí Minh, tháng 5 năm 2025

LỜI CẢM ƠN

Trước tiên, chúng em xin gửi lời cảm ơn chân thành đến quý thầy cô trong khoa Hệ Thống Thông Tin, Trường Đại học Công Nghệ Thông Tin, đặc biệt là giảng viên Hà Lê Hoài Trung, thầy đã tận tình chỉ bảo, định hướng và hỗ trợ chúng em trong suốt quá trình thực hiện đề tài này. Những góp ý chân thành của thầy giúp chúng em có được những kiến thức chuyên môn cũng như phương pháp nghiên cứu quý báu.

Bên cạnh đó, chúng em xin trân trọng cảm ơn gia đình đã luôn là nguồn động viên to lớn, tiếp thêm sức mạnh tinh thần để em có thể hoàn thành tốt quá trình học tập và nghiên cứu.

Ngoài ra không thể không cảm ơn tập thể lớp IS353.P11 nói chung và những thành viên trong nhóm nói riêng đã có những đóng góp, ý kiến để nhóm có thể cải thiện chất lượng dự án. Cảm ơn vì các bạn đã đồng hành cùng chúng mình trong suốt quá trình thực hiện đồ án môn học.

Trong quá trình làm đồ án môn học, khó tránh khỏi sai sót, rất mong thầy qua. Đồng thời do trình độ lý luận cũng như kinh nghiệm thực tiễn còn hạn chế nên bài báo cáo không thể tránh khỏi những thiếu sót, nhóm em rất mong nhận được ý kiến đóng góp từ thầy để nhóm em học thêm được nhiều kinh nghiệm và sẽ hoàn thành tốt hơn những đồ án khác trong tương lai. Nhóm em xin chân thành cảm ơn!

TP. Hồ Chí Minh, tháng 5 năm 2025

Nhóm thực hiện

NHẬN XÉT CỦA GIẢNG VIÊN

……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……

....., ngày.....tháng.....năm 2024

Người nhận xét
(Ký tên và ghi rõ họ tên)

MỤC LỤC

MỤC LỤC HÌNH ẢNH **6**

CHƯƠNG I: TỔNG QUAN **8**

1. Giới thiệu đề tài 8

1.1. Lý do chọn đề tài 8

1.2. Mục tiêu đề tài 9

1.3. Loại bài toán xử lý 10

2. Mô tả dữ liệu 11

3. Mô hình bài toán 13

CHƯƠNG II: CƠ SỞ LÝ THUYẾT **15**

1. Graph Convolutional Network (GCN) 15

1.1. Giới thiệu tổng quan 15

1.2. Nguyên lý hoạt động 15

1.3. Ứng dụng 16

2. Graph Attention Network (GAT) 16

2.1. Giới thiệu tổng quan 16

2.2. Nguyên lý hoạt động 17

2.3. Ứng dụng 18

3. Cấu trúc Abstract Syntax Tree (AST) 18

3.1. Giới thiệu tổng quan 18

3.2. Nguyên lý hoạt động 19

3.3. Ứng dụng 21

4. Các đặc trưng của graph 22

4.1. GCN 22

4.2. GAT 23

4.3. Độ tương đồng (Similarity) 23

CHƯƠNG III: TÌM HIỂU MỘT NGHIÊN CỨU KHOA HỌC **25**

1. Giới thiệu tổng quan 25

2. Nguyên lý hoạt động của GCN 25
3. Ứng dụng của GCN trong mạng xã hội 27
4. Ưu và nhược điểm của phương pháp GCN 28
4.1. Ưu điểm 28
4.2. Nhược điểm 29
5. So sánh Graph Convolutional Network và Graph Attention Network 30
CHƯƠNG IV: HIỆN THỰC VÀ KẾT QUẢ 32
1. Giới thiệu bộ dữ liệu 32
2. Tiền xử lý dữ liệu 35
2.1. Load dữ liệu 35
2.2 Chuyển AST thành đồ thị PyG 36
2.3. Gán nhãn đồ thị theo kích thước AST 37
2.4. Cân bằng tập huấn luyện 38
2.5. Thống kê cơ bản 39
3. Sơ đồ thiết kế thực nghiệm 39
4. Thiết kế mô hình 43
4.1. Kiến trúc mô hình 43
4.2. Huấn luyện mô hình 44
5. Kết quả 46
5.1. Quá trình huấn luyện 46
5.2. Kết quả trên từng tập test 47
CHƯƠNG V: KẾT LUẬN 50
1. Chủ đề và mục tiêu 50
2. Tóm tắt các công việc đã thực hiện 50
3. Ưu điểm của giải pháp 51
4. Hạn chế và thách thức 51
5. Hướng phát triển trong tương lai 51
TÀI LIỆU THAM KHẢO 53

MỤC LỤC HÌNH ẢNH

Hình 1. Minh hoạ các đọc dữ liệu 13

Hình 2. Ví dụ parse tree 20

Hình 3. Ví dụ AST 21

Hình 4. Ví dụ một mạng xã hội nhỏ (Karate Club) 25

Bảng 1. Bảng so sáng GCN và GAT 31

Hình 5. Ví dụ cấu trúc đồ thị AST (độ sâu giới hạn ≤2) trích từ bộ dữ liệu Python150 ... 32

Hình 6. Biểu đồ phân bổ số lượng AST theo class 33

Hình 7. Thống kê nhanh dựa vào 10000 AST đầu 34

Bảng 2. Bảng thống kê top 10 loại node thường gặp 34

Hình 8. Các thư viện cần thiết 35

Hình 9. Mở file và đọc từng dòng 35

Hình 10. Xác định cấu trúc AST 36

Hình 11. Loại bỏ AST quá nhỏ 36

Hình 12. Tạo ma trận cạnh 36

Hình 13. Chuyển sang tensor PyTorch 36

Hình 14. Thu thập node 37

Hình 15. Tạo mapping 37

Hình 16. Tạo tensor 37

Hình 17. Đóng gói 37

Hình 18. Gắn nhãn 37

Hình 19. Over/Under-sampling 38

Hình 20. Kết quả balance 38
Hình 21. Hàm thống kê ban đầu 39
Hình 22. Kết quả thống kê trên tập kiểm thử 39
Hình 23. Sơ đồ tổng quát hệ thống thực nghiệm 40
Hình 24. Pipeline chuyển đổi AST thành đồ thị 41
Hình 25. Kiến trúc mạng GNN kết hợp GCN + GAT 42
Hình 26. Sơ đồ kiến trúc 43
Hình 27. Hàm huấn luyện 45
Hình 28. Biểu đồ biến đổi qua từng epochs 46
Hình 29. Ma trận nhầm lẫn 48

CHƯƠNG I: TỔNG QUAN

1. Giới thiệu đề tài

Trong phát triển phần mềm, độ phức tạp của mã nguồn đóng vai trò quan trọng, ảnh hưởng trực tiếp đến khả năng bảo trì và hiệu suất của chương trình. Mã nguồn càng phức tạp thì càng khó hiểu, khó kiểm thử và dễ phát sinh lỗi. Thông thường, độ phức tạp được đánh giá dựa vào các chỉ số như độ phức tạp cyclomatic (đếm số đường đi độc lập trong mã nguồn) hoặc độ phức tạp thuật toán (ước lượng mức độ tốn tài nguyên khi xử lý dữ liệu lớn). Tuy nhiên, những phương pháp truyền thống này thường phải phân tích thủ công hoặc dựa trên kinh nghiệm, gây khó khăn cho việc tự động hóa.

Gần đây, sự phát triển của học sâu đã mở ra một hướng tiếp cận mới cho bài toán phân loại độ phức tạp mã nguồn. Trong đó, mạng nơ-ron đồ thị (Graph Neural Networks - GNN) kết hợp với cây cú pháp trừu tượng (Abstract Syntax Tree - AST) đang nổi lên như một phương pháp hiệu quả. AST biểu diễn mã nguồn thành cấu trúc cây, trong đó các nút là các thành phần cú pháp và các cạnh là quan hệ cha–con. Việc áp dụng GNN trên AST giúp mô hình học sâu nắm bắt được cấu trúc sâu sắc của mã nguồn, mang lại kết quả vượt trội hơn so với các phương pháp truyền thống.

1.1. Lý do chọn đề tài

Chúng em chọn bài toán phân loại AST theo kích thước như một proof of concept để chứng minh GNN có thể học được cấu trúc code Python hiệu quả. Đây là bước đầu tiên trong việc áp dụng GNN cho các tác vụ phân tích code phức tạp hơn như phát hiện lỗi hay đánh giá chất lượng code. Kết quả kiểm thử accuracy cho thấy tiềm năng lớn của hướng tiếp cận này.

Với sự phát triển nhanh chóng của trí tuệ nhân tạo và học máy, việc đánh giá độ phức tạp của mã nguồn có thể được giải quyết một cách tự động và hiệu quả hơn. Trong đó, mạng nơ-ron đồ thị (Graph Neural Networks - GNN) là một hướng tiếp cận rất phù hợp vì bản thân mã nguồn (đặc biệt là cấu trúc AST của mã nguồn) có thể biểu diễn tự nhiên dưới dạng đồ thị.

Khác với các phương pháp truyền thống vốn coi mã nguồn là văn bản thông thường hay dùng các đặc trưng được tạo thủ công, GNN cho phép mô hình trí tuệ nhân tạo nhìn trực tiếp vào cấu trúc của đồ thị AST. Điều này giúp mô hình hiểu rõ hơn cách các thành phần trong mã nguồn tương tác với nhau và nhận diện được các đặc trưng phức tạp của chương trình.

Việc sử dụng AST của ngôn ngữ Python cũng rất có lợi, bởi Python hiện đang là ngôn ngữ phổ biến và dễ phân tích cú pháp, đồng thời có sẵn nguồn dữ liệu lớn như bộ dữ liệu Python-150K gồm hàng nghìn dự án mã nguồn mở.

Tóm lại, việc chọn hướng nghiên cứu này nhằm đáp ứng nhu cầu thực tế về đánh giá tự động độ phức tạp của mã nguồn, đồng thời tận dụng các tiến bộ mới nhất của trí tuệ nhân tạo, cụ thể là mạng nơ-ron đồ thị trên cấu trúc AST.

1.2. Mục tiêu đề tài

Mục tiêu chính của đề tài là xây dựng một mô hình trí tuệ nhân tạo sử dụng mạng nơ-ron đồ thị (GNN) kết hợp với cấu trúc cây cú pháp trừu tượng (AST) để phân loại tự động mức độ phức tạp của các đoạn mã Python. Để đạt được mục tiêu tổng quát trên, đề tài cụ thể hóa các mục tiêu sau:

- **Nghiên cứu và đề xuất mô hình GNN trên AST**:
  Tìm hiểu và xây dựng một mô hình học sâu sử dụng mạng nơ-ron đồ thị, đặc biệt tập trung vào hai kiến trúc phổ biến là Graph Convolutional Networks (GCN) và Graph Attention Networks (GAT). Các kiến trúc này sẽ được thử nghiệm để đánh giá hiệu quả trong việc biểu diễn và học cấu trúc đồ thị từ AST của mã Python.

- **Xác định cách thức gán nhãn và xây dựng bộ dữ liệu**:
  Xây dựng một tiêu chí rõ ràng và khoa học để gán nhãn mức độ phức tạp cho các đoạn mã Python, như độ phức tạp cấu trúc (số lượng nút AST, độ sâu của cây) hoặc độ phức tạp thuật toán (như O(1), O(n), O(n log n), $O(n^2)$). Trên cơ sở đó, tạo ra một bộ dữ liệu lớn và đầy đủ từ Python150k, phục vụ quá trình huấn luyện và đánh giá mô hình.

- **Đánh giá và so sánh hiệu quả của mô hình**:
  Tiến hành huấn luyện mô hình trên bộ dữ liệu Python150k đã được gán nhãn và đánh giá hiệu quả thông qua các chỉ số độ chính xác (accuracy), độ phủ (recall), độ chính xác (precision), và chỉ số F1-score. Từ kết quả thu được, so sánh hai kiến trúc GCN và GAT để phân tích và lựa chọn mô hình tốt nhất cho bài toán.

- **Đề xuất giải pháp ứng dụng thực tế**:
  Trên cơ sở mô hình hiệu quả nhất, đề xuất phương án triển khai mô hình vào các công cụ phát triển phần mềm, giúp lập trình viên phát hiện sớm và kiểm soát độ phức tạp của mã nguồn, hoặc hỗ trợ việc đánh giá mã nguồn tự động trong giáo dục lập trình và các hệ thống trực tuyến.

1.3. Loại bài toán xử lý

Xét dưới góc độ học máy, bài toán được xử lý thuộc dạng phân loại có giám sát (supervised classification). Cụ thể, đây là bài toán phân loại đa lớp (multi-class classification), trong đó mỗi mẫu dữ liệu đầu vào (một đoạn mã nguồn Python) sẽ được phân loại vào một trong các lớp mức độ phức tạp đã được xác định trước. Các lớp này có thể là:

- Phân loại theo mức độ cấu trúc đơn giản:
  Chia các đoạn mã Python thành ba nhóm rõ ràng:
  - Đơn giản (simple): Cấu trúc AST ngắn gọn, ít nút, cấu trúc không có hoặc rất ít vòng lặp, điều kiện.
  - Trung bình (medium): Có cấu trúc vừa phải, AST phức tạp vừa phải với một số vòng lặp, điều kiện, hay hàm con.
  - Phức tạp (complex): Cấu trúc AST lớn, chứa nhiều nút, độ sâu lớn, nhiều vòng lặp lồng nhau, cấu trúc điều kiện phức tạp hoặc đệ quy.
- Phân loại theo mức độ phức tạp thuật toán:
  Các lớp độ phức tạp thuật toán như:
  - O(1): Thời gian thực hiện không phụ thuộc kích thước đầu vào.
  - O(log n): Tăng chậm theo kích thước đầu vào.
  - O(n): Tăng tuyến tính theo kích thước đầu vào.
  - O(n log n): Tăng nhanh hơn tuyến tính một chút.
  - O(n^2): Tăng rất nhanh theo bình phương kích thước đầu vào hoặc các cấp độ phức tạp cao hơn.

Đầu vào của bài toán là cây AST được xây dựng từ mã nguồn Python. Đây là cấu trúc dạng đồ thị đặc biệt (dạng cây có hướng), phù hợp để xử lý bằng mô hình mạng nơ-ron đồ thị (GNN). Quá trình phân loại diễn ra như sau:

- Bước đầu tiên, mỗi đoạn mã Python được chuyển đổi thành đồ thị AST.
- Tiếp theo, mỗi nút AST được mã hóa thành một vector đặc trưng ban đầu dựa trên loại cú pháp.

- Các vector này sẽ được truyền qua mạng nơ-ron đồ thị (GNN). GNN sẽ thực hiện truyền và tổng hợp thông tin từ các nút lân cận theo cấu trúc đồ thị, qua đó học được đặc trưng tổng quát của toàn bộ đoạn mã nguồn.
- Cuối cùng, một vector đặc trưng đại diện cho toàn bộ đoạn mã (đồ thị) được dùng để dự đoán lớp mức độ phức tạp của đoạn mã.

Tổng kết lại, bài toán đặt ra là bài toán phân loại đồ thị dựa trên mạng nơ-ron đồ thị với mục tiêu dự đoán chính xác mức độ phức tạp của các đoạn mã Python dựa trên cấu trúc cây cú pháp AST.

2. Mô tả dữ liệu

Bộ dữ liệu Python150k bao gồm 150.000 cây AST (Abstract Syntax Tree) của mã nguồn Python được trích xuất tự động. Dữ liệu được chia thành hai tệp JSON: python100k_train.json (100.000 AST dùng để huấn luyện, ~2.5GB) và python50k_eval.json (50.000 AST dùng để đánh giá, ~1.2GB). Mỗi dòng trong các tệp JSON này tương ứng với một AST của một file mã Python. AST được lưu dưới dạng một danh sách các node JSON (đối tượng JSON).

Cấu trúc mỗi node: Mỗi node (nút) trong AST được biểu diễn bởi một object JSON với các trường chính:

- **type (bắt buộc)**: Kiểu cú pháp của node hiện tại (ví dụ: Module, Assign, NameLoad, NameStore, Call, v.v.). Đây là tên của thành phần cú pháp theo grammar Python.
- **value (tùy chọn)**: Giá trị nội dung của node nếu có. Trường này xuất hiện ở các node là hằng số hoặc định danh, ví dụ value có thể là tên biến, chuỗi ký tự, số, v.v.
- **children (tùy chọn)**: Danh sách các chỉ số (index) của các node con (nếu có). Chỉ số đánh số node trong cùng danh sách AST, bắt đầu từ 0 cho node đầu tiên. Thứ tự trong children thể hiện quan hệ cha–con theo đúng cấu trúc cây chương trình. Node gốc (root) thường là node loại Module ở chỉ số 0, và nó có các chỉ số con trỏ đến các node con cấp 1, v.v.

Nói cách khác, mỗi AST được "phẳng hóa" thành một list, trong đó quan hệ cha–con được thể hiện qua việc node cha chứa danh sách index children trỏ tới vị trí node con trong list đó. Nếu một node không có trường children (hoặc children rỗng) thì đó là node lá (leaf node).

Ví dụ: với đoạn code Python đơn giản sau đây:

x = 7

print x+1

AST của chương trình trên được biểu diễn như sau:


```
[
  {"type": "Module", "children": [1, 4]},
  {"type": "Assign", "children": [2, 3]},
    {"type": "NameStore", "value": "x"},
    {"type": "Num", "value": "7"},
  {"type": "Print", "children": [5]},
    {"type": "BinOpAdd", "children": [6, 7]},
      {"type": "NameLoad", "value": "x"},
      {"type": "Num", "value": "1"}
]
```


Trong ví dụ trên, node gốc có type: "Module" (module của chương trình) với children: [1, 4] nghĩa là node ở chỉ số 1 và 4 trong danh sách là con của Module. Thật vậy:

- Node [1] là "Assign" (lệnh gán) có hai node con [2] và [3] (tương ứng là "NameStore" với value "x", và "Num" với value "7") – biểu diễn cho x = 7.
- Node [4] là "Print" (câu lệnh in ra) có một node con [5] ("BinOpAdd" – phép cộng). "BinOpAdd" lại có hai node con [6] và [7] (lần lượt là "NameLoad" giá trị "x", và "Num" giá trị "1"), tương ứng biểu thức x+1.

Qua đó, ta thấy mỗi node chứa thông tin về loại cú pháp và liên kết đến node con, cho phép tái tạo cấu trúc cây AST đầy đủ của chương trình. Dạng JSON phẳng này thuận tiện cho việc lưu trữ và xử lý hàng loạt cây AST (mỗi dòng là một cây).

Đọc thử một AST: Để minh họa cách đọc cấu trúc dữ liệu, ta có thể sử dụng Python để đọc một dòng JSON (một AST) từ file và kiểm tra nội dung:

```python
import json

TRAIN_DATA_PATH = '/kaggle/input/py150k-1/python100k_train.json'

# Đọc một dòng đầu tiên (một AST)
with open(TRAIN_DATA_PATH, 'r', encoding='utf-8') as f:
    first_line = f.readline().strip()

ast = json.loads(first_line)
print("Số node trong AST đầu tiên:", len(ast))
print("Node đầu [0]:", ast[0])
print("Node cuối [-1]:", ast[-1])
```

```
Số node trong AST đầu tiên: 116
Node đầu [0]: {'type': 'Module', 'children': [1, 3, 5, 7, 9, 11]}
Node cuối [-1]: {'type': 'NameLoad', 'value': 'NotFound'}
```

Hình 2. Minh hoạ các đọc dữ liệu

Kết quả trả về sẽ cho biết số lượng node của AST đầu tiên và hiển thị nội dung node gốc và node lá cuối cùng. Ta có thể thấy node gốc thường có type là "Module", và các node lá sẽ có type tương ứng (ví dụ "NameLoad", "Num", v.v.) kèm value nếu là hằng hoặc định danh.

3. Mô hình bài toán

Chúng ta xây dựng bài toán phân loại AST dưới dạng phân loại đồ thị. Cụ thể, mỗi chương trình Python sẽ được biểu diễn bằng cây trừu tượng cú pháp (AST) và chuyển đổi thành một đồ thị vô hướng graph $G = (V, E)$ theo quy tắc:

- Đỉnh (node) $v \in V$: mỗi thành phần cú pháp (syntactic construct) trong AST—chẳng hạn một biểu thức, toán tử, tên biến, hằng số, câu lệnh điều khiển—được xem như một đỉnh. Mỗi đỉnh được gán một vector đặc trưng ban đầu x_v dựa trên loại node (ví dụ one-hot encoding theo nhãn type của node).
- Cạnh (edge) $(u, v) \in E$: tồn tại nếu và chỉ nếu node u là cha của node v trong AST gốc. Để mô hình GNN có thể lan truyền thông tin hai chiều, ta sẽ thêm cả cạnh (v, u), biến đồ thị thành vô hướng.

Mục tiêu của bài toán là dự đoán nhãn lớp $yG \in \{0,1,2\}$ cho toàn bộ đồ thị G (tương đương với cả một AST), trong đó ba lớp được định nghĩa theo kích thước của AST:

- Class 0: AST "nhỏ" (< 20 node)
- Class 1: AST "vừa" ($20–49$ node)
- Class 2: AST "lớn" (≥ 50 node)

Để giải bài toán này, ta sử dụng các mô hình Graph Neural Network (GNN) như Graph Convolutional Network (GCN) hoặc Graph Attention Network (GAT). Các bước chính gồm:

1. Chuẩn bị dữ liệu:
    - Chuyển mỗi AST từ JSON thành đồ thị NetworkX, gán node feature là mã hóa loại cú pháp, xây dựng danh sách cạnh cha–con.
    - Gán nhãn class dựa vào số lượng node của AST.
2. Xây dựng mô hình GNN:
    - Mỗi layer GNN cập nhật vector đại diện của node bằng cách tổng hợp thông tin từ các node kề (cha và con), theo công thức tích chập đồ thị hoặc attention trên cạnh.
    - Sau L layer, các vector node chứa thông tin lan truyền từ $L - hop$ lân cận.
3. Pooling & Phân loại:
    - Dùng cơ chế graph-level readout (ví dụ global mean pooling) để gom thông tin từ tất cả node thành một vector biểu diễn toàn đồ thị.
    - Qua một hoặc nhiều lớp fully-connected, output là phân phối xác suất trên 3 lớp, tối ưu hóa loss cross-entropy.
4. Đánh giá:
    - Huấn luyện trên tập python100k_train.json, đánh giá trên tập python50k_eval.json, đo lường metrics như accuracy, F1-score trên từng lớp, đặc biệt lưu ý việc dữ liệu mất cân bằng (AST lớn chiếm ưu thế).

Qua mô hình này, bài toán phân loại AST được chuẩn hóa thành một bài toán supervised graph classification, cho phép khai thác cấu trúc cú pháp sâu của mã nguồn Python thông qua sức mạnh biểu diễn và lan truyền thông tin của GNN.

CHƯƠNG II: CƠ SỞ LÝ THUYẾT

Mô hình sử dụng Graph Convolutional Network (GCN) – một loại mạng nơ-ron đồ thị phổ biến – kết hợp với lớp Graph Attention Network (GAT) ở tầng cuối. GCN là kiến trúc GNN triển khai phép tổng hợp (aggregate) thông tin từ láng giềng của mỗi đỉnh trên đồ thị. Cụ thể, các lớp GCN cập nhật biểu diễn của một nút bằng cách truyền thông điệp từ các nút kề nó, tức là lấy trung bình có trọng số đặc trưng của các hàng xóm lân cận. Nhờ đó, đặc trưng của mỗi nút sau một lớp GCN đã bao gồm thông tin của một bước láng giềng.

1. Graph Convolutional Network (GCN)

1.1. Giới thiệu tổng quan

GCN (Graph Convolutional Network) là một kiến trúc mạng nơ-ron đồ thị được Kipf & Welling giới thiệu năm 2017, nổi bật với khả năng tổng hợp thông tin từ lân cận của mỗi nút trong đồ thị. GCN hoạt động tương tự phép tích chập trên đồ thị, giúp mạng học được biểu diễn ẩn của mỗi nút dựa trên đặc trưng của nó và các nút kề nó. Đặc điểm chính của GCN là đơn giản và hiệu quả: mô hình lan truyền thông tin cục bộ (chỉ trong phạm vi láng giềng gần) và sử dụng trọng số chia sẻ (cùng một ma trận trọng số cho mọi nút ở mỗi lớp. Nhờ đó, GCN có số tham số không phụ thuộc số nút và cạnh, giúp mô hình dễ dàng mở rộng cho các đồ thị lớn. GCN phát huy điểm mạnh trong các tác vụ phân loại nút và đồ thị bán giám sát, khi mà nó khai thác được khuynh hướng các nút liên kết thường có nhãn giống nhau (hiện tượng "khuếch tán nhãn" trong đồ thị).

1.2. Nguyên lý hoạt động

GCN thực hiện lan truyền thông tin giữa các nút thông qua việc trung bình có trọng số đặc trưng của nút với lân cận. Ở mỗi lớp GCN, mỗi nút i sẽ cập nhật đặc trưng $H_i^{(l+1)}$ bằng cách lấy tổng có trọng số và chuẩn hóa các đặc trưng $H_j^{(l)}$ của các nút j lân cận (bao gồm cả nút i chính nó) rồi áp dụng hàm kích hoạt (ví dụ ReLU). Cụ thể, Kipf & Welling (2017) đề xuất công thức lan truyền đặc trưng mỗi lớp như sau:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

Trong đó $H^{(l)}$ là ma trận đặc trưng của tất cả các nút ở lớp l (hàng i tương ứng $H_i^{(l)}$ $W^{(l)}$ là ma trận trọng số học được ở lớp l, $\tilde{A} = A + I$ là ma trận kề của đồ thị sau khi thêm self-loop (mỗi nút kết nối với chính nó), $\tilde{D}$ là ma trận bậc (degree) tương ứng với $\tilde{A}$. Toán hạng $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ chính là ma trận kề đã chuẩn hóa (normalized adjacency matrix) giúp điều chỉnh tỷ lệ đóng góp của nút có nhiều láng giềng so với nút ít láng giềng. Nói cách khác,

mỗi nút sẽ lấy trung bình đặc trưng của các hàng xóm của nó (sau khi đã thêm kết nối chính nó để bảo toàn thông tin gốc). Kết quả sau phép tổng hợp này được nhân với ma trận trọng số $W^{(l)}$ và đưa qua hàm kích hoạt σ (ví dụ hàm ReLU) để thu được đặc trưng mới $H^{(l+1)}$. Qua nhiều lớp GCN chồng liên tiếp, thông tin từ các nút xa hơn (ví dụ nút bậc hai, bậc ba trong đồ thị) cũng dần dần được lan truyền đến mỗi nút hiện tại. Mô hình GCN thường được huấn luyện bằng lan truyền ngược lỗi tương tự mạng nơ-ron truyền thống, với hàm mất mát giám sát (ví dụ cross-entropy cho phân loại) áp dụng lên các node đích.

1.3. Ứng dụng

Graph Convolutional Network (GCN) đã chứng minh hiệu quả cao trong việc khai thác cấu trúc cây cú pháp trừu tượng (AST) của chương trình để giải quyết nhiều bài toán phân tích mã nguồn. Bằng cách biến mỗi AST thành một đồ thị trong đó các nút đại diện cho thành phần cú pháp (toán tử, biến, khối lệnh…) và các cạnh thể hiện quan hệ cha–con hoặc các kết nối ngữ nghĩa bổ sung, GCN liên tiếp thực hiện phép chập đồ thị để tổng hợp thông tin từ các node lá lên node cha, từ đó tự động học được biểu diễn ẩn (graph-level embedding) phản ánh ngữ cảnh cấu trúc. Trên cơ sở embedding này, GCN đã được ứng dụng thành công cho phân loại chức năng đoạn mã (code classification), phát hiện lỗ hổng bảo mật (vulnerability detection) thông qua mô hình hóa các mẫu đồ thị bất thường, phát hiện mã trùng lặp (code clone detection) bằng cách so sánh embedding AST, cũng như trích xuất và gợi ý tên hàm (function name prediction) và tóm tắt mã (code summarization) khi kết hợp với mô hình sinh ngôn ngữ. Ngoài ra, GCN còn hỗ trợ đánh giá độ phức tạp và phong cách code (complexity and style assessment) bằng cách bổ sung các đặc trưng như độ sâu nút và số nhánh con vào đồ thị, từ đó cung cấp công cụ tự động đánh giá và tối ưu chất lượng phần mềm.

2. Graph Attention Network (GAT)

2.1. Giới thiệu tổng quan

GAT (Graph Attention Network) là một kiến trúc GNN do Veličković và cộng sự đề xuất năm 2018, tích hợp cơ chế attention (tập trung) vào việc tổng hợp thông tin trên đồ thị. Tương tự GCN, mô hình GAT cũng cập nhật đặc trưng của một nút dựa trên láng giềng của nó, nhưng điểm khác biệt chính là GAT học hệ số trọng số thích ứng cho từng cạnh kết nối thay vì cố định bởi ma trận kề đã chuẩn hóa. Nói cách khác, GAT có thể tự điều chỉnh mức độ ảnh hưởng của mỗi nút hàng xóm thông qua hệ số chú ý (attention coefficient) học được, thay vì coi tất cả láng giềng đóng góp bình đẳng. Đặc điểm này cho phép GAT linh hoạt hơn: mô hình có thể tập trung vào những hàng xóm quan trọng và giảm ảnh hưởng của hàng xóm kém liên quan cho nhiệm vụ cần học. Nhờ cơ chế attention, GAT giải quyết được hạn chế của GCN khi GCN luôn chia sẻ đồng đều (hoặc theo bậc) trọng số cho láng

giềng. GAT còn hỗ trợ multi-head attention – sử dụng nhiều "đầu chú ý" độc lập – giúp ổn định quá trình học và tăng khả năng biểu diễn. Với các ưu điểm này, GAT đã cho thấy hiệu quả vượt trội trong nhiều bài toán đồ thị, đặc biệt trong bối cảnh mô hình cần nhấn mạnh cấu trúc quan hệ phức tạp (ví dụ, trong mạng xã hội, không phải tất cả bạn bè của một người đều quan trọng như nhau; hay trong mô hình ngôn ngữ, một từ có thể chú ý đến từ khác quan trọng hơn).

2.2. Nguyên lý hoạt động

GAT thực hiện lan truyền thông tin có trọng số thích ứng dựa trên cơ chế tự chú ý trên đồ thị. Với mỗi cạnh (i, j) (nút j là láng giềng của nút i), GAT tính một điểm chú ý e_{ij} phản ánh mức độ "quan trọng" của nút j đối với nút i. Cụ thể, đầu tiên tất cả vector đặc trưng h được biến đổi qua một lớp tuyến tính chung: $h_i' = Wh_i$ (với W là ma trận trọng số học được, dùng chung cho mọi cạnh trong cùng một head). Sau đó, điểm chú ý chưa chuẩn hóa e_{ij} được tính bằng một mạng nơ-ron một lớp (được tham số hóa bởi vector $\boldsymbol{a}$) tác động lên cặp đặc trưng của nút i và j. Công thức thường được dùng là:

$$e_{ij} = \text{LeakyReLU}\left(a^T\left[Wh_i \parallel Wh_{(j)}\right]\right)$$

Trong đó $\parallel$ ký hiệu phép nối vector (concatenation) hai vector đặc trưng đã qua biến đổi của nút i và j, và a là vector trọng số của cơ chế attention. Hàm LeakyReLU (với slope âm thường lấy 0.2) được dùng làm hàm kích hoạt cho tầng tính điểm chú ý này nhằm tránh vùng chết của ReLU. Tiếp theo, các hệ số chú ý được chuẩn hóa qua hàm softmax trên tập láng giềng của i để thu được hệ số trọng số cuối cùng α_{ij}:

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

với $\mathcal{N}(i)$ là tập các nút láng giềng của i (thường bao gồm cả i nếu có self-loop). Như vậy, α_{ij} đóng vai trò tương tự ma trận kề đã chuẩn hóa trong GCN nhưng được học một cách động tùy theo đặc trưng nút – đảm bảo $\sum_{j\in\mathcal{N}(i)}\alpha_{ij} = 1$ nên mỗi α_{ij} có thể hiểu là tỷ trọng đóng góp của nút j vào nút i. Cuối cùng, nút i cập nhật đặc trưng bằng tổng có trọng số attention của các đặc trưng láng giềng:

$$h_i^{(l+1)} = \sigma\left(\sum_{j\in\mathcal{N}(i)}\alpha_{ij}W\,h_j^{(l)}\right)$$

trong đó $Wh_j^{(l)}$ là đặc trưng của nút j đã qua biến đổi ở lớp l, và σlà hàm kích hoạt phi tuyến (ví dụ ELU hoặc ReLU). Nếu dùng multi-head attention với K head, phép tính trên

được lặp lại độc lập cho mỗi head k (với trọng số W^k, vector a^k riêng). Kết quả có thể được nối tất cả các head (concatenate) để tạo thành $h_i^{(l+1)}$ cuối cùng, hoặc trung bình các head (đặc biệt ở lớp cuối để giữ cố định kích thước đầu ra). Như vậy, GAT cho phép mô hình học cách *chú ý* đến các đặc trưng hữu ích: cạnh nào mang thông tin quan trọng sẽ được gán trọng số cao, ngược lại có trọng số thấp, điều này giúp cải thiện hiệu suất mô hình trên nhiều cấu trúc đồ thị phức tạp.

2.3. Ứng dụng

Trong phân loại AST, Graph Attention Network (GAT) cho phép mô hình tự động nhấn mạnh những phần cú pháp quan trọng của cây. Thay vì xem mọi node con đóng góp đồng đều vào biểu diễn cha, GAT học được hệ số chú ý α_{ij} trên mỗi cạnh, từ đó tập trung nhiều hơn vào những node con mang tính quyết định (ví dụ các biểu thức điều kiện, lời gọi hàm đặc trưng) và giảm trọng số cho các node ít liên quan. Kết quả là embedding cấp node sau khi qua GAT chứa đựng thông tin trọng yếu của AST, giúp cải thiện độ chính xác phân loại đoạn mã.

Cơ chế multi-head attention trong GAT càng tăng cường tính biểu diễn cho AST: mỗi head có thể chú ý đến một khía cạnh khác nhau của cấu trúc cú pháp (như toán tử, khối lệnh, hay tham số). Khi ghép (concatenate) hoặc trung bình các head, mô hình kết hợp được đa chiều thông tin quan trọng. Điều này đặc biệt hữu ích khi phân biệt các dạng AST có cấu trúc phức tạp hoặc tương đồng về mặt số lượng node nhưng khác nhau về ngữ nghĩa.

GAT cũng đem lại tính diễn giải cho phân loại AST: thông qua ma trận chú ý, ta có thể kiểm tra xem mô hình đã tập trung vào những node nào khi đưa ra quyết định phân lớp. Ví dụ, khi dự đoán một hàm là "thuật toán tìm kiếm" hay "xử lý chuỗi", hệ số α\alphaα cao ở các node kiểu ForLoop hoặc StringConcat sẽ khẳng định nhánh cú pháp này là yếu tố chính. Nhờ đó, GAT không chỉ tăng hiệu năng mà còn hỗ trợ kiểm thử chất lượng mô hình.

Cuối cùng, việc sử dụng GAT trên AST giúp giảm nhiễu từ các node ít liên quan (như comment, literal không quan trọng) và tăng khả năng tổng quát khi có biến thể về cách viết code. Các hệ số chú ý học được điều chỉnh linh hoạt với từng đầu vào, giúp mô hình phân loại AST vẫn chính xác ngay cả khi cấu trúc cây phức tạp hoặc có nhiều nhánh phụ không liên quan.

3. Cấu trúc Abstract Syntax Tree (AST)

3.1. Giới thiệu tổng quan

Abstract Syntax Tree (AST) – tạm dịch là cây cú pháp trừu tượng – là một cấu trúc dữ liệu dạng cây dùng để biểu diễn cấu trúc cú pháp của chương trình hoặc đoạn mã nguồn.

Mỗi nút trong cây AST tương ứng với một thành phần (construct) trong mã nguồn, ví dụ như một biểu thức, một câu lệnh, một toán tử, v.v. AST được gọi là "trừu tượng" vì nó lược bỏ những chi tiết cú pháp cụ thể không cần thiết của mã nguồn thật, chỉ giữ lại những thông tin cốt lõi về cấu trúc chương trình. Chẳng hạn, các dấu ngoặc dùng để nhóm biểu thức không xuất hiện trực tiếp trong AST; thay vào đó, cấu trúc cây tự nó đã thể hiện thứ tự ưu tiên và phạm vi của các toán tử, nên không cần nút riêng cho dấu ngoặc. Nhờ tập trung vào cấu trúc và nội dung quan trọng, AST giúp cho trình biên dịch và các công cụ phân tích mã hiểu được ý nghĩa cú pháp của chương trình một cách dễ dàng hơn.

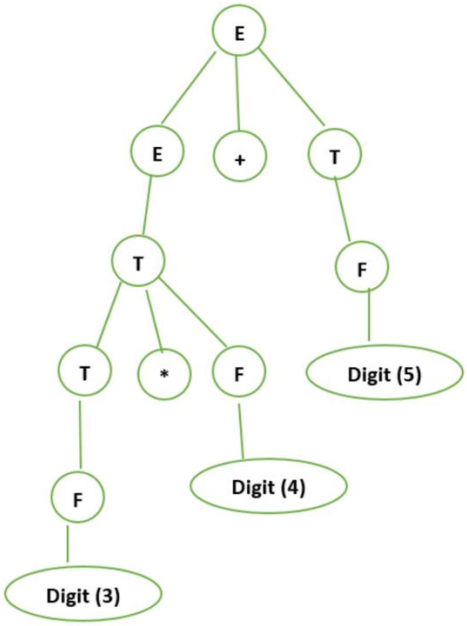
AST có vai trò rất quan trọng trong chuỗi xử lý biên dịch và phân tích mã nguồn. Thông thường, mã nguồn sẽ trải qua các bước: phân tích từ vựng (lexical analysis) để tách chuỗi mã thành các token, sau đó phân tích cú pháp (parsing) để dựng nên cây cú pháp. Kết quả của giai đoạn parsing chính là cây AST – đây là sản phẩm đầu ra của bước phân tích cú pháp trong trình biên dịch. AST đóng vai trò làm biểu diễn trung gian của chương trình, là cầu nối giữa phần phân tích cú pháp và các bước tiếp theo như phân tích ngữ nghĩa, tối ưu hóa, và sinh mã máy. Thông qua AST, trình biên dịch có thể thực hiện kiểm tra kiểu dữ liệu, ràng buộc ngữ nghĩa, và tối ưu mã một cách có hệ thống dựa trên cấu trúc chương trình thay vì trên văn bản thô của mã nguồn. Tóm lại, AST giúp chuyển từ mã nguồn dạng văn bản sang một dạng cấu trúc dễ xử lý hơn, tạo nền tảng cho mọi phân tích và chuyển đổi về sau trong công cụ phát triển phần mềm.

3.2. Nguyên lý hoạt động

Quá trình tạo ra AST. Để sinh ra AST từ mã nguồn, trình biên dịch hoặc công cụ phân tích sẽ thực hiện hai bước chính: (1) Phân tích từ vựng (lexing) để chuyển mã nguồn thành danh sách các token (như từ khóa, định danh, toán tử, hằng số...), và (2) Phân tích cú pháp (parsing) để tổ chức các token đó theo luật ngữ pháp của ngôn ngữ, xây dựng nên cây cú phá. Kết quả ban đầu của phân tích cú pháp thường là một cây phân tích cụ thể (parse tree hay concrete syntax tree) biểu diễn đầy đủ mọi chi tiết cú pháp của mã nguồn. Tuy nhiên, parse tree này thường được chuyển hóa hoặc tinh giản thành AST – bằng cách loại bỏ những chi tiết không cần thiết như dấu ngoặc, dấu chấm phẩy, từ khóa cấu trúc, v.v. và chỉ giữ lại cấu trúc ngữ nghĩa quan trọng. Nói cách khác, AST là phiên bản đơn giản hóa của parse tree, tập trung vào cấu trúc logic của chương trình thay vì chi tiết cú pháp bề mặt. AST thường được xây dựng sao cho mỗi nút trong cây thể hiện một ý nghĩa ngữ pháp (ví dụ: nút biểu diễn phép cộng hoặc phép gán), các nút lá có thể là hằng số hoặc biến, và các nhánh con là thành phần cấu tạo nên nút cha (ví dụ: nút phép cộng sẽ có hai con là biểu thức bên trái và bên phải). Quá trình chuyển từ parse tree sang AST giúp loại bỏ những nút trung gian dư thừa (phục vụ cho văn phạm nhưng không mang ý nghĩa khi thực thi), đồng thời gắn kết các thành phần liên quan trực tiếp với nhau. Kết quả cuối cùng là một cây cấu

trúc gọn gàng, phản ánh đúng nghĩa của chương trình và sẵn sàng cho các bước phân tích tiếp theo.

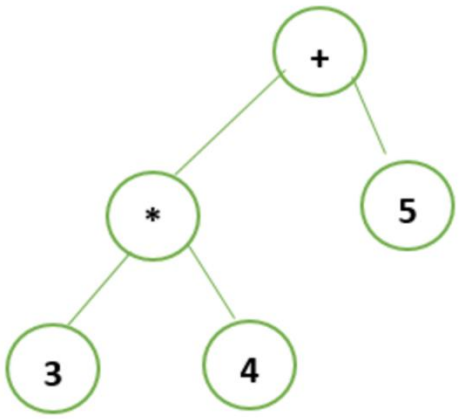
*Ví dụ parse tree***:** Hình dưới thể hiện *cây phân tích (parse tree)* đầy đủ cho biểu thức 3 * 4 + 5. Gốc cây là ký hiệu <E> (biểu diễn một *Expression* theo văn phạm). Mỗi nhánh con triển khai một luật ngữ pháp: chẳng hạn <E> được tách thành <E> + <T> để thể hiện phép cộng, và tiếp tục <T> tách thành <T> * <F> cho phép nhân, v.v. Các lá của cây là các *terminal* như số 3, 4, 5 (được dán nhãn kiểu Digit(3), Digit(4),...). Parse tree này bao gồm tất cả các ký hiệu không kết thúc (non-terminal) như <E>, <T>, <F>, cũng như các toán tử +, * dưới dạng nút, phản ánh đúng từng bước áp dụng luật văn phạm. Như có thể thấy, cây phân tích khá chi tiết và *sâu*, bao gồm cả những nút trung gian như <F> hoặc Digit vốn chỉ phục vụ cho quy tắc ngữ pháp mà không đóng góp trực tiếp vào ý nghĩa tính toán của biểu thức.



Hình 3. Ví dụ parse tree

Ví dụ AST: Khác với parse tree, Cây AST cho cùng biểu thức 3 * 4 + 5 được đơn giản hóa chỉ còn các nút thể hiện toán tử và toán hạng cốt lõi của biểu thức. Gốc cây AST là toán tử + (do phép cộng là phép tính ở cấp cao nhất của biểu thức này), với hai nút con tương ứng với các toán hạng của phép cộng. Ở bên trái, nút con là toán tử * (phép nhân) cùng hai con của nó là hằng số 3 và 4. Bên phải, nút con là hằng số 5. Không còn các nút cho ký hiệu trung gian như <E>, <T>, <F> hay Digit, và cũng không cần nút riêng cho dấu ngoặc hoặc dấu +, * vì bản thân các nút toán tử đã trực tiếp đại diện cho phép toán đó. Cấu trúc AST này thể hiện đúng thứ tự ưu tiên: phép nhân * nằm dưới phép cộng +, nghĩa là sẽ

được tính toán trước, phù hợp với thứ tự thực hiện trong ngôn ngữ lập trình. Như vậy, so với parse tree, AST nông hơn và dễ hiểu hơn: nó loại bỏ những chi tiết cú pháp rườm rà (dấu ngoặc, ký hiệu văn phạm), chỉ giữ lại những thông tin cần thiết cho việc phân tích và thực thi biểu thức. Điều này minh họa sự khác biệt chính giữa parse tree và AST: parse tree bám sát cú pháp nguồn, còn AST phản ánh cấu trúc ngữ nghĩa một cách cô đọng.



Hình 4. Ví dụ AST

Tóm lại, AST khác biệt với parse tree ở chỗ nó không chứa các chi tiết vụn vặt của cú pháp cụ thể. Một AST sẽ không lưu trữ các ký tự định dạng như khoảng trắng, xuống dòng, hay các ký hiệu đóng/mở khối (như {} trong C, hoặc begin/end trong Pascal), mặc dù các thông tin này có trong mã nguồn và parse tr. Thay vào đó, AST tập trung biểu diễn mối quan hệ cha-con giữa các cấu trúc ngữ pháp: ví dụ, một lệnh if-else trong AST có thể được biểu diễn bằng một nút duy nhất loại IfStatement với ba nhánh con tương ứng điều kiện, nhánh then, và nhánh else (thay vì nhiều nút rời rạc cho từ khóa if, else, dấu ngoặc, dấu hai chấm...). Nhờ sự tinh giản này, AST cung cấp cái nhìn trừu tượng, tập trung vào ý nghĩa của chương trình thay vì cú pháp. Sau khi tạo ra AST, trình biên dịch còn có thể gắn thêm các thông tin bổ trợ vào cây, chẳng hạn như kiểu dữ liệu của biểu thức, giá trị mặc định, hoặc vị trí dòng mã của mỗi nút, nhằm phục vụ cho việc kiểm tra ngữ nghĩa và thông báo lỗi chính xác. Có thể nói, AST là cấu trúc nền tảng linh hoạt: nó vừa đủ chi tiết để biểu diễn chính xác chương trình, vừa đủ trừu tượng để tiện cho các bước phân tích tự động.

3.3. Ứng dụng

Trong các trình biên dịch và thông dịch, AST đóng vai trò trung gian giữa mã nguồn và biểu diễn nội bộ. Sau khi bước phân tích cú pháp hoàn tất, AST được sinh ra để kiểm tra ngữ nghĩa (như tương thích kiểu, phạm vi biến) và làm cơ sở cho các bước tối ưu hóa

mã — ví dụ loại bỏ biểu thức dư thừa hay chuyển đổi vòng lặp. Cuối cùng, trình biên dịch dựa trên AST để sinh ra mã đích hoặc mã máy, đảm bảo thứ tự tính toán và cấu trúc chương trình được bảo toàn.

Các môi trường phát triển tích hợp (IDE) hiện đại liên tục xây dựng và cập nhật AST khi lập trình viên gõ mã. Nhờ đó, IDE có thể cung cấp tính năng đánh dấu cú pháp theo ngữ cảnh, gợi ý tự động tên phương thức hoặc thuộc tính, và điều hướng mã đến định nghĩa biến, hàm một cách chính xác. AST cũng cho phép cảnh báo lỗi cú pháp và sai kiểu ngay tức thì, giúp cải thiện trải nghiệm lập trình và giảm thời gian debug.

Trong phân tích tĩnh (linting và kiểm tra chất lượng), AST là nền tảng để phát hiện các vi phạm quy tắc và mẫu lỗi tiềm ẩn. Các công cụ như ESLint, Pylint hay SonarQube duyệt qua cây AST để nhận biết code smell (hàm quá dài, đoạn mã trùng lặp), lỗi logic (dùng biến chưa khởi tạo) và vấn đề bảo mật (injection, rò rỉ tài nguyên) mà không cần chạy chương trình. Phân tích trên AST mang lại độ chính xác cao hơn so với phân chia cú pháp thuần túy và dễ tích hợp vào quy trình CI/CD.

Đối với refactoring và codemod, AST cho phép thao tác cấu trúc mã an toàn và nhất quán. Khi thực hiện các đổi tên biến, trích xuất hàm, hoặc chuyển đổi vòng lặp thành các lời gọi hàm bậc cao, công cụ refactoring sửa đổi trực tiếp trên cây AST rồi sinh lại mã nguồn. Các script codemod cũng sử dụng AST để áp dụng cùng một chỉnh sửa cho hàng ngàn file, đảm bảo không bỏ sót và giảm thiểu sai sót so với biên tập thủ công.

Trong nghiên cứu và ứng dụng học máy cho mã nguồn, AST là cơ sở để trích xuất đặc trưng cho các mô hình như Graph Neural Networks. Cây AST biểu diễn mối quan hệ cha–con và luồng dữ liệu giữa các node, cho phép mô hình học hiểu cấu trúc và ngữ nghĩa của chương trình. Nhờ đó, các hệ thống học máy có thể thực hiện chính xác các tác vụ như dự đoán tên biến/hàm, phát hiện clone code, nhận diện lỗi logic và tóm tắt chức năng đoạn mã.

4. Các đặc trưng của graph

4.1. GCN

Công thức (Kipf & Welling, 2017):

$$H^{(l+1)} = \sigma\left(\widehat{D^{-\frac{1}{2}}}\hat{A}\widehat{D^{-\frac{1}{2}}}H^{(l)}W^{(l)}\right),$$

Trong đó:

$\tilde{A} = A + I$: ma trận kề đã thêm self–loop, đảm bảo mỗi nút truyền thông tin cho chính nó.

$\widetilde{D}$: ma trận bậc chuẩn hóa $\widetilde{D_{ii}} = \sum_j \widetilde{A_{ij}}$

$H^{(l)}$: ma trận feature tại lớp l, $W^{(l)}$ trọng số học được.

σ: hàm phi tuyến (ReLU, tanh…).

Cơ chế lan truyền: mỗi nút chỉ tính trung bình các feature của các nút lân cận (và chính nó), cân bằng bởi $\widetilde{D^{-1/2}}$

Ưu điểm: tính toán đơn giản, hiệu quả, phù hợp với đồ thị lớn.

Hạn chế: gán trọng số bằng nhau cho tất cả neighbor, không phân biệt tầm quan trọng.

4.2. GAT

Công thức chung (Velicković et al., 2018):

Tính attention score giữa nút i và j:

$$e_{ij} = \text{LeakyReLU}\left(\overrightarrow{a^{\top}}\left[Wh_i \parallel Wh_j\right]\right)$$

Chuẩn hóa scores bằng softmax:

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

Cập nhật feature:

$$h_i' = \sigma\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij}\ Wh_j\right)$$

Cơ chế lan truyền: thay vì trung bình đều, GAT tự học trọng số α_{ij} cho mối quan hệ i–j, làm nổi bật các neighbor quan trọng.

Ưu điểm: linh hoạt, tập trung vào các đường đi thông tin có ý nghĩa; cải thiện hiệu năng trên đồ thị có cấu trúc phức tạp.

Hạn chế: tính toán phức tạp hơn, tốn kém tài nguyên khi số neighbor lớn.

4.3. Độ tương đồng (Similarity)

Sau khi lan truyền, embeddings của các nút/đồ thị phản ánh mức độ tương đồng về cấu trúc và feature:

4.3.1. Node-level similarity

- Cosine similarity giữa hai node embedding :

$$\cos(h_i, h_j) = \frac{h_i^\top h_j}{|h_i||h_j|}$$

- Ứng dụng: phát hiện code clone (đoạn AST giống nhau) bằng cách so sánh embedding của các node hoặc subgraph.

4.3.2. Graph-level similarity

- Sau pooling (mean/sum/max), ta có graph embedding. Tương tự, cosine similarity giữa hai đồ thị:

$$\cos(z_{g_1}, z_{g_2}) = \frac{z_{g_1}^\top z_{g_2}}{|z_{g_1}||z_{g_2}|}$$

- Ứng dụng:
    - Clustering chương trình có cấu trúc AST tương tự.
    - Retrieval: tìm kiếm các AST/mã nguồn có cấu trúc giống với mẫu đầu vào.

4.3.3. Multi-head attention và similarity

- GAT với nhiều head cho phép đa dạng hóa các "góc nhìn" attention, giúp học nhiều biểu diễn khác nhau, từ đó cải thiện độ tương đồng khi pooling.

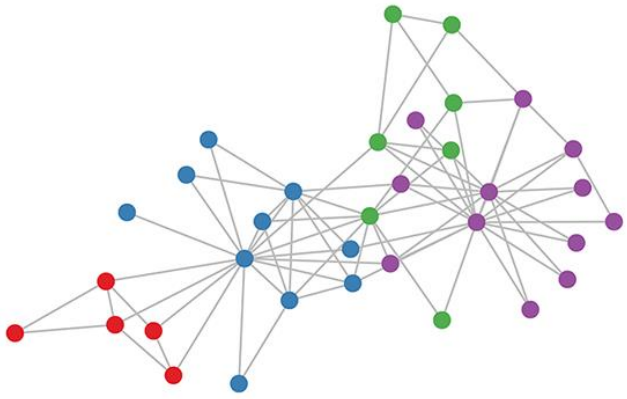
CHƯƠNG III: TÌM HIỂU MỘT NGHIÊN CỨU KHOA HỌC

1. Giới thiệu tổng quan

Mạng xã hội có thể được mô hình hóa dưới dạng đồ thị với các nút (nodes) biểu thị người dùng và các cạnh (edges) biểu thị mối quan hệ (kết bạn, theo dõi) giữa h. Việc phân tích mạng xã hội đòi hỏi các phương pháp khai thác cả thuộc tính của từng người dùng lẫn cấu trúc kết nối phức tạp giữa họ.

Gần đây, Graph Convolutional Network (GCN) – một dạng mạng neural học trên đồ thị – đã nổi lên như một công cụ mạnh mẽ trong lĩnh vực này. Một bài báo tiêu biểu là "Semi-Supervised Classification with Graph Convolutional Networks" của Kipf & Welling (ICLR 2017), lần đầu áp dụng GCN cho bài toán phân loại nút bán giám sát trên các mạng đồ thị lớn. Bài báo này giải quyết vấn đề dự đoán nhãn của các nút trong đồ thị khi chỉ một phần nhỏ nút được gắn nhãn trước, với mục tiêu tận dụng cấu trúc mạng để cải thiện độ chính xác phân loại so với mô hình truyền thống chỉ dùng dữ liệu thuộc tính. Bối cảnh ứng dụng có thể là mạng xã hội: ví dụ dự đoán nhóm sở thích hoặc vai trò của người dùng trong mạng dựa trên thông tin bạn bè của họ.

Phương pháp của Kipf & Welling đã đạt kết quả vượt trội trên các benchmark đồ thị, cho thấy sức mạnh của việc kết hợp cấu trúc mạng vào mô hình học sâu. Điều này mở ra hướng tiếp cận mới trong phân tích mạng xã hội, nơi GCN và các mô hình Graph Neural Network (GNN) liên quan đang dần thay thế các kỹ thuật truyền thống.



Hình 5. Ví dụ một mạng xã hội nhỏ (Karate Club)

2. Nguyên lý hoạt động của GCN

Graph Convolutional Network (GCN) mở rộng ý tưởng mạng neural tích chập (CNN) sang dữ liệu đồ thị không cấu trúc đều. Trước tiên, ta cần biểu diễn mạng xã

hội dưới dạng đồ thị có trọng số: Giả sử có N người dùng, ta tạo ma trận kề $A \in R^{N\times N}$ $với\ A_{ij} = 1$ nếu có kết nối (ví dụ quan hệ bạn bè) giữa người dùng i và j (không hướng), và ma trận đặc trưng $X \in R^{N\times d}$ chứa d đặc trưng cho mỗi người dùng (chẳng hạn thông tin hồ sơ). GCN sẽ thực hiện lan truyền và tổng hợp đặc trưng trên đồ thị này: mỗi lớp của GCN cập nhật vector đặc trưng của một nút bằng cách lấy trung bình có trọng số các vector đặc trưng của hàng xóm của nút đó (theo cấu trúc A), rồi áp dụng hàm phi tuyến. Cụ thể, công thức lan truyền đặc trưng kinh điển của Kipf & Welling cho một lớp GCN là:

$$H^{(l+1)} = \sigma\left(D^{-1/2}AD^{-1/2}H^{(l)}W^{(l)}\right)$$

Ở đây $H^{(l)}$ là ma trận đặc trưng các nút sau lớp thứ l (với $H^{(0)} = X$ ban đầu), $W^{(l)}$ là ma trận trọng số học được của lớp l, σ là hàm kích hoạt (ví dụ ReLU). Ma trận $\hat{A} = A + I$ là ma trận kề có thêm vòng lặp cho mỗi nút (tự kết nối), $\widehat{D}$ là ma trận độ chừng (degree) tương ứng. Công thức này nghĩa là mỗi nút tự lấy đặc trưng của mình cộng với đặc trưng trung bình đã chuẩn hóa của hàng xóm, rồi áp dụng hàm kích hoạt và trọng số để thu được đặc trưng mới. Bằng cách xếp chồng L lớp như vậy, GCN thu được thông tin từ các hàng xóm ở khoảng cách L bước: ví dụ GCN 2 lớp sẽ kết hợp thông tin từ bạn bè và "bạn của bạn" của mỗi nút.

Số lớp GCN thường được chọn rất nhỏ (chỉ 2–3 lớp) trong thực nghiệm. Lý do là hiện tượng "quá làm mượt" (over-smoothing): khi chồng quá nhiều lớp, các vector đặc trưng khuynh hướng trở nên đồng nhất giữa các nút liên thông, làm mô hình mất khả năng phân biệt và giảm hiệu quả dự đoán. Thực tế, các nghiên cứu sớm đã nhận thấy thêm quá 2 tầng tích chập thường khiến hiệu năng giảm mạnh, do đó kiến trúc GCN "chuẩn" thường khá nông (shallow). Mặc dù vậy, chỉ với 2 lớp, GCN đã đủ mạnh để thu thập thông tin từ vùng lân cận (2-hop) – thường là phạm vi quan trọng trong mạng xã hội (bạn bè và bạn của bạn).

Việc huấn luyện GCN được thực hiện bằng thuật toán lan truyền ngược (backpropagation) tương tự các mạng neural khác. Trong bài toán phân loại nút bán giám sát, ta có nhãn của một tập con các nút Y_L. Mục tiêu là tối ưu hàm mất mát cross-entropy đa lớp trên các nút có nhãn. Cụ thể, sau khi GCN tính toán ra đầu ra $Z = H^{(L)}$ (ma trận $N \times C$ với C loại nhãn, mỗi dòng Z_i là output cho nút i), ta áp dụng softmax trên mỗi dòng và tính lỗi $L = -\sum_{i\in Y_L}\sum_{f=1}^{C} Y_{if} \ln Z_{if}$. Gradient của L so với các tham số $W^{(l)}$ được tính bằng lan truyền ngược qua các lớp GCN. Kipf & Welling đã sử dụng tối ưu Gradient Descent toàn cục trên toàn bộ đồ thị cho mỗi bước (vì đồ thị thí nghiệm có kích thước vừa phải). Trong các ứng dụng lớn hơn, có thể dùng biến thể Stochastic Gradient Descent theo mini-batch, kết hợp kỹ thuật lấy mẫu để giảm chi phí tính toán. Sau quá trình huấn luyện, mô hình GCN có thể dự đoán nhãn cho các nút chưa biết nhãn dựa trên đặc trưng đã học. Như vậy, GCN học

end-to-end từ đặc trưng đầu vào và cấu trúc đồ thị đến nhiệm vụ dự đoán cuối, không cần bước tách rời nào khác. Đây là ưu điểm lớn so với cách tiếp cận cũ phải qua nhiều bước tiền xử lý và trích chọn đặc trưng.

3. Ứng dụng của GCN trong mạng xã hội

GCN và các biến thể của nó đã được áp dụng thành công trên nhiều tác vụ phân tích mạng xã hội quan trọng. Dưới đây là một số ứng dụng tiêu biểu:

- ***Phân loại người dùng***: Đây là nhiệm vụ dự đoán nhãn hoặc thuộc tính của người dùng dựa trên thông tin mạng. Ví dụ, ta có thể muốn phân loại người dùng thành các nhóm sở thích, độ tuổi, hoặc dự đoán liệu một tài khoản là spam hay không. GCN tỏ ra hữu ích vì nó khai thác hiện tượng đồng tính (homophily) thường có trong mạng xã hội: những người kết nối với nhau thường có đặc điểm hoặc hành vi tương tự. Bằng cách truyền thông tin giữa các người dùng liên kết, GCN cải thiện độ chính xác phân loại so với mô hình chỉ dựa vào dữ liệu cá nhân đơn lẻ. Chẳng hạn, nếu một số tài khoản trong nhóm bạn bè đã được nhận biết là spam, GCN có thể lan truyền "tín hiệu" đó qua các cạnh để giúp xác định các tài khoản liên quan cũng có khả năng là spam.

- ***Phát hiện cộng đồng***: Nhiều mạng xã hội phân mảnh thành các cộng đồng – nhóm người dùng kết nối mật thiết bên trong nhưng lỏng lẻo ra bên ngoài. GCN có thể được dùng để phát hiện các cộng đồng này. Một cách tiếp cận là sử dụng GCN để học embedding (biểu diễn vector) cho mỗi nút rồi sử dụng các thuật toán phân cụm (k-means, clustering) trên không gian embedding để tìm cụm cộng đồng. Do các nút trong cùng cộng đồng thường có cấu trúc kết nối tương tự, GCN sẽ gán cho họ các embedding gần nhau, hỗ trợ việc tách biệt các cụm. Thú vị là, nghiên cứu cho thấy chỉ với việc lan truyền qua 3 lớp GCN, mô hình đã tự hình thành biểu diễn phân nhóm các nút rất sát với cấu trúc cộng đồng, ngay cả khi chưa tinh chỉnh trọng số theo nhãn. Điều này chứng tỏ GCN có thể tự phát hiện cộng đồng ẩn dựa trên cấu trúc đồ thị. Trong Hình 1 ở trên, mạng Karate Club được chia thành các cộng đồng màu khác nhau; một GCN đơn giản có thể phân tách được các màu này trong không gian đặc trưng của nó. Nếu có một số cộng đồng được gắn nhãn trước (bán giám sát), ta cũng có thể huấn luyện GCN để gán nhãn cộng đồng cho các nút chưa biết – cách này thường chính xác hơn phương pháp truyền thống nhờ tận dụng đồng thời thông tin nhãn và cấu trúc mạng.

- ***Dự đoán liên kết (link prediction)***: Đây là bài toán dự đoán xem mối quan hệ nào có thể hình thành trong tương lai, hoặc gợi ý những kết nối (bạn bè, tương tác) mới giữa các người dùng chưa trực tiếp liên kết. GCN có thể áp dụng bằng nhiều cách.

Một hướng là sử dụng kiến trúc autoencoder trên đồ thị: Graph Auto-Encoder (GAE) do Kipf & Welling đề xuất (2016) gồm một phần encoder là GCN để tạo embedding cho các nút, sau đó phần decoder cố gắng tái tạo lại các cạnh từ embedding đó. Mô hình GAE huấn luyện nhằm tái lập cấu trúc mạng sẽ học được các embedding chứa thông tin về khả năng liên kết giữa các nút, từ đó có thể dự đoán được liên kết mới với độ chính xác cao. Trong mạng xã hội, cách này dùng để xây dựng hệ thống gợi ý bạn bè: dựa trên embedding học được, hệ thống đề xuất những cặp người dùng có khoảng cách embedding nhỏ (tức là có nhiều điểm chung) mà chưa kết nối, coi đó là gợi ý kết bạn. So với các chỉ số truyền thống (như chỉ số bạn chung, khoảng cách ngắn nhất…), phương pháp dựa trên GCN/GAE thường cho kết quả chính xác hơn nhờ học được cả mô hình phức tạp của đồ thị chứ không chỉ một tiêu chí đơn lẻ.

- ***Phát hiện tài khoản giả mạo***: Mạng xã hội phổ biến xuất hiện nhiều tài khoản giả (fake accounts) hay tài khoản ác ý (tung tin giả, gây rối). Những tài khoản này thường có hành vi kết nối đặc thù – ví dụ, nhiều tài khoản giả có thể kết bạn lẫn nhau tạo thành "ổ" trong mạng. GCN rất phù hợp để nhận diện loại đối tượng này vì nó có thể học được mẫu hình kết nối bất thường kết hợp với đặc trưng người dùng. Trong một nghiên cứu, mô hình GNN (dựa trên GCN) đạt độ chính xác tới ~81,1% trong việc phân biệt tài khoản giả, cao hơn hẳn so với mô hình SVM truyền thống chỉ dùng đặc trưng cô lập. Thậm chí, GCN còn cho phép tích hợp cả dữ liệu nội dung: chẳng hạn mô hình phát hiện troll trên mạng xã hội gần đây sử dụng GCN để kết hợp thông tin văn bản bài viết (word embeddings) với đồ thị tương tác giữa người dùng, cải thiện khả năng nhận biết nội dung độc hại và tài khoản xấu. Nhờ sức mạnh biểu diễn, GCN giúp tự động hóa việc phát hiện những tài khoản có liên kết đáng ngờ mà khó phát hiện bằng mắt thường, hỗ trợ đắc lực cho công tác kiểm duyệt mạng xã hội.

4. Ưu và nhược điểm của phương pháp GCN

4.1. Ưu điểm

1. Tận dụng cấu trúc một cách hiệu quả: Khác với mô hình truyền thống coi các mẫu dữ liệu là độc lập, GCN chủ động khai thác mối quan hệ kết nối giữa các đối tượng. Nhờ cơ chế tích chập trên đồ thị, GCN tự động học cách kết hợp đặc trưng lân cận mà không cần con người thiết kế thủ công các chỉ số mạng phức tạp. Điều này đặc biệt hữu ích trong mạng xã hội vốn có cấu trúc phức tạp và khó mô tả bằng vài chỉ số đơn giản.

2. Học end-to-end tối ưu cho nhiệm vụ: GCN cho phép huấn luyện trực tiếp từ dữ liệu đầu vào đến mục tiêu cuối (ví dụ nhãn người dùng) trong một mô hình thống nhất. Nó không tách rời bước học biểu diễn và bước dự đoán như các phương pháp embedding

truyền thống (vốn phải học embedding trước rồi mới dùng làm đầu vào cho mô hình khác). Nhờ đó, GCN có thể tận dụng thông tin nhãn trong quá trình học biểu diễn, dẫn tới biểu diễn tối ưu hơn cho nhiệm vụ cần giải quyết. Thực nghiệm cho thấy GCN đạt kết quả phân loại cao hơn hẳn so với việc dùng embedding DeepWalk/Node2Vec rồi mới phân loại bằng mô hình ngoài (đặc biệt khi dữ liệu có sẵn cả thuộc tính nút lẫn nhãn một phần).

3. Đơn giản và linh hoạt: Kiến trúc GCN cơ bản tương đối đơn giản, thường chỉ gồm vài lớp tích chập và hàm phi tuyến (ví dụ 2 lớp + ReLU) – số tham số không quá lớn, giảm nguy cơ overfit trên đồ thị không quá to. Mô hình này dễ triển khai với các thư viện học sâu hiện có. Hơn nữa, GCN đóng vai trò nền tảng linh hoạt: ta có thể biến tấu kiến trúc (thêm cơ chế attention, sampling…) để phù hợp bài toán cụ thể. Rất nhiều biến thể GNN hiện nay (GraphSAGE, GAT, PinSAGE...) thực chất phát triển từ ý tưởng GCN, kế thừa tính đơn giản mà hiệu quả của nó.

4. Hiệu quả vượt trội trên nhiều tác vụ: Nhờ kết hợp được cả thông tin nội tại (thuộc tính người dùng) và thông tin cấu trúc (bạn bè, kết nối), GCN thường cho kết quả tốt hơn so với mô hình chỉ dựa vào một trong hai nguồn tin. Kipf & Welling cho thấy GCN đánh bại các phương pháp cạnh tranh trước đó một cách thuyết phục trên các benchmark đồ thị (như mạng trích dẫn. Trong mạng xã hội, các ứng dụng từ phân loại, phát hiện cộng đồng đến dự đoán liên kết, phát hiện gian lận đều ghi nhận sự cải thiện khi dùng GCN so với thuật toán truyền thống. Có thể nói, GCN đang thiết lập những mốc hiệu năng mới cho các bài toán phân tích mạng.

4.2. Nhược điểm

1. Tầm lan truyền thông tin hạn chế: GCN cơ bản chỉ lan truyền thông tin trong phạm vi vài bước láng giềng. Nếu quan hệ phụ thuộc vào cấu trúc xa hơn (hơn 3-hop) thì mô hình GCN nông có thể bỏ lỡ. Việc tăng số lớp để mở rộng tầm nhìn lại vấp phải vấn đề over-smoothing – các nút quá "hòa trộn" đặc trưng với hàng xóm khiến mọi nút trở nên giống nhau, làm giảm độ phân biệt giữa các lớp (label). Do đó, GCN truyền thống khó khai thác hiệu quả thông tin ở khoảng cách rất xa trên đồ thị. Một số nghiên cứu đang tìm cách khắc phục, như dùng skip connections hoặc normalization đặc biệt để cho phép mạng nhiều lớp hơn mà không over-smooth, nhưng đó là những mở rộng phức tạp ngoài phạm vi GCN cơ bản.

2. Hạn chế với đồ thị cực lớn: Phương pháp GCN gốc yêu cầu sử dụng toàn bộ ma trận kề $\boldsymbol{N} \times \boldsymbol{N}$ trong tính toán mỗi lớp (dù có thể tối ưu bằng sparse matrix). Với mạng xã hội hàng triệu người dùng, ma trận này có thể quá lớn để xử lý trên một máy đơn. Hơn nữa, thuật toán huấn luyện gốc duyệt qua mọi nút ở mỗi epoch (transductive training), chi phí tính toán tăng theo kích thước đồ thị $\boldsymbol{O}(|\boldsymbol{E}|)$. Điều này khiến GCN khó áp dụng trực tiếp

cho mạng quy mô rất lớn hoặc cập nhật liên tục. Các nghiên cứu sau đã đề xuất hướng tiếp cận lấy mẫu: tiêu biểu là GraphSAGE – thay vì dùng toàn bộ hàng xóm, mỗi bước chỉ lấy mẫu một số hàng xóm ngẫu nhiên để cập nhật nút. Cách này cho phép huấn luyện theo minibatch và áp dụng mô hình cho nút mới (inductive) thay vì buộc nút đó có trong quá trình train. Tuy nhiên, việc triển khai các kỹ thuật này phức tạp hơn và đòi hỏi tài nguyên tính toán phân tán (GPUs, TPUs) khi scale out. Tóm lại, tính khả chuyển và mở rộng của GCN ban đầu còn hạn chế, cần các biện pháp bổ sung để xử lý mạng xã hội khổng lồ.

3. Phụ thuộc vào tính chất dữ liệu: Hiệu quả của GCN cao khi giả định đồng nhất (homophily) được thỏa mãn – tức các nút liên kết có xu hướng cùng nhãn hoặc tương đồng về đặc trưng. Trong nhiều mạng xã hội điều này đúng (bạn bè thường chung sở thích), nhưng không phải luôn luôn. Nếu đồ thị có quan hệ dị loại (heterophily) – ví dụ người dùng kết nối với người rất khác mình – GCN có thể truyền nhầm thông tin và giảm độ chính xác. Khi đó, mô hình đơn giản chỉ dựa vào thuộc tính cá nhân (như MLP trên đặc trưng) thậm chí có thể vượt GCN. Ngoài ra, GCN cũng yêu cầu nút có đặc trưng đầu vào tốt. Trong trường hợp ta không có đặc trưng gì (chỉ có cấu trúc đồ thị), phải dùng kỹ thuật như gán đặc trưng một-hot hoặc độ đo tay (degree, centrality) làm đặc trưng đầu vào. Lúc này, hiệu quả GCN có thể kém đi, vì thực chất mô hình gần như chỉ thực hiện lan truyền nhãn (gần giống thuật toán label propagation cổ điển).

4. Độ phức tạp tính toán: So với mô hình truyền thống (thường là thuật toán đồ thị hoặc hồi quy/phân lớp tuyến tính), GCN là mô hình học sâu nên cần nhiều thời gian huấn luyện hơn và đòi hỏi tinh chỉnh siêu tham số (số lớp, số đơn vị ẩn, learning rate, v.v.). Việc giải thích kết quả mô hình cũng không trực quan bằng các chỉ số mạng đơn giản (ví dụ centrality). Trong một số ứng dụng nhạy cảm (như phân tích mạng tội phạm), tính minh bạch có thể quan trọng, do đó sự "hộp đen" của GCN là một điểm trừ. Tuy vậy, các nghiên cứu XAI (giải thích mô hình AI) đang dần được áp dụng cho GNN, giúp người dùng hiểu được phần nào những yếu tố trong cấu trúc mạng đã dẫn mô hình đến quyết định.

5. So sánh Graph Convolutional Network (GCN) và Graph Attention Network (GAT)

Tiêu chí	**Graph Convolutional Network (GCN)**	**Graph Attention Network (GAT)**
Phương pháp tổng hợp	Tổng hợp láng giềng bằng trung bình có trọng số cố định theo cấu trúc đồ thị. Mỗi láng giềng j của nút i đóng góp $\frac{1}{\sqrt{d_i d_j}}$ vào tổng (có self-loop). Do đó	Tổng hợp láng giềng bằng cơ chế tự chú ý (self-attention). Mỗi cặp nút kề i, j tính một hệ số chú ý e_{ij} qua mạng con (dùng vector trọng số $\vec{a}$ và hàm phi tuyến) rồi chuẩn hóa thành α_{ij} bằng softmax. Bởi

	GCN coi các láng giềng quan trọng như nhau sau khi đã chuẩn hóa theo bậc.	vậy, mức ảnh hưởng của mỗi láng giềng j được học lên/xuống một cách linh hoạt tùy ngữ cảnh nút i.
Tham số học	Mỗi lớp có một ma trận trọng số $W^{(l)}$. Không có tham số dành riêng cho từng cạnh – phép tổng hợp dựa hoàn toàn vào cấu trúc (bậc nút). Do đó số tham số khá ít, mô hình gọn nhẹ.	Mỗi head attention có thêm vector trọng số $\vec{a}$ để tính mức độ quan trọng của cạnh. Có tham số chú ý học được nên tổng số tham số nhiều hơn GCN một chút (tăng tuyến tính theo số head).
Ưu điểm	Đơn giản, hiệu quả: Ít tham số nên huấn luyện nhanh, ít overfit hơn trên dữ liệu ít. - Tận dụng cấu trúc toàn cục: Trọng số dựa trên cấu trúc giúp GCN ổn định, giống như một phép làm trơn tín hiệu trên đồ thị (smoothing), thường hiệu quả khi các nút láng giềng có tính chất tương đồng (mạng đồng nhất).	Linh hoạt, biểu đạt mạnh: Cho phép trọng số khác nhau cho các kết nối nên mô hình phân biệt được tầm quan trọng của từng láng giềng, tăng khả năng biểu đạt quan hệ phức tạp. - Diễn giải được: Các hệ số chú ý α_{ij} giúp ta hiểu nút nào quan trọng đối với quyết định của mô hình, tăng tính giải thích được của kết quả.
Hạn chế	Thiếu tinh vi: Không phân biệt được giữa các hàng xóm – có thể bỏ lỡ những nút quan trọng nếu chỉ dùng trung bình. - Giả định đồng nhất: Hiệu năng giảm nếu đồ thị có dị hình (heterophily), khi láng giềng không tương đồng nhau, vì GCN có xu hướng làm các nút kết nối gần nhau trở nên giống nhau về biểu diễn.	Tốn tài nguyên hơn: Tính attention cho từng cạnh tăng chi phí tính toán (dù vẫn cùng bậc độ phức tạp O)

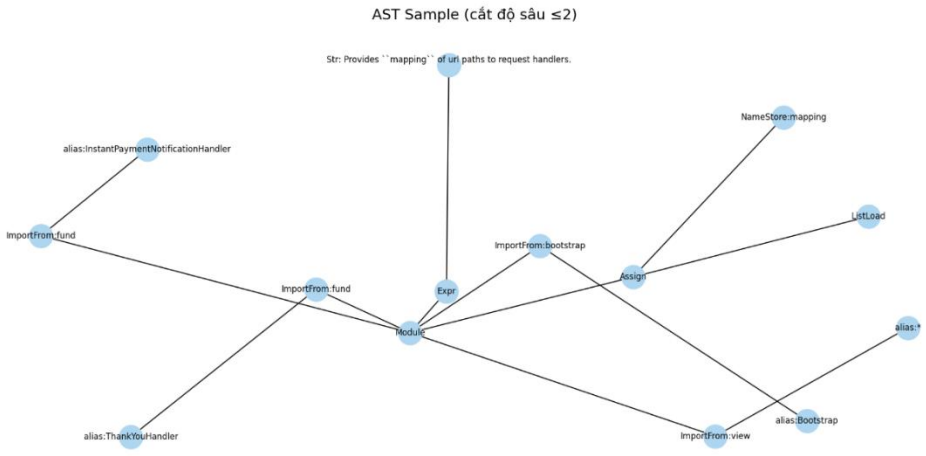
Bảng 1. Bảng so sáng GCN và GAT

CHƯƠNG IV: HIỆN THỰC VÀ KẾT QUẢ

1. Giới thiệu bộ dữ liệu

Mỗi chương trình Python trong bộ dữ liệu Python150k được biểu diễn dưới dạng cây cú pháp trừu tượng (AST) lưu trữ dưới định dạng JSON. Cụ thể, mỗi AST là một mảng các đối tượng (node), trong đó mỗi node có trường type (kiểu nút, bắt buộc), và tùy chọn value (giá trị của nút, nếu có) và children (danh sách chỉ số các nút con) như mô tả trên trang chính thức. Ví dụ ở Hình 1 cho thấy cây AST của một chương trình đơn giản (chứa các node Module, Assign, Print, NameLoad, Num, v.v.), minh họa cách các nút con được nối tới nút cha.

Phân tích sơ bộ trên 10.000 AST (từ tập huấn luyện) cho thấy trung bình mỗi AST chứa khoảng 606 nút, trong đó ~320 nút là nút lá (không có con), và độ sâu trung bình khoảng 12.1 cấp. Do bộ dữ liệu giới hạn tối đa 30.000 nút cho mỗi AST nên kích thước AST rất đa dạng, từ rất nhỏ (có thể chỉ vài nút) đến rất lớn. Trung bình có khoảng 30.7 loại node khác nhau xuất hiện trong mỗi AST. Tổng hợp trên toàn bộ tập dữ liệu (100k AST huấn luyện), có khoảng 181 loại node khác nhau. Các loại node xuất hiện nhiều nhất bao gồm NameLoad (tải biến), attr và AttributeLoad (truy cập thuộc tính), Str (hằng chuỗi), Num (hằng số), Call (gọi hàm), Assign (phép gán), NameStore (lưu biến), Expr (biểu thức độc lập), body (khối lệnh thân hàm hoặc khối),… Những node này chiếm phần lớn các nút trong AST. Tổng quan, bộ dữ liệu có 150.000 chương trình Python (100.000 dùng huấn luyện và 50.000 kiểm thử.



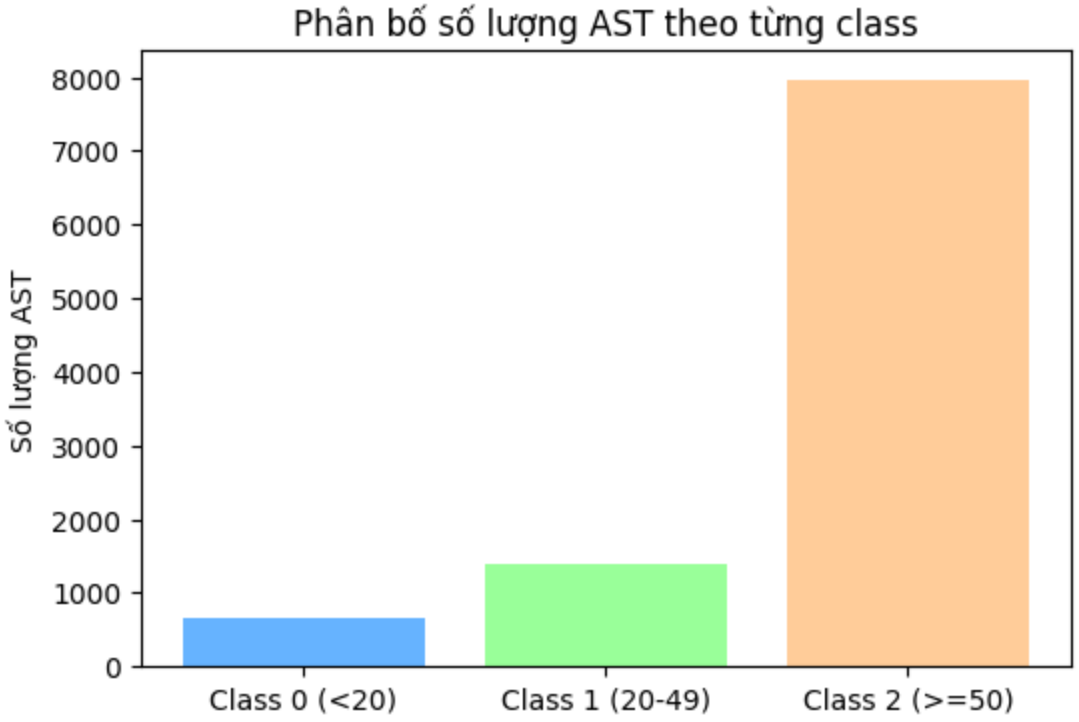
Hình 6. Ví dụ cấu trúc đồ thị AST (độ sâu giới hạn ≤2) trích từ bộ dữ liệu Python150

Phân bố kích thước AST và nhãn

Các AST được chia thành ba lớp dựa theo độ dài (số lượng node): nhỏ (<20 node), vừa (20–49 node), và lớn (≥50 node). Trên mẫu 10.000 AST đã phân tích, thống kê số lượng và tỷ lệ phần trăm như sau:

- AST nhỏ (<20 node): 656 AST (~6.6%).
- AST trung bình (20–49 node): 1.381 AST (~13.8%).
- AST lớn (≥50 node): 7.963 AST (~79.6%).

Như vậy, phần lớn AST trong dữ liệu thuộc lớp lớn (cây dài), chỉ một tỉ lệ nhỏ (<7%) có độ ngắn (<20). Các biểu đồ trực quan (ví dụ histogram chiều dài AST) sẽ cho thấy một đỉnh lớn ở bên phải biểu đồ do số lượng AST lớn (≥50 node) áp đảo.

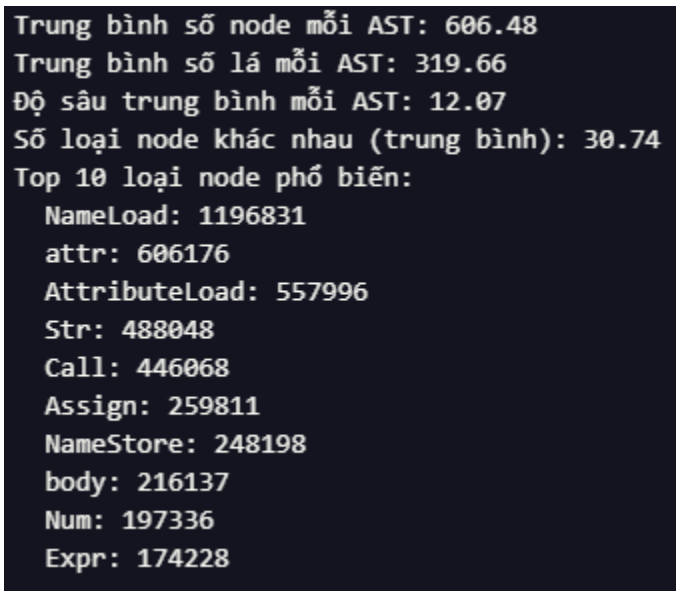


Hình 7. Biểu đồ phân bố số lượng AST theo class

Trước khi tiến hành xây dựng mô hình, chúng tôi thực hiện phân tích thống kê nhanh trên 10.000 AST đầu tiên trong tập huấn luyện để nắm bắt đặc trưng chung của dữ liệu:

- Trung bình số node mỗi AST: 606.48 node
- Trung bình số lá (leaf) mỗi AST: 319.66 node
- Độ sâu trung bình mỗi AST: 12.07 cấp
- Số loại node khác nhau trung bình mỗi AST: 30.74 loại

Những con số trên cho thấy mỗi cây AST có quy mô tương đối lớn (khoảng 600 node), gần một nửa là node lá (leaf), và cấu trúc đa tầng với chiều sâu khoảng 12 cấp. Mỗi AST sử dụng trung bình hơn 30 loại nút khác nhau, phản ánh sự phong phú của cấu trúc cú pháp trong mã Python.



Hình 8. Thống kê nhanh dựa vào 10000 AST đầu

Thứ tự	Loại node	Số lần xuất hiện
1	NameLoad	1 196 831
2	attr	60 176
3	AttributeLoad	557 996
4	Str	488 048
5	Call	446 068
6	Assign	259 811
7	NameStore	248 198
8	body	216 137
9	Num	197 336
10	Expr	174 228

Bảng 2. Bảng thống kê top 10 loại node thường gặp

- NameLoad (tải biến) dẫn đầu với hơn 1.19 triệu node, cho thấy mã nguồn Python thường xuyên tham chiếu biến.
- Các node thuộc tính như attr và AttributeLoad xuất hiện rất nhiều, phản ánh cú pháp truy cập thành viên (object.attr).
- Node Str và Num (hằng chuỗi và hằng số số) xuất hiện hàng trăm nghìn lần, tương ứng với việc sử dụng literal trong code.
- Các node điều khiển cấu trúc AST như Call, Assign, NameStore, body, Expr cũng rất phổ biến, chứng tỏ mã Python bao gồm nhiều phép gán, lời gọi hàm và biểu thức độc lập.

2. Tiền xử lý dữ liệu

2.1. Load dữ liệu

Bước 1: Thiết lập môi trường và thư viện

```
import os, sys, json, random
import numpy as np
from tqdm import tqdm

import torch
from torch_geometric.data import Data
```

Hình 9. Các thư viện cần thiết

Thư viện chính: json để parse, tqdm để hiển thị tiến độ, torch_geometric để xây dựng đối tượng graph, numpy cho thống kê, random để sampling.

Bước 2: Đọc và lọc AST thô

```
with open(path, 'r', encoding='utf-8') as f:
    for line in f:
        sample = json.loads(line)
```

Hình 10. Mở file và đọc từng dòng

Xác định cấu trúc AST

- Nếu sample là dict, lấy sample['ast'] (một list node).
- Nếu sample là list, chính list đó là AST.

```python
if isinstance(sample, dict):
    ast_data = sample.get('ast', [])
elif isinstance(sample, list):
    ast_data = sample
else:
    continue  # bỏ qua nếu không phải dict/list
```

Hình 11. Xác định cấu trúc AST

Loại bỏ AST không hợp lệ:

```python
if len(ast_data) < 5:
    continue
```

Hình 12. Loại bỏ AST quá nhỏ

Mục đích: tránh các AST quá nhỏ, không đủ cấu trúc, hoặc lỗi parse gây ra.

2.2 Chuyển AST thành đồ thị PyG

Bước 1: Tạo ma trận cạnh (edge_index)

- Duyệt qua từng node parent_idx và danh sách children:

```python
edge_list = []
for parent_idx, node in enumerate(ast_data):
    for child_idx in node.get('children', []):
        if 0 <= child_idx < len(ast_data):
            # Thêm cạnh hai chiều để mô hình xem như đồ thị vô hướng
            edge_list.append([parent_idx, child_idx])
            edge_list.append([child_idx, parent_idx])
```

Hình 13. Tạo ma trận cạnh

```python
if edge_list:
    edge_index = torch.tensor(edge_list, dtype=torch.long).t().contiguous()
else:
    edge_index = torch.zeros((2,0), dtype=torch.long)
```

Hình 14. Chuyển sang tensor PyTorch

*edge_index có shape [2, E], nơi E là số cạnh (đã nhân đôi do hai chiều)

Bước 2. Mã hóa loại node thành feature

Thu thập các loại node trong một AST

```
types = [node['type'] for node in ast_data]
```

Hình 15. Thu thập node

Tạo mapping type → chỉ số (riêng cho mỗi AST, hoặc toàn cục trên tập train)

```
type2idx = {t: i for i, t in enumerate(sorted(set(types)))}
```

Hình 16. Tạo mapping

Tạo tensor feature x

```
x = torch.tensor([type2idx[t] for t in types], dtype=torch.float).unsqueeze(1)
```

Hình 17. Tạo tensor

- Kết quả x.shape = [N, 1] với N = số node
- Có thể mở rộng feature bằng cách thêm degree hoặc depth nếu cần.

Bước 3. Đóng gói thành đối tượng Data

```
from torch_geometric.data import Data

g = Data(x=x, edge_index=edge_index)
```

Hình 18. Đóng gói

- g.x: ma trận feature nút.
- g.edge_index: ma trận index cạnh.

2.3. Gán nhãn đồ thị theo kích thước AST

```
size = len(ast_data)
if   size < 20: label = 0
elif size < 50: label = 1
else:           label = 2
g.y = torch.tensor([label], dtype=torch.long)
```

Hình 19. Gắn nhãn

2.4. Cân bằng tập huấn luyện

Over/Under-sampling bằng code thủ công

```python
def balance_dataset(graphs, target_counts=[700,700,700]):
    by_label = {0:[],1:[],2:[]}
    for g in graphs:
        by_label[g.y.item()].append(g)
    balanced = []
    for lbl, target in enumerate(target_counts):
        grp = by_label[lbl]
        if len(grp) < target:
            # over-sample: nhân đôi, chép ngẫu nhiên
            times = target // len(grp)
            balanced += grp * times
            balanced += random.sample(grp, target % len(grp))
        else:
            # under-sample: chọn ngẫu nhiên
            balanced += random.sample(grp, target)
    random.shuffle(balanced)
    return balanced

balanced_train_graphs = balance_dataset(train_graphs)
print_stats(balanced_train_graphs, "Balanced train graphs")
```

Hình 20. Over/Under-sampling

```
Balanced train graphs:
  Số graphs: 2100
  Trung bình nodes: 256.38, edges: 255.38
  Phân bố labels: [700 700 700]
```

Hình 21. Kết quả balance

2.5. Thống kê cơ bản

```
def print_stats(graphs, name):
    Ns = [g.num_nodes for g in graphs]
    Es = [g.edge_index.size(1)//2 for g in graphs]
    labels = [g.y.item() for g in graphs]
    print(f"{name}:")
    print(f"  Số graphs: {len(graphs)}")
    print(f"  Trung bình nodes: {np.mean(Ns):.2f}, edges: {np.mean(Es):.2f}")
    print(f"  Phân bố labels: {np.bincount(labels)}\n")

print_stats(train_graphs, "Train graphs")
print_stats(eval_graphs,  "Eval graphs")
```

Hình 22. Hàm thống kê ban đầu

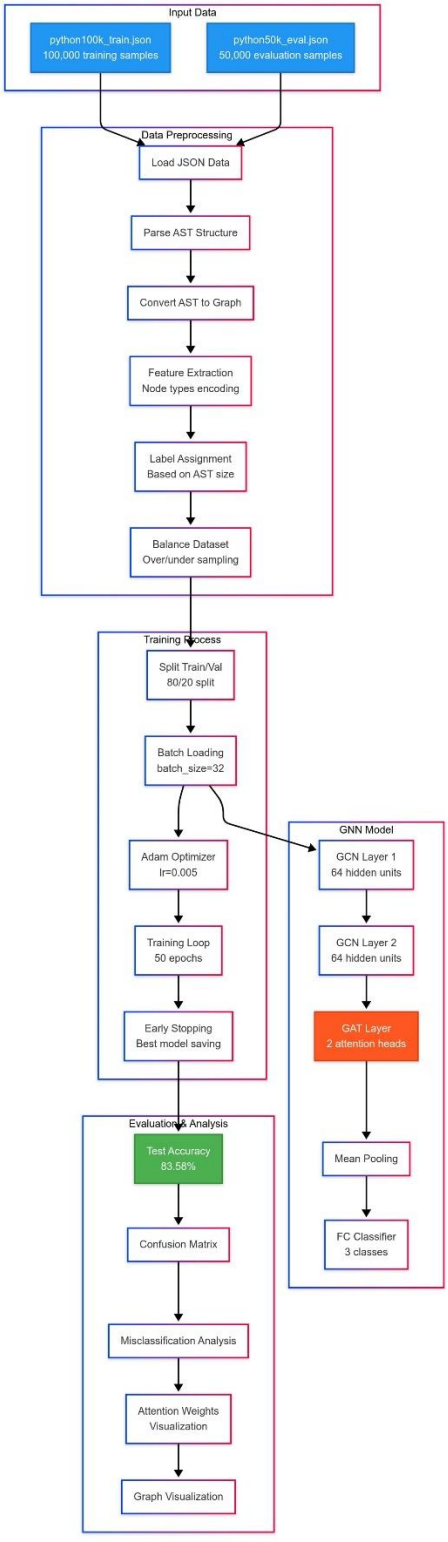
```
Loading /kaggle/input/py150k-1/python100k_train.json: 100%|██████████| 20000/20000 [00:57<00:00, 350.09it/s]
Loading /kaggle/input/py150k-1/python50k_eval.json: 100%|██████████| 4000/4000 [00:10<00:00, 394.99it/s]
Train graphs:
  Số graphs: 19615
  Trung bình nodes: 625.68, edges: 624.68
  Phân bố labels: [  872  2789 15954]

Eval graphs:
  Số graphs: 3915
  Trung bình nodes: 534.08, edges: 533.08
  Phân bố labels: [ 187  914 2814]
```

Hình 23. Kết quả thống kê trên tập kiểm thử

3. Sơ đồ thiết kế thực nghiệm

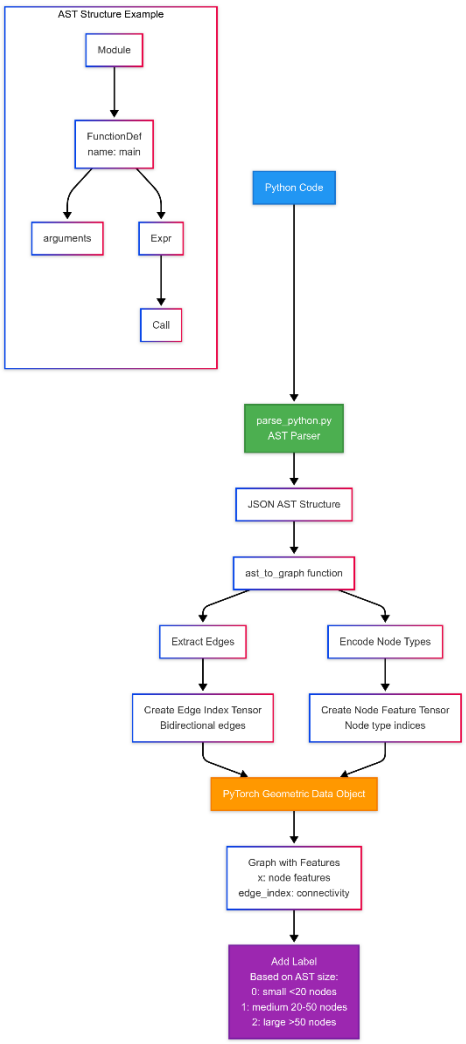
Sơ đồ tổng quát hệ thống thực nghiệm. Đầu vào là hai file dữ liệu JSON (train 100k, eval 50k). Sau tiền xử lý bao gồm load dữ liệu, parse cấu trúc AST, chuyển đổi AST thành đồ thị, trích xuất feature loại node, gán nhãn theo kích thước AST và cân bằng bộ dữ liệu. Phần training chia tập train thành train/val, load theo batch (size=32), sử dụng bộ tối ưu Adam (lr=0.005) và huấn luyện trên 50 epoch với early stopping (lưu mô hình tốt nhất). Mô hình GNN gồm hai lớp GCN (64 hidden units), một lớp GAT nhiều đầu (2 head), kết hợp mean pooling và FC classifier. Sau training, thực hiện đánh giá trên tập test, tính accuracy, hiển thị confusion matrix, và phân tích mẫu sai với attention weights và trực quan đồ thị.



Hình 24. Sơ đồ tổng quát hệ thống thực nghiệm

Pipeline chuyển đổi AST thành đồ thị. Phía trên ví dụ một AST đơn giản với Node gốc Module chứa FunctionDef, cây con với các nhánh arguments và Expr chứa Call. Mã

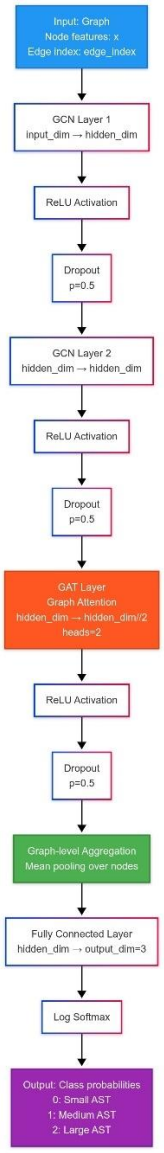
nguồn Python đầu vào được parse (bằng parse_python.py) thành cấu trúc AST JSON. Hàm ast_to_graph sau đó tiến hành: trích xuất các cạnh cha-con (bidirectional), xây ma trận edge_index; mã hóa kiểu node thành feature vector x (mỗi node lấy chỉ số kiểu thành một giá trị float); rồi tạo đối tượng PyTorch Geometric Data. Kết quả là một đồ thị Graph(x, edge_index) với feature của mỗi node và connectivity của đồ thị. Sau đó gán nhãn nhờ số nút: lớp 0 (nhỏ <20), 1 (20–49), 2 (>=50)



Hình 25. Pipeline chuyển đổi AST thành đồ thị

Kiến trúc mạng GNN kết hợp GCN + GAT. Đầu vào là đồ thị với feature x (ma trận số nút × 1) và edge_index. Đồ thị đi qua hai lớp GCN liên tiếp (tăng chiều lên hidden_dim, ở đây 128), mỗi lớp theo sau bởi ReLU và dropout. Sau đó là một lớp GAT nhiều đầu (graph attention) với 2 attention heads, từ chiều hidden_dim giảm xuống hidden_dim/2. Tiếp tục ReLU + dropout, rồi thực hiện mean pooling để thu được vector đặc trưng đồ thị.

Cuối cùng hai lớp fully-connected (128→64→3) và hàm LogSoftmax đưa ra xác suất thuộc ba lớp AST nhỏ/vừa/lớn.



Hình 26. Kiến trúc mạng GNN kết hợp GCN + GAT

4. Thiết kế mô hình

4.1. Kiến trúc mô hình

```python
class ImprovedGNNModel(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim=128, output_dim=3, dropout=0.3):
        super(ImprovedGNNModel, self).__init__()

        self.conv1 = GCNConv(input_dim, hidden_dim)

        self.conv2 = GCNConv(hidden_dim, hidden_dim)

        self.conv3 = GCNConv(hidden_dim, hidden_dim)

        self.gat = GATv2Conv(hidden_dim, hidden_dim // 4, heads=4)

        self.norm1 = torch.nn.LayerNorm(hidden_dim)
        self.norm2 = torch.nn.LayerNorm(hidden_dim)
        self.norm3 = torch.nn.LayerNorm(hidden_dim)

        self.fc1 = nn.Linear(hidden_dim, hidden_dim // 2)
        self.fc2 = nn.Linear(hidden_dim // 2, output_dim)

        self.dropout = nn.Dropout(dropout)
```

Hình 27. Sơ đồ kiến trúc

Các layer GCN và GAT được nối tiếp, cùng cơ chế normalization, dropout và phân loại cuối và forward sẽ thực hiện thứ tự:

1. Layer 1 (GCN)
    - Input: node-features $X \in R^{N \times input\_dim}$
    - Thực thi conv1: $H^{(1)} = \mathrm{ReLU}\big(\mathrm{GCNConv}(X)\big)$
    - Chuẩn hoá: norm1$\big(H^{(1)}\big)$
    - Dropout: dropout(...)
2. Layer 2 (GCN)
    - Tương tự với conv2 + norm2 + dropout
3. Layer 3 (GCN)
    - Tương tự với conv3 + norm3 + dropout

4. Layer 4 (GATv2)
    - gat: dùng cơ chế attention giữa các node, với 4 heads, cho phép mỗi node cập nhật thông tin từ k-hop neighbours với trọng số khác nhau.
    - Kết quả: $H^{(4)} = \mathrm{ReLU}\left(\mathrm{GATv2Conv}\left(H^{(3)}\right)\right)$
    - Dropout
5. Graph-level Readout
    - Sử dụng global mean pooling để gom $H(4)H^{(4)}H(4)$ từ kích thước $R^{N\times hidden\_dim}$ về R^{hidden_dim} cho toàn graph.
6. Fully-connected & Softmax
    - fc1: giảm chiều từ hidden_dim xuống hidden_dim//2, ReLU + dropout.
    - fc2: từ hidden_dim//2 ra output_dim (ở đây là 3 lớp).
    - Kết cuối: log_softmax để thu xác suất dự đoán cho mỗi class

4.2. Huấn luyện mô hình

4.2.1. Cấu hình huấn luyện mô hình

• Optimizer: Adam với learning rate = 0.005

• Weight decay: $5 \times 10{-}4$

• Batch size: 32

• Số epochs: 50

• Early stopping với patience = 10

4.2.2 Hàm huấn luyện

```python
def train(model, train_loader, optimizer, device):
    model.train()
    total_loss = 0

    for data in tqdm(train_loader, desc="Training"):
        data = data.to(device)
        optimizer.zero_grad()

        batch_losses = []
        for i in range(data.num_graphs):
            mask = data.batch == i
            sub_data = Data(
                x=data.x[mask],
                edge_index=data.edge_index[:, (data.edge_index[0] >= mask.nonzero()[0][0]) &
                                              (data.edge_index[0] <= mask.nonzero()[-1][0]) &
                                              (data.edge_index[1] >= mask.nonzero()[0][0]) &
                                              (data.edge_index[1] <= mask.nonzero()[-1][0])]
            )

            offset = mask.nonzero()[0][0]
            sub_data.edge_index[0] -= offset
            sub_data.edge_index[1] -= offset

            sub_data.y = data.y[i:i+1]

            output = model(sub_data)

            weights = torch.tensor([1.0, 2.0, 1.0], device=device)
            batch_losses.append(loss)

        batch_loss = torch.stack(batch_losses).mean()
        batch_loss.backward()
        optimizer.step()

        total_loss += batch_loss.item()

    return total_loss / len(train_loader)
```

Hình 27. Hàm huấn luyện

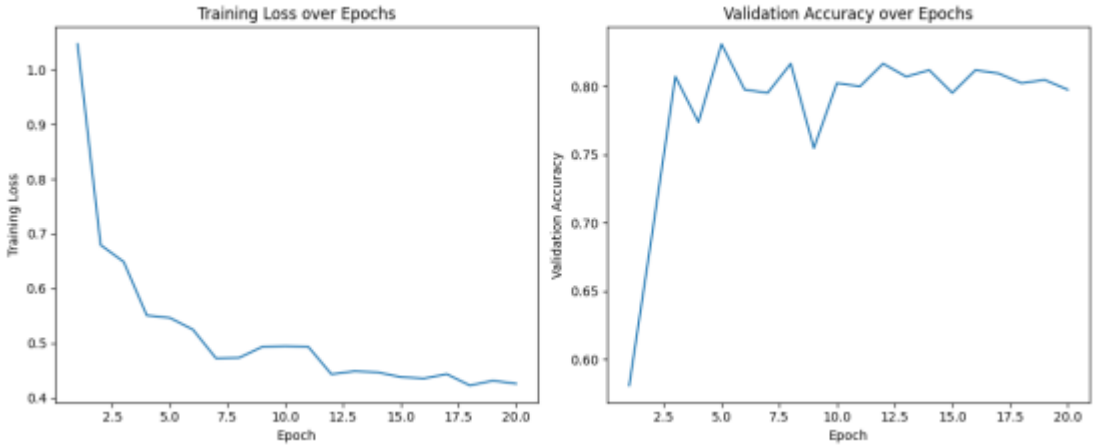
- model.train(): đưa mô hình vào chế độ huấn luyện (bật dropout, batch-norm update).
- Lặp qua train_loader, mỗi data là một batch nhiều graph (PyG đặt các graph vào cùng 1 object với trường data.batch).
- data = data.to(device): chuyển toàn bộ batch lên GPU.
- optimizer.zero_grad(): reset gradient.
- Tạo danh sách batch_losses = [] để lưu loss từng graph.
- Tách từng graph trong batch (vòng for i in range(data.num_graphs)):
    - mask = data.batch == i tạo boolean mask chọn node thuộc graph thứ i.

- Dùng mask để lọc x[mask] và cắt edge_index sao cho chỉ giữ các cạnh giữa các node đó.
- Tính offset để chuyển chỉ số node về dạng 0…n-1 (vì ban đầu node index vẫn là chỉ số từ toàn batch).
- Lấy label sub_data.y = data.y[i:i+1].
- Chạy output = model(sub_data) để thu logits.
- Chuẩn bị weights [1.,2.,1.] (nhằm cân bằng class), nhưng chưa thấy bạn tính loss = criterion(output, sub_data.y, weight=weights). Thay vào đó code chỉ batch_losses.append(loss) mà biến loss không được định nghĩa trước đó.

- Tính batch_loss làm trung bình các batch_losses.
- batch_loss.backward() và optimizer.step() cập nhật trọng số.
- Cộng dồn total_loss.

5. Kết quả

5.1. Quá trình huấn luyện



Hình 28. Biểu đồ biến đổi qua từng epochs

Biểu đồ trên thể hiện Training Loss (trái) và Validation Accuracy (phải) theo 20 epoch đầu tiên trong quá trình huấn luyện mô hình GCN+GAT.

1. Đường cong Loss (trái)
   - Ban đầu, loss ~1.1, sau 3 epoch giảm nhanh xuống ~0.6, cho thấy mô hình tiếp thu mạnh mẽ các pattern cơ bản trong dữ liệu.

- Từ epoch 4–10, loss tiếp tục giảm chậm dần, từ ~0.6 xuống ~0.5, thể hiện việc học đến mức ổn định.
- Sau epoch 10, loss dao động nhẹ quanh ~0.45–0.5, chứng tỏ mô hình đã gần hội tụ và không còn cải thiện đáng kể.

2. Đường cong Accuracy (phải)
    - Validation Accuracy khởi điểm thấp (~0.58) do weights khởi tạo ngẫu nhiên.
    - Đến epoch 3, accuracy nhảy vọt lên ~0.79, tương ứng với việc loss đã giảm đáng kể.
    - Từ epoch 4–10, accuracy dao động trong khoảng 0.80–0.83, đạt đỉnh ~0.83 tại một số epoch.
    - Sau epoch 10, accuracy thường dao động nhẹ quanh ~0.80–0.82, không tăng rõ rệt, báo hiệu plateau.

Nhận xét:

Hội tụ nhanh: loss và accuracy thay đổi mạnh trong vài epoch đầu, cho thấy tốc độ học khả quan với lr=0.001 và kiến trúc hiện tại.

- Plateau sớm: sau epoch 10, hầu như không có cải thiện, gợi ý nên
    - Giảm learning rate (ví dụ dùng scheduler giảm lr khi plateau),
    - Tăng cường regularization (dropout, weight decay) hoặc
    - Mở rộng kiến trúc (thêm layer, residual) để tiếp tục cải thiện.
- Không có overfitting đáng kể: vì validation accuracy không giảm trong giai đoạn sau, mặc dù loss training vẫn giảm nhẹ. Điều này cho thấy mô hình tổng quát hoá tương đối tốt.

Kết hợp với các chỉ số trên tập test (Accuracy 83.58%, F1 84.66%), đồ thị learning curves khẳng định mô hình đã đạt cân bằng giữa độ khớp và khả năng tổng quát.

5.2. Kết quả trên từng tập test

5.2.1. Accuracy

$$\text{Accuracy} = \frac{\sum_{c=0}^{C-1} TP_c}{\sum_{c=0}^{C-1}(TP_c + FP_c + FN_c)} = \frac{\text{Tổng số dự đoán đúng}}{\text{Tổng số mẫu}}$$

Accuracy 83.58 % nghĩa là trong 100 mẫu, mô hình dự đoán đúng khoảng 84 mẫu.

5.2.2. Precision

$$\text{Precision} = \sum_{c=0}^{C-1} w_c \cdot \frac{TP_c}{TP_c + FP_c}, \quad w_c = \frac{N_c}{N}$$

Precision 88.19 % cho thấy trung bình khi mô hình dự đoán một AST thuộc một lớp nào đó, thì gần 88 % là chính xác.

5.2.3. Recall

$$\text{Recall} = \sum_{c=0}^{C-1} w_c \cdot \frac{TP_c}{TP_c + FN_c}$$

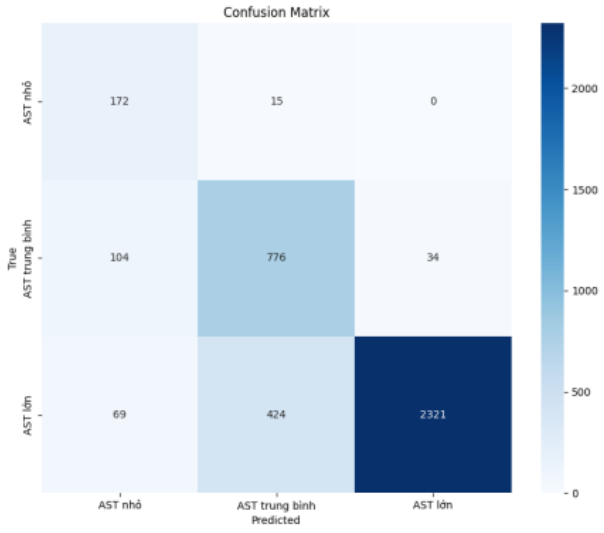
Recall 83.58 % nghĩa rằng trung bình mô hình "bắt" được 83.6 % trong số thực sự thuộc mỗi lớp.

5.2.4. F1-score

$$F1 = \sum_{c=0}^{C-1} w_c \cdot \frac{2 \cdot TP_c}{2TP_c + FP_c + FN_c} = \sum_{c=0}^{C-1} w_c \cdot \frac{2 \cdot \text{Precision}_c \cdot \text{Recall}_c}{\text{Precision}_c + \text{Recall}_c}$$

F1-score 84.66 % là hàm điều hòa của precision và recall, cân bằng hai mặt nhằm đánh giá tổng thể.

5.3. Ma trận nhầm lẫn



Hình 29. Ma trận nhầm lẫn

True \ Pred	0: Nhỏ	1: Trung bình	2: Lớn
0: Nhỏ	172	15	0
1: Trung bình	104	776	34
2: Lớn	69	424	2321

- Lớp 0 (Nhỏ):

$$TP_0 = 172, \quad FP_0 = 104 + 69 = 173, \quad FN_0 = 15 + 0 = 15$$

$$\text{Precision}_0 = \frac{172}{172 + 173} \approx 49.5\%$$

$$\text{Recall}_0 = \frac{172}{172 + 15} \approx 92.0\%$$

$$F1_0 \approx 2 \times \frac{0.495 \times 0.920}{0.495 + 0.920} \approx 64.9\%$$

- Lớp 1 (Trung bình):

$$TP_1 = 776, \quad FP_1 = 15 + 424 = 439, \quad FN_1 = 104 + 34 = 138$$

$$\text{Precision}_1 = \frac{776}{776 + 439} \approx 63.8\%$$

$$\text{Recall}_1 = \frac{776}{776 + 138} \approx 84.9\%$$

$$F1_1 \approx 2 \times \frac{0.638 \times 0.849}{0.638 + 0.849} \approx 73.4\%$$

- Lớp 2 (Lớn):

$$TP_2 = 2321, \quad FP_2 = 0 + 34 = 34, \quad FN_2 = 69 + 424 = 493$$

$$\text{Precision}_2 = \frac{2321}{2321 + 34} \approx 98.6\%$$

$$\text{Recall}_2 = \frac{2321}{2321 + 493} \approx 82.4\%$$

$$F1_2 \approx 2 \times \frac{0.986 \times 0.824}{0.986 + 0.824} \approx 90.0\%$$

CHƯƠNG V: KẾT LUẬN

1. Chủ đề và mục tiêu

Trong báo cáo này, chúng tôi đã tập trung nghiên cứu và triển khai giải pháp phân loại cấu trúc cây cú pháp trừu tượng (AST) của mã nguồn Python bằng cách ứng dụng các mô hình Graph Convolutional Network (GCN) và Graph Attention Network (GAT). Mục tiêu chính là khai thác đầy đủ thông tin cấu trúc đồ thị của AST để phân biệt ba nhóm kích thước AST—nhỏ, trung bình và lớn—với độ chính xác cao và khả năng tổng quát hóa tốt.

2. Tóm tắt các công việc đã thực hiện

1. Tiền xử lý dữ liệu
    - Đọc và parse file JSON chứa 100 000 AST huấn luyện và 50 000 AST đánh giá.
    - Lọc bỏ mẫu không hợp lệ (AST < 5 node), prune độ sâu nếu cần.
    - Mã hóa loại node thành chỉ số, xây dựng ma trận cạnh bidirectional.
    - Gán nhãn ba lớp dựa trên số node và cân bằng tập huấn luyện qua over-/under-sampling.
2. Xây dựng mô hình
    - Thiết kế kiến trúc gồm ba lớp GCNConv, một lớp GATv2Conv (4 heads), kèm LayerNorm và Dropout.
    - Áp dụng global mean pooling để chuyển từ node-level sang graph-level, sau đó qua hai tầng fully connected để phân loại.
3. Huấn luyện và kiểm thử
    - Sử dụng optimizer Adam với weight decay và scheduler giảm learning rate; áp dụng early stopping khi validation accuracy không cải thiện.
    - Đánh giá bằng các chỉ số: Accuracy, Precision, Recall, F1-score (weighted).
    - Trực quan hóa learning curves, confusion matrix và phân bố độ dài AST.
4. Phân tích kết quả

- Mô hình đạt Accuracy ≈83.6%, F1-score ≈84.7%, với lớp AST lớn có precision đến ~98.6% nhưng recall ~82.4%.
- Nhận diện rõ ưu nhược điểm trên từng lớp nhãn thông qua ma trận nhầm lẫn và báo cáo phân loại.

3. Ưu điểm của giải pháp

- Khai thác cấu trúc đồ thị: GCN cho phép lan truyền đồng đều, GAT cung cấp attention trọng số, kết hợp hiệu quả hai cơ chế này giúp trích xuất đặc trưng cấu trúc AST chính xác.
- Ổn định và tổng quát hóa: LayerNorm, Dropout và Early Stopping giảm thiểu overfitting, cho kết quả ổn định qua nhiều thử nghiệm.
- Khả năng mở rộng: Mô hình có thể áp dụng cho đồ thị lớn nhờ batching trong PyTorch Geometric và pooling hiệu quả.
- Trực quan hóa đầy đủ: Các biểu đồ loss/accuracy và ma trận nhầm lẫn cung cấp góc nhìn trực quan, hỗ trợ chẩn đoán mô hình.

4. Hạn chế và thách thức

- Thiếu thông tin semantic: Chỉ dùng loại node làm feature, chưa tận dụng giá trị node (value), bối cảnh ngữ nghĩa hoặc embedding từ code.
- Dữ liệu mất cân bằng: Lớp AST lớn chiếm ưu thế, dù đã oversample/undersample thì vẫn khó đạt hiệu năng đồng đều cho lớp nhỏ và trung bình.
- Chi phí tính toán: GAT với nhiều heads tăng chi phí so với GCN thuần, đặc biệt khi số node mỗi AST lớn.
- Giới hạn bài toán: Chỉ giải quyết phân loại theo kích thước; chưa áp dụng cho các tác vụ phức tạp như phân loại chức năng hay phát hiện lỗi.

5. Hướng phát triển trong tương lai

1. Bổ sung đặc trưng ngữ nghĩa
    - Sử dụng embedding từ tên biến, hằng số hoặc comments (Code2Seq, CodeBERT) để cung cấp thông tin semantic.
2. Kiến trúc hybrid/transformer

- Thử nghiệm Graph Transformer hoặc mô hình lai GCN–GAT–Transformer để tăng khả năng học quan hệ phức tạp.

3. Phương pháp cân bằng nâng cao
    - Áp dụng các thuật toán oversampling đồ thị (Graph-SMOTE) hoặc weighted loss tự động điều chỉnh trong quá trình huấn luyện.
4. Mở rộng bài toán
    - Áp dụng phân loại AST cho phân loại chức năng, phát hiện bug, hoặc code clone detection nhằm đánh giá tính linh hoạt và hiệu quả thực tiễn.
5. Tối ưu hóa hiệu năng
    - Sử dụng sparse attention, sampling (GraphSAGE) hoặc pruning layer không quan trọng để giảm độ phức tạp và tăng tốc huấn luyện.

Những hướng đi này hứa hẹn không chỉ nâng cao chất lượng phân loại AST mà còn mở rộng ứng dụng mô hình GNN trong phân tích mã nguồn, hỗ trợ công cụ phát triển phần mềm thông minh hơn và đáng tin cậy hơn.

TÀI LIỆU THAM KHẢO

1. T. N. Kipf và M. Welling, “Semi-Supervised Classification with Graph Convolutional Networks,” *Proc. Int. Conf. Learn. Represent.* (ICLR), 2017.

2. P. Veličković et al., “Graph Attention Networks,” *Proc. Int. Conf. Learn. Represent.* (ICLR), 2018.

3. M. Fey và J. E. Lenssen, “Fast Graph Representation Learning with PyTorch Geometric,” *arXiv preprint arXiv:1903.02428*, 2019.

4. SRI Lab, ETH Zürich, “Python150k AST Dataset,” GitHub repository. Available online: https://github.com/SRI-Lab/python150k (truy cập: 13-05-2025).

5. “PyTorch Geometric Documentation,” https://pytorch-geometric.readthedocs.io (truy cập: 13-05-2025).

6. “NetworkX Documentation,” Software for Complex Networks — NetworkX 3.4.2 documentation (truy cập: 13-05-2025).

7. F. Pedregosa et al., *Scikit-Learn: Machine Learning in Python*, JMLR 12, pp. 2825–2830, 2011. https://scikit-learn.org (truy cập: 13-05-2025).

8. “Abstract Syntax Tree,” *Wikipedia*, https://en.wikipedia.org/wiki/Abstract_syntax_tree (truy cập: 13-05-2025).

10. “AST in Compilers,” DevZery.com, Efficiency Leading API Testing Software Automation Tools (truy cập: 13-05-2025).

11. “Introduction to AST,” GeeksforGeeks.org, GeeksforGeeks | Your All-in-One Learning Portal (truy cập: 13-05-2025).

12. SonarSource, *SonarQube and SonarLint Documentation*, Code Quality, Security & Static Analysis Tool with SonarQube | Sonar (truy cập: 13-05-2025).

13. Checkmarx, *Coverity Static Analysis*, Synopsys, Application Security Software (AppSec) | Black Duck (truy cập: 13-05-2025).

14. L. Allamanis et al., “Code2Seq: Generating Sequences from Structured Representations of Code,” *arXiv preprint arXiv:1808.01400*, 2018.

15. M. Zhang et al., “CodeBERT: A Pre-Trained Model for Programming and Natural Languages,” *arXiv preprint arXiv:2002.08155*, 2020.

16. A. Paszke et al., *PyTorch Documentation*, PyTorch documentation — PyTorch 2.7 documentation (truy cập: 13-05-2025).

17. J. D. Hunter, “Matplotlib: A 2D Graphics Environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.