TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN, ĐHQG-HCM

KHOA CÔNG NGHỆ THÔNG TIN

---○🕮○---

BÁO CÁO ĐỒ ÁN KIẾN TRÚC PHẦN MỀM

Neuron.io

Website thiết kế kiến trúc mạng học sâu bằng đồ thị và sinh ra mã nguồn tự động

Sinh viên thực hiện: 19120106 - Nguyễn Ngọc Khôi Nguyên

19120424 – Phan Nguyễn Thanh Tùng

19120535 – Phạm Quang Huy

Giáo viên phụ trách: Trần Minh Triết

Nguyễn Đức Huy

Lê Khánh Duy

Trần Văn Quý

Thành phố Hồ Chí Minh – Năm 2023

Mục lục

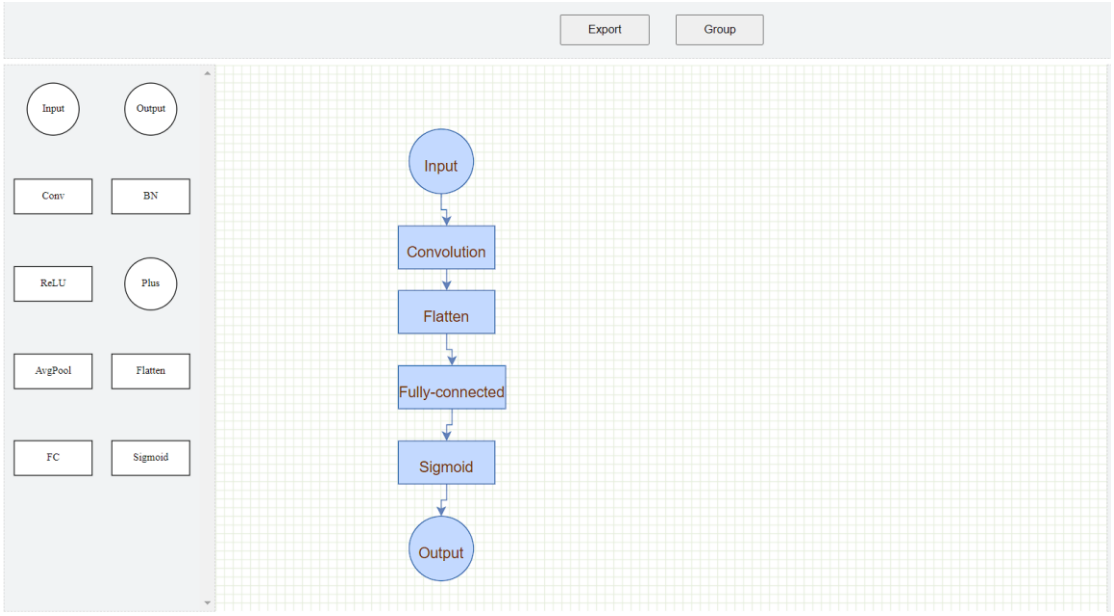
I. Mô tả đề tài

1. Tên đề tài

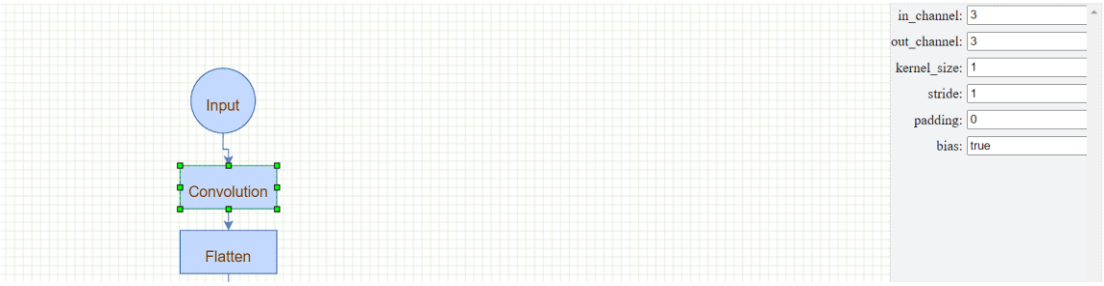
Trang web hỗ trợ xây dựng mô hình học sâu bằng giao diện kéo thả.

2. Chức năng chính

- Kéo thả các node xử lý và nối chúng lại với nhau để tạo thành một mô hình hoàn chỉnh (hiện chỉ hỗ trợ tạo các node "Input", "Output", "Convolution", "Batch Normalization", "ReLU", "Element-wise Plus", "Average Pooling", "Flatten", "Fully-connected" và "Sigmoid")



- Chọn 1 node: hiện bảng chỉnh sửa thông số node đó:



- Nhấn nút "Group": Gộp nhóm các node lại tạo thành 1 composite node để tái sử dụng.

- Nhấn nút “Export”: Xuất mô hình dưới dạng code Pytorch [1]

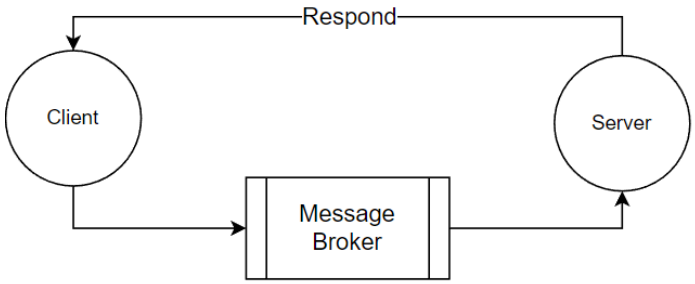
```python
import torch
import torch.nn as nn


class ResNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.Conv2d_4 = nn.Conv2d(in_channels=3, out_channels=3, kernel_size=1, stride=1, padding=0, bias=True)
        self.BatchNorm2d_5 = nn.BatchNorm2d(num_features=3)
        self.ReLU_6 = nn.ReLU()
        self.Conv2d_7 = nn.Conv2d(in_channels=3, out_channels=3, kernel_size=1, stride=1, padding=0, bias=True)
        self.BatchNorm2d_8 = nn.BatchNorm2d(num_features=3)
        self.AdaptiveAvgPool2d_20 = nn.AdaptiveAvgPool2d(output_size=[1, 1])
        self.Linear_22 = nn.Linear(in_features=100, out_features=1)
        self.Sigmoid_23 = nn.Sigmoid()

    def forward(self, x0):
        x1 = x0
        x0 = self.Conv2d_4(x0)
        x0 = self.BatchNorm2d_5(x0)
        x0 = self.ReLU_6(x0)
        x0 = self.Conv2d_7(x0)
        x0 = self.BatchNorm2d_8(x0)
        x0 = torch.add(x1, x0, )
        x0 = self.AdaptiveAvgPool2d_20(x0)
        x0 = torch.flatten(x0, start_dim=1)
        x0 = self.Linear_22(x0)
        x0 = self.Sigmoid_23(x0)
        return x0
```

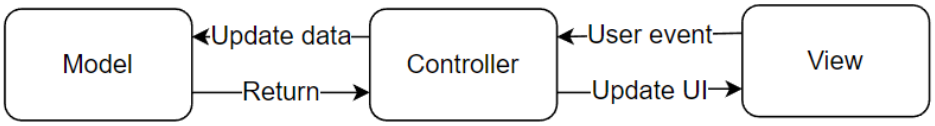
II. Kỹ thuật và kiến trúc

1. Client – Server



1.1. Client:

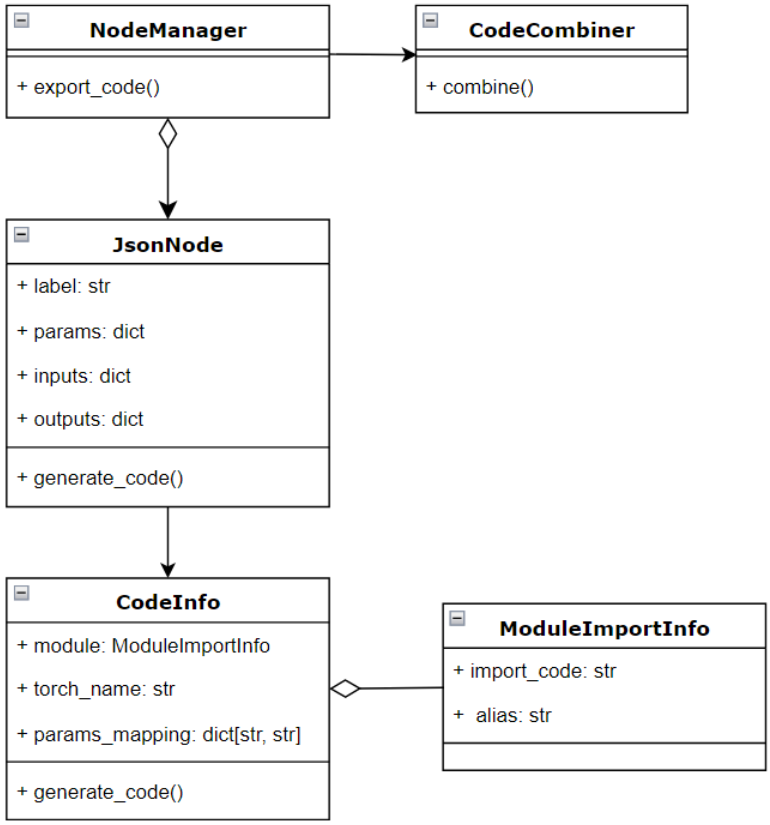
- Hiển thị giao diện và bắt các sự kiện từ người dùng, tuỳ sự kiện mà sẽ xử lý tại Client hoặc gửi lên cho Server xử lý.
- Sử dụng thư viện mxGraph [2] để xây dựng mô hình.
- Thiết kế theo kiến trúc MVC:



  - View: Hiển thị cho người dùng: **HTML**, **CSS**, **mxGraphView**.
  - Controller: Bắt và điều hướng các sự kiện từ người dùng cho phần Model hoặc gửi đến Server: **Javascript**, **mxGraphHandler**.
  - Model: Lưu trữ thông tin trạng thái của các thành phần trong giao diện: **mxGraphModel** (thông tin toàn bộ đồ thị), **mxCell** (thông tin 1 cạnh hoặc góc của mô hình, 1 đỉnh là 1 node trong mô hình).

1.2. Server:

- Gửi thông tin các node cho Client và nhận xử lý chuyển đổi mô hình sang code Pytorch.
- Sơ đồ lớp:



- **JsonNode**: lưu trữ thông tin từng node đọc từ json (label, params, inputs, outputs,…)
- **NodeManager**: Lớp quản lý tất cả **JsonNode** của mô hình, đọc dữ liệu đầu vào và khởi tạo tất cả node. Dùng để sinh ra mã nguồn.
- **CodeCombiner**: Lớp phụ trợ sử dụng bởi NodeManger để tổng hợp code sinh ra từ các **JsonNode** trong **NodeManager** thành kết quả cuối cùng.
- **CodeInfo**: Dùng để chuyển đổi giữa dữ liệu node của đồ thị sang thông tin pytorch tương ứng. Mỗi lớp con kế thừa lớp này sẽ biểu diễn input, output, một hàm, hoặc một lớp tương ứng trong pytorch.
- **ModuleImportInfo**: Lớp để lưu trữ thông tin import các thư viện, module tương ứng với lớp **CodeInfo**.

1.3. Message Broker

- Client lưu trữ dồ thị trong một object thuộc class **mxGraphModel**.
- Server chỉ nhận dữ liệu dạng JSON để tiện cho việc kiểm thử và mở rộng.
- Message Broker đảm nhiệm chuyển đổi 1 object thuộc class **mxGraphModel** thành danh sách các Node dưới dạng JSON.
- Cấu trúc của một Node bao gồm:
    - Id: dùng để định danh node
    - Inputs: một dictionary chứa các node đầu vào của node này, các key của dictionary này là tên của các node đầu vào và được sinh tự động để phục các node cần đảm bảo thứ tự của các node đầu vào để thực thi. Ví dụ: trong phép trừ a – b = c, ta cần đảm báo thứ tự a là đầu vào thứ nhất và b là đầu vào thứ hai.
    - Ouputs: một dictionary chứa các node đầu ra của node này, các key của dictionảy này là tên của các node đầu vào và được sinh tự động để truy xuất node và sử dụng như đầu vào của một node khác

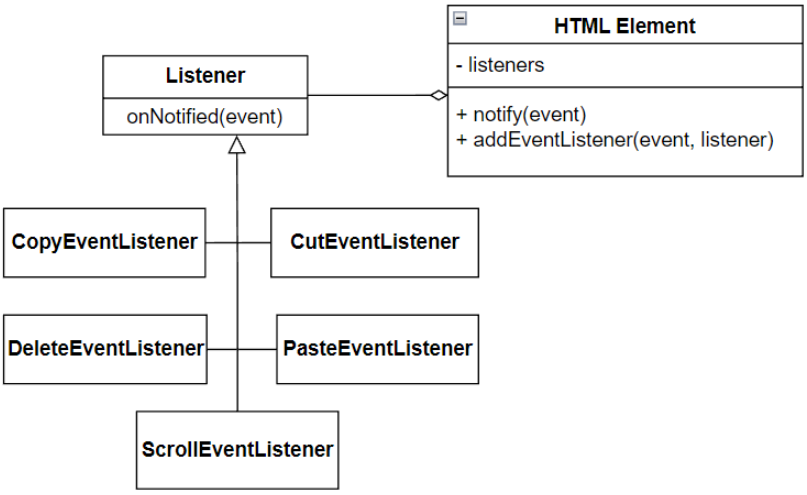
*** Cơ chế sinh tên node tự động đảm bảo việc tên của node khi node là node đầu vào của một node bất kỳ luôn giống với tên của node khi node là node đầu ra của một node bất kỳ.

2. Flexible Object

- Mỗi loại node khác nhau có các trường (tham số) khác nhau. Gộp chung tất cả trường lại thành 1 trường chung duy nhất gọi là “params”.
- Tuỳ vào loại object mà trường “params” này sẽ chứa các keys khác nhau.
    - Convolution: in_channel, out_channel, kernel_size, stride, padding, bias
    - Average Pooling: output_size
    - Batch Normalization: channel
    - Flatten: start_dim
    - Fully-connect: in_dim, out_dim
    - Composite: children
- Các node không có tham số trường “params” sẽ để trống. Ví dụ:
    - Input node
    - Output node
    - Element-wise plus
    - ReLU

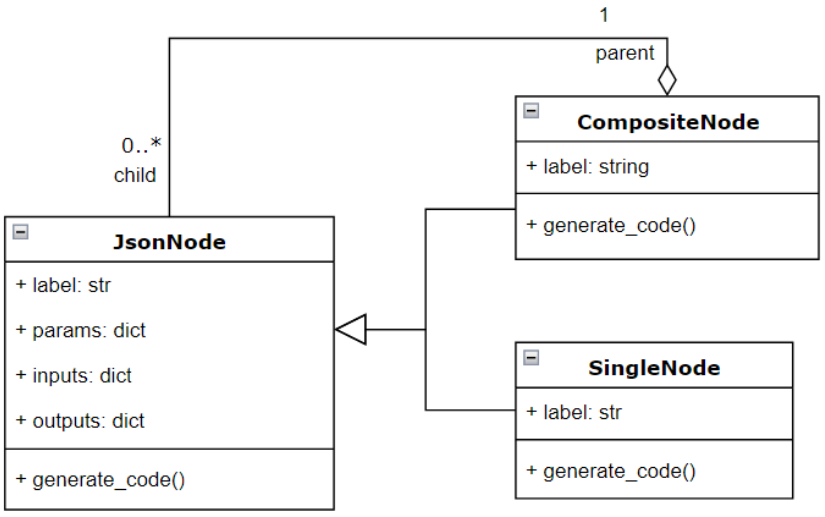
3. Design pattern

3.1. Observer



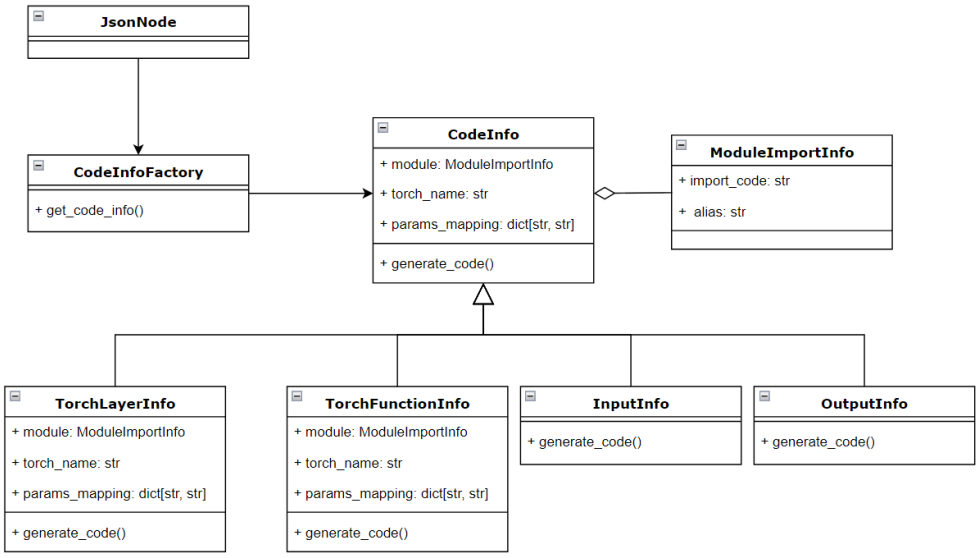
- Các lớp **…EventListener** nghe các sự kiện Copy, Cut, Delete, Paste, Scroll từ giao diện HTML để gọi các xử lý tương ứng.

3.2. Composite



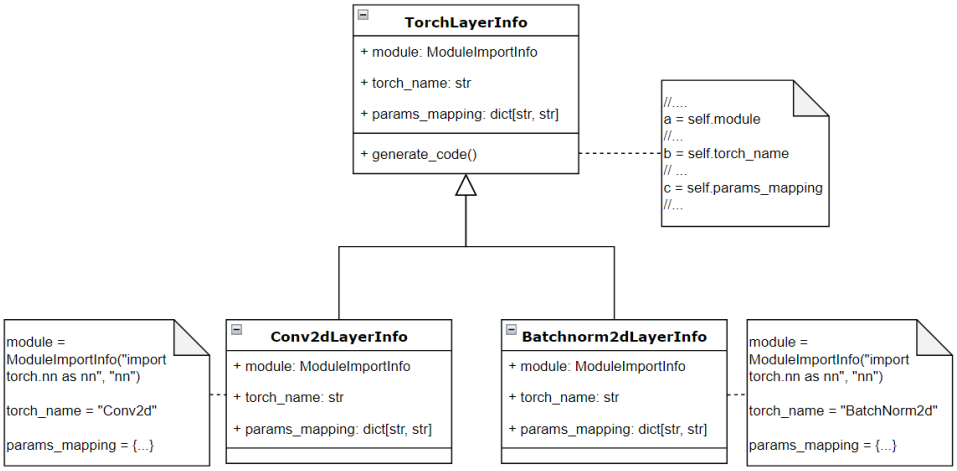
- Hệ thống hỗ trợ người dùng gộp các node con lại để tạo thành một node lớn (composite node) cho mục đích đơn giản hóa bản vẽ và tái sử dụng. Server áp dụng Composite Pattern để giải quyết vấn đề này. **CompositeNode** sẽ chứa danh sách các node con của nó và generate_code bằng cách gọi generate_code của tất cả node con.

3.1. Factory



- Server sử dụng Factory Pattern để lớp **JsonNode** lấy được lớp **CodeInfo** tương ứng với label của nó và sử dụng để sinh ra mã nguồn.

3.1. Template



- Các lớp con của **CodeInfo** là **TorchLayerInfo** (đại diện cho một lớp của pytorch) và **TorchFunctionInfo** (đại diện cho một hàm của pytorch) đều có hàm generate_code riêng và sử dụng thông tin module, torch_name, params_mapping của chính nó. Các lớp con của hai lớp này sẽ viết lại các thông tin này.

III. Demo

- Link demo youtube

IV. Hướng phát triển

1. Chỉnh sửa composite node

- Cho phép người dùng chỉnh sửa tham số của từng node con trong **CompositeNode** để tiện cho việc tái sử dụng.

2. Lưu và tải mô hình từ file

- Việc sử dụng Composite Pattern cho phép hệ thống coi một mô hình đã xây dựng như một **CompositeNode**. Việc hỗ trợ lưu lại mô hình đã xây dựng của cho phép người dùng tự tạo các thiết kế của riêng mình và tái sử dụng chứ không cần phải xây lại từ đầu mỗi lần.

3. Chạy thử mô hình

- Với châm ngôn "trực quan hóa kiến trúc để dễ dàng quan sát", bước phát triển tiếp theo của hệ thống sẽ hỗ trợ người dùng chạy thử mô hình của họ vẽ ra và quan sát kết quả. Các trọng số của mô hình có thể khởi tạo ngẫu nhiên hoặc hỗ trợ người dùng tải lên.

4. Cho phép tạo custom node

- Việc chỉ cho phép sử dụng những node có sẵn từ server sẽ làm giới hạn khả năng sáng tạo các mô hình học sâu. Nhờ trường "params" là một flexible object, hệ thống có thể cho người dùng tạo ra một custom node với các tham số tuỳ ý cùng với đoạn lệnh xử lý để server có thể thực thi.

5. Clean code

- Cải thiện mã nguồn sinh ra để "clean" hơn (giảm tên biến thừa, gộp các composite thành các hàm/class riêng, tạo document cho code,…)

V. Tham khảo

[1] Pytorch: thư viện Python hỗ trợ xây dựng mạng học sâu.
[2] mxGraph: thư viện Javascript hỗ trợ xây dựng đồ thị bằng cách kéo thả.