

**ĐẠI HỌC BÁCH KHOA HÀ NỘI**

VIỆN ĐIỆN TỬ VIỄN THÔNG

□□&□□

**ĐỒ ÁN TỐT NGHIỆP CỬ NHÂN**

**Đề tài:**

**MÔ PHỎNG PHẦN CỨNG CHO MÔ HÌNH HỌC MÁY TSMAE BẰNG NGÔN NGỮ BẬC CAO**

**ĐẠI HỌC BÁCH KHOA HÀ NỘI**

Giảng viên hướng dẫn: Hoàng Mạnh Thắng

Chữ ký của GVHD

Sinh viên thực hiện: Lưu Khánh Duy - 20213847

Lớp ĐIỆN TỬ 03-K66

**Hà Nội 6/2025**

**ĐÁNH GIÁ ĐỒ ÁN TỐT NGHIỆP**

**(DÀNH CHO CÁN BỘ THÀNH VIÊN HỘI ĐỒNG)**

Hội đồng số:

Họ tên SV: Lưu Khánh Duy MSSV:20213847

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Tiêu chí**  (Điểm tối đa) | **Hướng dẫn đánh giá tiêu chí** | **Điểm tiêu chí** |
| 1 | **Chất lượng**  **slides/Bản vẽ kỹ thuật**  (1,5 điểm) | Sử dụng các minh họa hỗ trợ: Hình ảnh, biểu đồ rõ nét và phù hợp, dễ hiểu |  |
| Không quá nhiều từ, biết sử dụng từ khoá; bố cục logic, có đánh số trang |
| 2 | **Kỹ năng thuyết trình**  (1,5 điểm) | Tự tin, làm chủ nội dung trình bày, đúng thời gian quy định |  |
| Dễ hiểu, dễ theo dõi, lô-gic, lôi cuốn. |
| 3 | **Nội dung và kết quả đạt được** (4 điểm) | Nêu rõ tính cấp thiết, ý nghĩa khoa học và thực tiễn của đề tài, các vấn đề và các giả thuyết, phạm vi ứng dụng của đề tài. Thực hiện đầy đủ quy trình nghiên cứu: Đặt vấn đề, mục tiêu đề ra, phương pháp nghiên cứu/ giải quyết vấn đề, kết quả đạt được, đánh giá và kết luận. |  |
| Nội dung và kết quả được trình bày một cách logic và hợp lý, được phân tích và đánh giá thỏa đáng. Biện luận phân tích kết quả mô phỏng/  phần mềm/ thực nghiệm, so sánh kết quả đạt được với kết quả trước đó có liên quan. |
| Chỉ rõ phù hợp giữa kết quả đạt được và mục tiêu ban đầu đề ra đồng thời cung cấp lập luận để đề xuất hướng giải quyết có thể thực hiện trong tương lai. Hàm lượng khoa học/ độ phức tạp cao, có tính mới/ tính sáng tạo trong nội dung và kết quả đồ án. |
| 4 | **Trả lời câu hỏi**  (2,5 điểm) | Trả lời ngắn gọn, chính xác, đi thẳng vào vấn đề của câu hỏi. |  |
| Nắm vững kiến thức cơ bản liên quan đến lĩnh vực nghiên cứu/ công việc của đồ án. |
| 5 | **Điểm thành tích** (1 điểm) | Có bài báo KH được đăng hoặc chấp nhận đăng/ đạt giải SV NCKH giải 3 cấp Trường trở lên/ Các giải thưởng khoa học trong nước, quốc tế từ giải 3 trở lên/ Có đăng ký bằng phát minh sáng chế. **(1 điểm)** |  |
| Được báo cáo tại hội đồng cấp Trường trong hội nghị SV NCKH nhưng không đạt giải từ giải 3 trở lên/ Đạt giải khuyến khích trong cuộc  thi khoa học trong nước, quốc tế/ Kết quả đồ án là sản phẩm ứng dụng có tính hoàn thiện cao, yêu cầu khối lượng thực hiện lớn. **(0,5 điểm)** |
|  |  | **Điểm tổng các tiêu chí:** |  |
|  | | **Điểm bảo vệ:** |  |

Cán bộ thành viên HD:

**Cán bộ thành viên HĐ**(Ký và ghi rõ họ tên)

**LỜI NÓI MỞ ĐẦU**

Trong bối cảnh các hệ thống trí tuệ nhân tạo (AI) ngày càng phát triển mạnh mẽ và được triển khai trên nhiều nền tảng phần cứng khác nhau, việc nghiên cứu và hiện thực hóa mô hình AI trên các hệ thống nhúng, đặc biệt là FPGA/ASIC, trở thành một hướng đi đầy tiềm năng.

Trong đồ án này, tôi lựa chọn sử dụng SystemC - một chuẩn mô tả hệ thống phần cứng ở mức cao - với mục tiêu mô phỏng hoạt động của mô hình AI, cụ thể là mô hình TSMAE (Time-Series Memory-Augmented Autoencoder), trên nền tảng phần cứng. Mục tiêu cuối cùng là tạo ra một kiến trúc có thể mở rộng và triển khai thực tế trong các hệ thống cảnh báo sớm dựa trên chuỗi thời gian như cảm biến sinh học, công nghiệp IoT, v.v.

Trong suốt quá trình thực hiện đồ án, em đã nhận được sự quan tâm, hướng dẫn tận tình từ **Thầy Hoàng Mạnh Thắng** – người đã luôn theo sát và hỗ trợ em vượt qua những khó khăn chuyên môn. Chính thầy là người đưa ra đề tài và định hướng cho em đồng thời cũng luôn sát sao với việc báo cáo hang tuần đồng thời đưa ra rất nhiều lời khuyên bổ ích cho việc làm đồ án và cả hành trình đi làm sau này.

Em xin chân thành cảm ơn.

**LỜI CAM ĐOAN**

Tôi tên **Lưu Khánh Duy**, mã số sinh viên 20213847, sinh viên lớp Điện tử 03, khóa 66. Người hướng dẫn là **Thầy Hoàng Mạnh Thắng**. Tôi xin cam đoan toàn bộ nội dung được trình bày trong đồ án “Mô phỏng và triển khai mô hình học máy Time-Series Memory Autoencoder” là kết quả của quá trình tìm hiểu và nghiên cứu của tôi. Mọi thông tin trích dẫn đều tuân thủ quy định về sở hữu trí tuệ; các tài liệu tham khảo được liệt kê rõ ràng. Tôi xin chịu hoàn toàn trách nhiệm với những nội dung được viết trong đồ án này.

Hà Nội, ngày 27 tháng 6 năm 2025

**Người cam đoan**

**Lưu Khánh Duy**

**MỤC LỤC**

[DANH MỤC HÌNH VẼ 7](#_Toc201611551)

[DANH MỤC BẢNG 8](#_Toc201611552)

[CHƯƠNG 1. GIỚI THIỆU 1](#_Toc201611553)

[1.1 Đặt vấn đề 1](#_Toc201611554)

[1.2 Giới thiệu đề tài 1](#_Toc201611555)

[1.2.1 Nội dung 1](#_Toc201611556)

[1.2.2 Mục tiêu 2](#_Toc201611557)

[1.2.3 Nhiệm vụ nghiên cứu 2](#_Toc201611558)

[1.2.4 Phương pháp nghiên cứu 2](#_Toc201611559)

[1.3 Kết luận 3](#_Toc201611560)

[CHƯƠNG 2. CƠ SỞ LÝ THUYẾT VÀ CÁC NGHIÊN CỨU LIÊN QUAN 4](#_Toc201611561)

[2.1 Cơ sở lý thuyết về AI và mạng neural 4](#_Toc201611562)

[2.1.1 Recurrent Neural Network (RNN) 7](#_Toc201611563)

[2.1.2 LSTM(Long short term memory) 8](#_Toc201611564)

[2.1.3 So sánh và lựa chọn mô hình phù hợp 9](#_Toc201611565)

[2.2 Giới thiệu về TSMAE 10](#_Toc201611566)

[*2.2.1* Bộ mã hóa LSTM (Encoder) 10](#_Toc201611567)

[*2.2.2* Bộ nhớ tăng cường (Memory Module) 10](#_Toc201611568)

[*2.2.3* Bộ giải mã LSTM (Decoder) 11](#_Toc201611569)

[2.3 Kiến trúc của TSMAE 11](#_Toc201611570)

[2.3.1 LSTM Encoder 12](#_Toc201611571)

[*2.3.2* Memory module 14](#_Toc201611572)

[2.3.3 LSTM Decoder 15](#_Toc201611573)

[2.3.4 Loss Function 16](#_Toc201611574)

[2.4 Triển khai TSMAE bằng python 16](#_Toc201611575)

[2.4.1 Mô hình TSMAE 16](#_Toc201611576)

[2.4.2 Data Preprocessing 17](#_Toc201611577)

[2.4.3 Memory Module implementation 19](#_Toc201611578)

[3.3.4. TSMAE implementation 21](#_Toc201611579)

[CHƯƠNG 3. XÂY DỰNG MÔ HÌNH TSMAE TRÊN PHẦN CỨNG 26](#_Toc201611580)

[3.1 Tách trọng số từ mô hình bằng Python 26](#_Toc201611581)

[3.2 Kiến trúc của TSMAE khi dùng systemC 28](#_Toc201611582)

[3.3 LSTM\_neural 29](#_Toc201611583)

[3.3.1 Module LSTM\_neural 29](#_Toc201611584)

[3.3.2 LSTM\_neural diagram flow 31](#_Toc201611585)

[3.3.3 Code cho LSTM\_neural 32](#_Toc201611586)

[3.4 Memory\_module 36](#_Toc201611587)

[3.4.1 Module Memory\_module 36](#_Toc201611588)

[3.4.2 MEMORY\_module flow diagram 38](#_Toc201611589)

[3.4.3 Code cho Memory\_module 39](#_Toc201611590)

[3.5 Driver 42](#_Toc201611591)

[3.5.1 Module Driver 42](#_Toc201611592)

[3.5.2 Driver flow diagram 43](#_Toc201611593)

[3.5.3 Code cho module Driver 44](#_Toc201611594)

[3.6 Output\_Layer 46](#_Toc201611595)

[3.6.1 Module Output\_Layer 46](#_Toc201611596)

[3.6.2 Output\_layer flow diagram 47](#_Toc201611597)

[3.6.3 Code cho Output\_layer 48](#_Toc201611598)

[3.7 TSMAE 50](#_Toc201611599)

[3.7.1 Code TSMAE 50](#_Toc201611600)

[3.7.2 Giải thích code 56](#_Toc201611601)

[3.8 Tổng kết chương 57](#_Toc201611602)

[CHƯƠNG 4. KIỂM TRA VÀ MÔ PHỎNG HỆ THỐNG 58](#_Toc201611603)

[4.1 Testbench cho systemC 58](#_Toc201611604)

[4.1.1 Code cho testbench 58](#_Toc201611605)

[4.1.2 Giải thích code 63](#_Toc201611606)

[4.1.3 Kết quả sau khi mô phỏng 64](#_Toc201611607)

[4.2 Testbench cho python 64](#_Toc201611608)

[4.2.1 Code cho testbench 64](#_Toc201611609)

[4.2.2 Giải thích Code 66](#_Toc201611610)

[4.2.3 Kết quả sau khi mô phỏng 66](#_Toc201611611)

[4.3 So sánh kết quả giữa python và systemC 67](#_Toc201611612)

[4.4 Tổng kết chương **Error! Bookmark not defined.**](#_Toc201611613)

[CHƯƠNG 5. KẾT LUẬN 69](#_Toc201611614)

[4. TÀI LIỆU THAM KHẢO 71](#_Toc201611615)

# DANH MỤC HÌNH VẼ

[Hình 1: Sơ đồ khối một hệ thống ANN cơ bản 4](#_Toc201438822)

[Hình 2: Mô hình một neural cơ bản của McCulloch và Pitts 5](#_Toc201438823)

[Hình 3: Mô hình mạng RNN 7](#_Toc201438824)

[Hình 4: Cấu trúc một state của LSTM 8](#_Toc201438825)

[Hình 5: Kiến trúc của TSMAE 11](#_Toc201438826)

[Hình 6: Kiến trúc của TSMAE 12](#_Toc201438827)

[Hình 7: Traning Loss 24](#_Toc201438828)

[Hình 8: kết quả sau khi tái tạo cho dữ liệu bình thường và dữ liệu không bình thường 24](#_Toc201438829)

[Hình 9: Kiến trúc của TSMAE dùng systemC 28](#_Toc201438830)

[Hình 10: Module LSTM\_neural 29](#_Toc201438831)

[Hình 11: LSTM\_neural flow diagram 31](#_Toc201438832)

[Hình 12: Module MEMORY\_module 36](#_Toc201438833)

[Hình 13: MEMORY\_module flow diagram 38](#_Toc201438834)

[Hình 14: Module Driver 42](#_Toc201438835)

[Hình 15: Driver flow diagram 43](#_Toc201438836)

[Hình 16: Module Output\_layer 46](#_Toc201438837)

[Hình 17: Output\_layer flow diagram 47](#_Toc201438838)

[Hình 18: Chuỗi dữ liệu ban đầu 67](#_Toc201438839)

[Hình 19: Chuỗi dữ liệu được tái tạo bằng Python 67](#_Toc201438840)

[Hình 20: Chuỗi dữ liệu được tái tạo bằng SystemC 68](#_Toc201438841)

[Hình 21:Thời gian và tài nguyên máy tính khi chạy một chuỗi 68](#_Toc201438841)

# DANH MỤC BẢNG

[Bảng 1: Các phương pháp kết hợp đầu vào 5](#_Toc201438842)

[Bảng 2: Các hàm kích hoạt 6](#_Toc201438843)

[Bảng 3: So sánh các mô hình học sâu dựa trên đầu vào và số lượng tham số 9](#_Toc201438844)

[Bảng 4: Chức năng các chân vào ra của LSTM\_neural 29](#_Toc201438845)

[Bảng 5: Các cổng vào ra của Memory\_module 37](#_Toc201438846)

[Bảng 6: Các cổng vào ra của module Driver 42](#_Toc201438847)

[Bảng 7: Các cổng vào ra của module Driver 46](#_Toc201438848)

[Bảng 8:Lượng phần cứng cần thiết cho mô hình TSMAE 46](#_Toc201438848)

# GIỚI THIỆU

## Đặt vấn đề

Trong kỷ nguyên của cuộc cách mạng công nghiệp 4.0, trí tuệ nhân tạo (AI) đã trở thành một trong những công nghệ trọng yếu, đóng vai trò cốt lõi trong việc phân tích, dự đoán và ra quyết định tự động từ dữ liệu lớn. Đặc biệt, trong các hệ thống nhúng và thiết bị IoT, nhu cầu tích hợp các mô hình AI vào phần cứng ngày càng tăng nhằm giảm độ trễ, tiết kiệm năng lượng và tăng tính sẵn sàng. Theo công ty McKinsey[1] IoT có tiềm năng tạo ra một sức ảnh hưởng kinh tế từ 3.9 triệu tỉ đô đến 11.1 triệu tỉ đô hàng năm từ năm 2025

Một trong những hướng đi quan trọng là hiện thực hóa các mô hình học sâu (Deep Learning) như LSTM hay Autoencoder trực tiếp trên nền tảng phần cứng. Tuy nhiên, phần lớn các mô hình AI hiện nay được triển khai trên GPU hoặc CPU, vốn không phù hợp để đưa vào các thiết bị nhúng có tài nguyên hạn chế.

Bên cạnh đó, trong lĩnh vực phát hiện bất thường (anomaly detection), các mô hình truyền thống thường không thể phát hiện được các mẫu bất thường phức tạp ẩn trong chuỗi thời gian. Mô hình **TSMAE (Time-Series Memory-Augmented Autoencoder)** là một giải pháp hiện đại, kết hợp khả năng ghi nhớ chuỗi dài của LSTM với cơ chế bộ nhớ mở rộng, giúp cải thiện đáng kể độ chính xác trong phát hiện bất thường.

Tuy nhiên, để có thể triển khai TSMAE trên phần cứng, cần có một phương pháp mô tả kiến trúc logic ở mức hệ thống, dễ kiểm thử, đồng thời tương thích với các công cụ tổng hợp phần cứng.Khi sử dụng ngôn ngữ HDL sẽ giúp ích cho việc tổng hợp sang phần cứng tuy nhiên sẽ rất khó để có thể thiết kế ra được một hệ thống AI hoàn chỉnh do nhiều yếu tố như là số thực hay các hàm kích hoạt phức tạp. Công cụ **SystemC** là một trong những chuẩn công nghiệp hỗ trợ mô tả hệ thống ở cấp độ hành vi, phù hợp để mô phỏng, kiểm chứng và tiến tới tổng hợp phần cứng (synthesis) cho các thiết kế AI.

## Giới thiệu đề tài

### Nội dung

Trong thời đại dữ liệu lớn và kết nối vạn vật (IoT) phát triển mạnh mẽ, lượng dữ liệu thu thập được từ các cảm biến, thiết bị đầu cuối hay hệ thống giám sát ngày càng lớn và đa dạng. Một trong những bài toán thực tiễn quan trọng phát sinh từ đó là **phát hiện bất thường** (Anomaly Detection) trong các hệ thống IoT [2][3]– nhằm cảnh báo kịp thời các dấu hiệu nguy hiểm hoặc sai lệch tiềm ẩn từ dữ liệu chuỗi thời gian.

Để giải quyết bài toán này một cách hiệu quả, các mô hình học sâu như LSTM hay Autoencoder đã được áp dụng rộng rãi. Tuy nhiên, những mô hình truyền thống này thường gặp khó khăn trong việc ghi nhớ thông tin dài hạn hoặc biểu diễn các mẫu bất thường phức tạp. Gần đây, mô hình **Time-Series Memory-Augmented Autoencoder (TSMAE)** đã được đề xuất như một cải tiến vượt trội, kết hợp sức mạnh của mạng LSTM với một cơ chế bộ nhớ động để tăng khả năng ghi nhớ và phát hiện bất thường trong chuỗi thời gian.

Song song với sự phát triển của mô hình, nhu cầu triển khai mô hình AI trên phần cứng (như FPGA hoặc SoC) ngày càng cấp thiết nhằm phục vụ các ứng dụng thời gian thực, tiết kiệm điện năng và giảm độ trễ xử lý. Tuy nhiên, việc hiện thực hóa các kiến trúc học sâu như TSMAE trên phần cứng là một thách thức lớn do độ phức tạp mô hình và yêu cầu đồng bộ hóa dữ liệu. Đó là lý do vì sao SystemC – một ngôn ngữ mô tả hệ thống phần cứng ở mức hành vi – được sử dụng trong đồ án này để mô phỏng và kiểm thử hoạt động của mô hình TSMAE trên nền tảng logic. Project này sẽ tập trung vào cách th xây dựng TSMAE trên phần cứng.

### Mục tiêu

Tìm hiểu lý thuyết và hiện thực mô hình TSMAE trong python, đánh giá khả năng phát hiện bất thường trên dữ liệu chuỗi thời gian (ECG5000).

Chuyển đổi và mô phỏng mô hình TSMAE bằng ngôn ngữ SystemC, nhằm kiểm chứng tính khả thi trong môi trường phần cứng.

### Nhiệm vụ nghiên cứu

Để đạt được mục tiêu đề ra, đồ án tập trung vào các nhiệm vụ nghiên cứu chính sau:

* + - * Tìm hiểu tổng quan về các khái niệm cốt lõi về trí tuệ nhân tạo (AI), mạng neural nhân tạo (ANN), mạng hồi tiếp(RNN) và mạng nhớ dài hạn (LSTM)
      * Phân tích và thiết kế mô hình TSMAE
      * Thực nghiệm mô hình TSMAE bằng python.
      * Hiện thực mô hình TSMAE bằng ngôn ngữ systemC và thử nghiệm
      * Đánh giá khả năng triển khai phần cứng.

### Phương pháp nghiên cứu

Để thực hiện đồ án này, các phương pháp nghiên cứu sau được áp dụng:

**Phương pháp nghiên cứu tài liệu:** Thu thập và phân tích các tài liệu học thuật, giáo trình, bài báo khoa học, báo cáo kỹ thuật liên quan đến trí tuệ nhân tạo, mạng neural, LSTM, Autoencoder, và đặc biệt là mô hình TSMAE. Tìm hiểu về các chuẩn thiết kế phần cứng ở mức hệ thống như SystemC, phương pháp mô tả hành vi và kỹ thuật mô phỏng theo thời gian thực.

**Phương pháp mô hình hoá:** Thiết kế kiến trúc logic của mô hình TSMAE, bao gồm các thành phần encoder, memory module và decoder. Xây dựng pipeline huấn luyện mô hình dựa trên framework PyTorch, tích hợp các kỹ thuật tối ưu như normalization, sparsification.

**Phương pháp thực nghiệm:** Triển khai mô hình TSMAE trên môi trường Python, huấn luyện với tập dữ liệu ECG5000 (chuỗi thời gian dạng tín hiệu). Và vẽ biểu đồ để so sánh chuỗi dữ liệu đầu ra và đầu vào

**Phương pháp mô phỏng hệ thống:** Đánh giá mức độ chính xác và tính khả thi khi triển khai mô hình vào môi trường phần cứng thông qua việc tái tạo dữ liệu đầu vào và so sánh với kết quả tái tạo khi dùng python.

## Kết luận

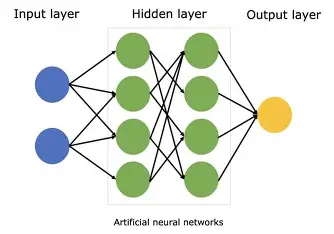
Trong chương này, đồ án đã trình bày các phương pháp nghiên cứu được sử dụng xuyên suốt quá trình thực hiện, bao gồm từ khảo sát lý thuyết, mô hình hóa kiến trúc TSMAE, huấn luyện thực nghiệm, đến mô phỏng phần cứng bằng SystemC. Việc kết hợp giữa phương pháp định tính và định lượng giúp đảm bảo cả tính chính xác kỹ thuật lẫn khả năng triển khai thực tế. Những phương pháp này tạo tiền đề vững chắc cho việc triển khai mô hình trong các chương tiếp theo và đánh giá tính khả thi của đề tài.

# CƠ SỞ LÝ THUYẾT VÀ CÁC NGHIÊN CỨU LIÊN QUAN

## Cơ sở lý thuyết về AI và mạng neural

Mạng neuron là một mô hình điện toán toán học nói chung mô phỏng lại sự hoạt động của hệ thống neuron sinh học. Năm 1943 McCulloch[4], một nhà sinh học chuyên về nghiên cứu neuron, và Pitts, một nhà thống kê đã xuất bản bài báo hội thảo với tiêu đề: “Một phép tính lý tưởng hợp lý của các ý tưởng nội tại trong hoạt động thần kinh” trong Nội san sinh toán học. Bài báo đã gây cảm hứng cho sự phát triển của máy tính số hiện đại sau này, hoặc bộ não điện tử. Vào khoảng thời gian này Frank Rosenblatt đã bị thúc đẩy bởi một bài báo về nghiên cứu sự tính toán của mắt, cuối cùng dẫn đến thế hệ đầu tiên của mạng neuron, biết đến như một perceptron. Phần này cung cấp một cái nhìn ngắn gọn về các mạng neuron nhân tạo. Các chủ đề sẽ được khảo sát kĩ hơn tại các chương sau. Mục đích của chương này là làm nổi bật những khái niệm cơ bản của các mô hình mạng neuron để chuẩn bị cho đọc giả đọc các chương tiếp theo.

Các thành phần cơ bản của mạng neural(ANN) bao gồm



Hình : Sơ đồ khối một hệ thống ANN cơ bản

* + - * **Lớp đầu vào (input layer):** nhận dữ liệu thô từ các cảm biến hay là các thiết bị ngoại vi
      * **Các lớp ẩn(hidden layers):** thực hiện các phép biến đổi phi tuyến tính
      * **Lớp đầu ra:** tạo ra kết quả dự đoán

A diagram of a mathematical equation

Description automatically generated

Hình : Mô hình một neural cơ bản của McCulloch và Pitts

Từ hình ta thấy mỗi neuron gồm có 2 phần: phần net function và hàm kích hoạt. The net funtion được xác định {yj; 1≤j≤N1≤j≤N} như cách các đầu vào mạng được tập hợp như thế nào bên trong neuron. Trong hình, một sự kết hợp tuyến tính trọng số được chấp thuận:

A black and white math symbol

Description automatically generated

{*wj*; 1 ≤ *j* ≤ *N*} là các tham số được hiểu như những trọng số liên kết. Đại lượng θ được gọi là độ chênh lệch và được dùng để mô hình hóa ngưỡng. Trong tài liệu, các phương pháp kết hợp đầu vào mạng khác nhau được đề nghị như bảng dưới đây.

Bảng : Các phương pháp kết hợp đầu vào

A white paper with black text and symbols

Description automatically generated

Ngoài ra còn có các hàm kích hoạt như ReLU, sigmoid, hay tanh được sử dụng để tăng khả năng biểu diễn phi tuyến của mạng. Quá trình huấn luyện mạng neural sử dụng thuật toán lan truyền ngược (Backpropagation) và tối ưu hóa bằng Gradient Descent.

Bảng : Các hàm kích hoạt

A math equations and formulas

Description automatically generated with medium confidence

Ngoài MLP cơ bản, nhiều loại mạng neural đã được phát triển để phục vụ các mục đích khác nhau. Một số mạng phổ biến bao gồm:

* **Convolutional Neural Network (CNN)**: Được sử dụng chủ yếu trong xử lý ảnh và thị giác máy tính. CNN sử dụng các lớp tích chập (convolution) và gộp (pooling) để trích xuất đặc trưng không gian từ ảnh đầu vào. Khả năng học đặc trưng cục bộ và giảm số lượng tham số giúp CNN rất hiệu quả trong các tác vụ như nhận dạng khuôn mặt, phân loại ảnh, v.v.
* **Recurrent Neural Network (RNN)**: Phù hợp với dữ liệu tuần tự như chuỗi thời gian hoặc ngôn ngữ. RNN duy trì trạng thái ẩn (hidden state) qua các bước thời gian, giúp ghi nhớ thông tin ngữ cảnh gần. Tuy nhiên, RNN gặp khó khăn trong việc ghi nhớ dài hạn do hiện tượng vanishing gradient.

Các bài toán như dự báo dữ liệu cảm biến, phân tích tín hiệu sinh học hoặc phát hiện bất thường trong dòng dữ liệu đều có một điểm chung quan trọng: **tính tuần tự và phụ thuộc theo thời gian**. Các phương pháp học máy truyền thống như SVM, Random Forest, hay các mạng MLP (Multilayer Perceptron) đều giả định các đầu vào là độc lập với nhau, nên không thể khai thác mối liên hệ giữa các bước thời gian.

**Recurrent Neural Network (RNN)** được thiết kế đặc biệt để giải quyết vấn đề này. Khác với các mạng feedforward, RNN có khả năng ghi nhớ trạng thái qua các bước thời gian nhờ cơ chế lan truyền trạng thái ẩn từ bước này sang bước tiếp theo. Điều này cho phép RNN học được **ngữ cảnh động** trong chuỗi, một yếu tố then chốt trong các bài toán như:

* Dự báo dữ liệu cảm biến công nghiệp;
* Nhận diện mẫu tín hiệu điện tim (ECG);
* Phát hiện bất thường trong chuỗi hành vi người dùng, dữ liệu mạng.

Mặc dù RNN truyền thống còn tồn tại nhược điểm như khó học được phụ thuộc dài hạn (vanishing gradient), nhưng nó vẫn là nền tảng cho các biến thể nâng cao như LSTM và GRU. Do đó, việc bắt đầu từ kiến trúc RNN giúp làm rõ cách các mô hình chuỗi hoạt động và làm nền để phát triển lên mô hình mạnh hơn như TSMAE.

### Recurrent Neural Network (RNN)

**Recurrent Neural Network (RNN)** là một kiến trúc mạng neural được thiết kế dành riêng cho dữ liệu tuần tự (sequential data), chẳng hạn như chuỗi thời gian, văn bản, âm thanh, hoặc tín hiệu cảm biến. Điểm đặc biệt của RNN là khả năng **ghi nhớ thông tin từ các bước thời gian trước đó**, giúp mô hình học được ngữ cảnh và mối quan hệ trong chuỗi dữ liệu.

Khác với mạng MLP truyền thống – nơi mỗi đầu vào được xử lý độc lập – RNN có một **trạng thái ẩn (hidden state)** được duy trì và cập nhật sau mỗi bước thời gian. Với mỗi bước t, RNN nhận đầu vào xt­­​ và trạng thái ẩn trước đó st-1 sau đó tính toán:

A diagram of a diagram

Description automatically generated

Hình : Mô hình mạng RNN

Công thức:

St = tanh(Wt\*St-1  + U\*St  + b)

yt = V\*St-1  + by

Trong đó:

* Wt,U​,V là các trọng số của mạng;
* St-1  là trạng thái ẩn tại thời điểm t;
* yt ​ là đầu ra tại các thời điểm t;
* b, by  là các bias.

Mạng RNN hoạt động dựa trên nguyên tắc **duy trì trạng thái ẩn (hidden state)** – một bộ nhớ tạm thời giúp ghi nhớ thông tin quá khứ trong quá trình xử lý chuỗi đầu vào. Cụ thể, quá trình diễn ra theo các bước như sau:

***1. Khởi tạo trạng thái ẩn ban đầu***

Trước bước đầu tiên (t=1), trạng thái ẩn St-1 ​ được khởi tạo thường là vector 0 hoặc một giá trị ngẫu nhiên.

***2. Xử lý tuần tự từng phần tử trong chuỗi***

Với mỗi bước thời gian ttt, mạng nhận đầu vào xt (là một phần tử trong chuỗi) và trạng thái ẩn từ bước trước đó St-1  ​, sau đó tính toán:

St =ϕ(Wxh \* xt   + Whh\* St-1 + bt )

Kết quả là trạng thái ẩn mới St ​, chứa thông tin tổng hợp từ quá khứ đến hiện tại.

***3. Sinh đầu ra (nếu có)***

Tại mỗi bước thời gian, RNN có thể sinh ra một đầu ra yty\_tyt​ thông qua hàm:

y t = Why \* ht + by

Đầu ra này có thể dùng để dự đoán từng phần tử tiếp theo (như trong bài toán dịch máy, dự đoán tín hiệu) hoặc dùng để tính lỗi (loss) trong quá trình huấn luyện.

***4. Truyền trạng thái ẩn sang bước kế tiếp***

Trạng thái hth\_tht​ được truyền sang bước t+1, trở thành một phần thông tin đầu vào cho bước sau.

***5. Huấn luyện qua lan truyền ngược theo thời gian (BPTT)***

Trong giai đoạn huấn luyện, mô hình sử dụng thuật toán **Backpropagation Through Time (BPTT)** để lan truyền sai số ngược qua nhiều bước thời gian và cập nhật trọng số.

### LSTM(Long short term memory)

Một biến thể mạnh mẽ hơn của RNN là mạng LSTM (Long Short-Term Memory), được xây dựng nhằm giải quyết nhược điểm khó học các quan hệ phụ thuộc dài hạn của RNN truyền thống. LSTM được thiết kế với cơ chế ghi nhớ thông minh, cho phép lưu giữ thông tin trong thời gian dài mà không cần điều chỉnh quá nhiều trong quá trình huấn luyện. Điều này giúp mô hình tự động chọn lọc và lưu trữ những thông tin quan trọng để sử dụng ở các thời điểm sau.

Hình : Cấu trúc một state của LSTM

A diagram of a flowchart

Description automatically generated

Tương tự RNN, LSTM gồm các khối được lặp lại theo trình tự thời gian. Tuy nhiên, mỗi khối của LSTM có cấu trúc phức tạp hơn: thay vì chỉ có một lớp mạng đơn giản như trong RNN chuẩn, mỗi đơn vị trong LSTM bao gồm tới bốn thành phần chính tương tác chặt chẽ với nhau, nhằm kiểm soát quá trình quên, lưu và truyền thông tin.

Cụ thể, đầu vào của mỗi bước thời gian trong LSTM gồm:

* xt: dữ liệu đầu vào tại thời điểm hiện tại (thường là đặc trưng được trích xuất từ một frame),
* ht-1: trạng thái ẩn từ bước trước,
* ct-1: trạng thái bộ nhớ từ bước trước.

LSTM sử dụng ba cổng chính:

* **Cổng quên (forget gate)** – điều khiển thông tin nào nên bị loại bỏ khỏi bộ nhớ,
* **Cổng đầu vào (input gate)** – xác định thông tin mới nào nên được ghi nhớ,
* **Cổng đầu ra (output gate)** – quyết định nội dung nào sẽ được truyền sang bước tiếp theo.

Trung tâm của LSTM là "dòng chảy bộ nhớ" từ ct-1 đến ct , giúp lưu giữ các thông tin quan trọng trong suốt chuỗi thời gian. Đây chính là yếu tố tạo nên khả năng ghi nhớ dài hạn – điểm mạnh nổi bật của LSTM. Kết hợp với cơ chế lưu giữ tạm thời (short-term memory) giống RNN, LSTM trở thành công cụ đặc biệt phù hợp cho các bài toán xử lý chuỗi dài như chuỗi khung hình video trong nhận diện hành động.

### So sánh và lựa chọn mô hình phù hợp

Bảng : So sánh các mô hình học sâu dựa trên đầu vào và số lượng tham số

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tiêu chí** | **CNN Autoencoder** | **RNN Autoencoder** | **LSTM Autoencoder** | **GCN Autoencoder** |
| **Dữ liệu phù hợp** | Dữ liệu hình ảnh, chuỗi có cấu trúc không tuần tự | Dữ liệu chuỗi thời gian ngắn, có tuần tự | Dữ liệu chuỗi dài có phụ thuộc xa | Dữ liệu dạng đồ thị (graph), mạng phức tạp |
| **Khả năng học phụ thuộc theo thời gian** | Thấp (không học được thứ tự) | Trung bình (phụ thuộc gần) | Cao (ghi nhớ phụ thuộc dài hạn) | Không chuyên xử lý chuỗi thời gian |
| **Khả năng tái tạo dữ liệu phi tuyến** | Tốt trong xử lý ảnh hoặc tín hiệu 2D | Hạn chế nếu chuỗi quá dài | Tốt với chuỗi phức tạp, đa nhịp | Phụ thuộc vào cấu trúc đồ thị đầu vào |
| **Yêu cầu tài nguyên tính toán** | Trung bình | Thấp đến trung bình | Cao hơn RNN do thêm cổng điều khiển | Cao, phụ thuộc vào cấu trúc graph |
| **Ứng dụng thực tế** | Phát hiện bất thường ảnh, nhiễu tín hiệu | Dự báo đơn giản, phân đoạn hành vi | ECG, cảm biến công nghiệp, phát hiện bất thường theo thời gian | Phát hiện cộng tác bất thường, mạng xã hội, đồ thị IoT |
| **Ưu điểm chính** | Xử lý tốt dữ liệu không gian | Dễ huấn luyện, đơn giản | Ghi nhớ lâu, học mối quan hệ phức tạp | Học biểu diễn từ cấu trúc kết nối |
| **Nhược điểm chính** | Không học được thứ tự | Mất thông tin nếu chuỗi dài | Tốn tài nguyên, cần tối ưu cẩn thận | Không áp dụng được cho dữ liệu phi đồ thị |

CNN rất mạnh trong trích xuất đặc trưng không gian từ ảnh hoặc heatmap nhưng không phù hợp lắm cho chuỗi thời gian dài vì thiếu khả năng mô hình hóa quan hệ thời gian.

Ngoài ra, số lượng tham số của CNN khá lớn (trên 15 triệu), dẫn đến chi phí tính toán cao, không phù hợp với bài toán thời gian thực.

RNN và LSTM là hai lựa chọn phổ biến cho dữ liệu chuỗi. Trong đó, LSTM khắc phục điểm yếu của RNN trong việc ghi nhớ dài hạn nhờ cơ chế cổng nhớ. Với chỉ khoảng 1–5 triệu tham số, LSTM cho hiệu quả tốt khi đầu vào là chuỗi keypoint trích xuất từ cảm biến, đồng thời đảm bảo nhẹ và dễ huấn luyện.

GCN (Graph Convolutional Network) đặc biệt hiệu quả khi xử lý dữ liệu xương người nhờ khả năng mô hình hóa mối quan hệ giữa các khớp dưới dạng đồ thị. Tuy nhiên, mô hình này thường có quy mô lớn (hơn 10 triệu tham số như ST-GCN) và đòi hỏi cấu hình cao, chưa phù hợp cho các ứng dụng chạy thực tế trên máy tính phổ thông.

Từ so sánh trên, có thể thấy việc lựa chọn LSTM có sự cân bằng giữa độ chính xác, chi phí tính toán trong đề tài.

## Giới thiệu về TSMAE

TSMAE, viết tắt của Time-Series Memory-Augmented Autoencoder, là một mô hình học sâu chuyên dùng để phát hiện bất thường trong dữ liệu chuỗi thời gian. Mô hình này kết hợp giữa ba yếu tố: mạng LSTM, autoencoder, và một bộ nhớ tăng cường. Mục tiêu của nó là học được cách biểu diễn và tái tạo lại dữ liệu bình thường một cách chính xác, để từ đó phát hiện ra các mẫu bất thường khi quá trình tái tạo gặp lỗi.

Mô hình tổng thế hệ thống sẽ gồm ba phần.

### Bộ mã hóa LSTM (Encoder)

Thành phần đầu tiên của TSMAE là bộ mã hóa, sử dụng mạng LSTM để xử lý dữ liệu chuỗi đầu vào. LSTM có khả năng ghi nhớ các quan hệ lâu dài trong chuỗi, vì vậy nó rất phù hợp để nén thông tin của toàn bộ chuỗi thành một biểu diễn ngắn gọn, còn gọi là vector mã hóa. Đây là bước giúp mô hình hiểu được "ý nghĩa" tổng quát của chuỗi dữ liệu.

### ****Bộ nhớ tăng cường (Memory Module)****

Sau khi dữ liệu được mã hóa, vector biểu diễn này sẽ được đưa vào một khối bộ nhớ. Khác với autoencoder thông thường, TSMAE có thêm một khối bộ nhớ chứa nhiều mẫu đại diện được học từ dữ liệu bình thường trong quá trình huấn luyện. Mô hình sẽ so sánh vector mã hóa của chuỗi đầu vào với các mẫu trong bộ nhớ, và từ đó trích xuất ra thông tin hữu ích nhất để bổ sung vào quá trình giải mã. Đây chính là bước giúp mô hình tăng khả năng khái quát và xử lý tốt hơn các biến thể trong dữ liệu.

### Bộ giải mã LSTM (Decoder)

Tiếp theo, thông tin tổng hợp từ encoder và bộ nhớ sẽ được đưa vào một mạng LSTM thứ hai, đóng vai trò là bộ giải mã. Mục tiêu của bộ giải mã là tái tạo lại toàn bộ chuỗi đầu vào từ thông tin đã được rút gọn và tăng cường ở các bước trước. Nếu chuỗi đầu vào là bình thường, quá trình tái tạo sẽ rất chính xác. Ngược lại, nếu chuỗi đầu vào là bất thường, mô hình sẽ khó tái tạo đúng, và từ đó sai số sẽ cao hơn.

## Kiến trúc của TSMAE

**Time** **Series Memory-Augmented Autoencoder (TSMAE)** là một khuôn khổ mô hình mới được thiết kế chuyên biệt cho bài toán phát hiện bất thường trong dữ liệu chuỗi thời gian. Mô hình này tích hợp một autoencoder dựa trên mạng LSTM với một **khối bộ nhớ phụ trợ**, cho phép học và khai thác hiệu quả các mẫu từ dữ liệu bình thường, đồng thời **hạn chế khả năng khái quát hóa đối với dữ liệu bất thường.**

**Kiến** trúc của TSMAE gồm ba thành phần chính:

1. **Bộ mã hóa LSTM (LSTM encoder),**
2. **Khối bộ nhớ (memory module),**
3. **Bộ giải mã LSTM (LSTM decoder).**

Ba thành phần này phối hợp với nhau để mô hình vừa ghi nhớ các mẫu chuẩn, vừa phản ứng rõ ràng khi gặp bất thường. Hình 5 minh họa cấu trúc tổng thể của mô hình TSMAE[4].

**A diagram of a computer hardware system

Description automatically generated**

Hình : Kiến trúc của TSMAE

### LSTM Encoder

**Bộ** **mã hóa LSTM (LSTM encoder)** là một thành phần quan trọng trong mô hình TSMAE, đóng vai trò xử lý dữ liệu chuỗi và học các quan hệ phụ thuộc theo thời gian. Nhiệm vụ của nó là chuyển đổi chuỗi dữ liệu đầu vào có dạng ma trận nhiều chiều (với chiều dài chuỗi là T và số đặc trưng mỗi bước là d) thành một vector đặc trưng nén (latent vector) có kích thước cố định.

Quá trình mã hóa này được thực hiện thông qua cơ chế hoạt động nội tại của các tế bào LSTM, cụ thể là thông qua **ba cổng chính** gồm:

* **Cổng quên (Forget gate):** quyết định những thông tin nào từ trạng thái bộ nhớ trước đó cần được loại bỏ;
* **Cổng đầu vào (Input gate):** quyết định thông tin mới nào sẽ được thêm vào trạng thái bộ nhớ hiện tại;
* **Cổng đầu ra (Output gate):** xác định phần thông tin nào sẽ được đưa vào trạng thái ẩn để truyền đi bước tiếp theo.

Mỗi tế bào LSTM duy trì hai trạng thái: **trạng thái ẩn** và **trạng thái bộ nhớ**. Nhờ sự tương tác giữa ba cổng và hai trạng thái này, LSTM có thể ghi nhớ các mẫu dữ liệu từ quá khứ và cập nhật liên tục theo thời gian. Quá trình xử lý này được minh họa trực quan trong Hình 7.

Trong đó, **cổng quên** là bước đầu tiên tại mỗi thời điểm, cho biết thông tin nào từ trạng thái bộ nhớ trước đó sẽ bị loại bỏ. Cổng này đóng vai trò quan trọng trong việc lọc bỏ thông tin không còn cần thiết để tránh tình trạng quá tải bộ nhớ và đảm bảo mạng chỉ giữ lại các tín hiệu quan trọng theo thời gian.

ft = σ(Wf \* ht-1 + bf)

A diagram of a flowchart

Description automatically generated

Hình : Kiến trúc của TSMAE

Kết quả phụ thuộc vào hàm kích hoạt sigmoid, trong đó:

* Xt là đầu vào hiện tại,
* ht​ là trạng thái ẩn trước đó,

Wb và b​f là các trọng số và độ lệch được học trong quá trình huấn luyện.

**Cổng đầu vào (Input Gate):**

Bước này xác định thông tin mới nào sẽ được lưu vào trạng thái bộ nhớ hiện tại.

Quá trình này gồm hai phần:

1. Một cổng sigmoid xác định phần nào của thông tin mới sẽ được cập nhật.
2. Một lớp kích hoạt tanh tạo ra vector trạng thái ứng viên, chứa thông tin mới có thể thêm vào bộ nhớ.

It = σ(Wi · [ht-1, xt] + bi)

c˜t = tanh(Wc · [ht-1, xt] + bc)

* **Cập nhật trạng thái bộ nhớ (Cell State Update):** Quá trình này kết hợp giữa thông tin được giữ lại từ cổng quên và thông tin mới từ cổng đầu vào để cập nhật trạng thái bộ nhớ hiện tại.

Công thức được tính như sau:

ct=ft ⊙ c˜t + it ⊙ ct

Trong đó:

* ct: trạng thái bộ nhớ tại thời điểm hiện tại.
* ft: đầu ra từ cổng quên, xác định phần nào của bộ nhớ cũ ct−1c\_{t-1}ct−1​ sẽ được giữ lại.
* i​t: đầu ra từ cổng đầu vào, xác định phần nào của trạng thái ứng viên c~t\tilde{c}\_tc~t​ sẽ được thêm vào.
* ⊙: phép nhân từng phần tử (element-wise).

**Cổng đầu ra (Output Gate):** Cổng này quyết định trạng thái ẩn hth\_tht​ sẽ được xuất ra tại bước thời gian hiện tại, dựa trên trạng thái bộ nhớ đã được cập nhật.

Công thức được tính như sau:

ot=σ(Wb ⋅ [ht ,xt] + bo)

ht=ot⊙tanh(ct)

Trong đó:

* o​t: đầu ra của cổng đầu ra (output gate), được tính từ đầu vào hiện tại và trạng thái ẩn của bước trước.
* ct:trạng thái bộ nhớ hiện tại.
* ht​: trạng thái ẩn (hidden state) tại thời điểm ttt, là kết quả đầu ra của tế bào LSTM.
* σ: hàm kích hoạt sigmoid.
* tanh: hàm kích hoạt hyperbolic tangent.
* ⊙: phép nhân từng phần tử (element-wise).

### Memory module

Khối bộ nhớ trong TSMAE có nhiệm vụ **tăng cường biểu diễn ẩn** (latent representation) bằng cách liên kết nó với các mẫu bộ nhớ đã được học từ trước. Các **mục bộ nhớ** này đại diện cho các mẫu điển hình (prototypical patterns) của dữ liệu bình thường.

*1. Truy xuất bộ nhớ thông qua vector địa chỉ*

Mô-đun bộ nhớ bắt đầu bằng việc tính toán **vector địa chỉ** – là một tập các hệ số thể hiện **mức độ tương đồng** giữa vector biểu diễn đầu vào z và từng memory item trong bộ nhớ mi​. Việc này được thực hiện bằng hàm softmax như sau:

A mathematical equation with black text

Description automatically generated with medium confidence

Trong đó:

* qi là độ tương đồng giữa biểu diễn ẩn zzz và mục bộ nhớ thứ iii;
* N là tổng số mục trong bộ nhớ
* Hàm số mũ kết hợp với softmax giúp chuyển độ tương đồng thành xác suất, đảm bảo các giá trị q nằm trong khoảng [0,1] và tổng bằng 1.

*2. Tái tạo biểu diễn bằng tổ hợp tuyến tính*

Sau khi có vector địa chỉ q, mô hình sẽ tái tạo lại biểu diễn ẩn z^ bằng **trung bình có trọng số** của các mục trong bộ nhớ:

A mathematical equation with numbers and symbols

Description automatically generated

Nói cách khác, mô hình lấy một phần của mỗi mẫu bộ nhớ, tỷ lệ theo mức độ giống với đầu vào, và kết hợp chúng lại để tạo ra một phiên bản “chuẩn hóa” của đầu vào.

*3. Sparsification – làm thưa vector truy xuất*

Để tránh việc mô hình dùng quá nhiều mục bộ nhớ để "lắp ghép" lại cả dữ liệu bất thường, một bước **làm thưa (sparsification)** được thêm vào. Cụ thể, vector địa chỉ qqq sẽ được sửa đổi như sau:

A black text on a white background

Description automatically generated

Trong đó λ là một ngưỡng làm thưa (sparsification threshold);

Nếu qi​ nhỏ hơn λ nó sẽ bị loại bỏ (coi như 0). Điều này giúp mô hình chỉ sử dụng **một vài mục bộ nhớ thực sự liên quan**, thay vì "mượn" thông tin từ nhiều mẫu một cách quá phức tạp.

*4. Chuẩn hóa lại vector truy xuất*

Sau khi làm thưa, vector q được **chuẩn hóa** để đảm bảo tổng các phần tử vẫn bằng 1, tránh lỗi chia cho 0:

A black text on a white background

Description automatically generated

Trong đó ϵ là một hằng số rất nhỏ để tránh chia cho 0 trong trường hợp tất cả các qiq\_iqi​ đều bị triệt tiêu.

*5. Xác định ngưỡng làm thưa λ*

Giá trị λ được xác định dựa trên số lượng mục bộ nhớ NNN, sao cho **trung bình mỗi mục có xác suất được chọn là 1/N**. Điều kiện được đề xuất như sau:

A number and a symbol

Description automatically generated with medium confidence

Việc này giúp kiểm soát số lượng mẫu được mô hình tham chiếu mỗi lần, giữ cho mô hình học sâu và chọn lọc hơn.

### LSTM Decoder

**Bộ** **giải mã LSTM (LSTM Decoder)** là thành phần thứ ba trong kiến trúc TSMAE, đóng vai trò khôi phục lại chuỗi dữ liệu gốc từ biểu diễn ẩn đã được mã hóa và tăng cường thông qua khối bộ nhớ. Trong kiến trúc autoencoder, decoder là phần "đối xứng" với encoder – nếu encoder thực hiện việc nén dữ liệu đầu vào, thì decoder làm nhiệm vụ tái tạo lại toàn bộ chuỗi đầu vào càng chính xác càng tốt.

* **Đầu vào của decoder:**

Decoder trong TSMAE không hoạt động trực tiếp trên biểu diễn ẩn z mà sử dụng phiên bản đã được hiệu chỉnh thông qua bộ nhớ – gọi là z^ Đây là biểu diễn đã được tăng cường bởi các mẫu dữ liệu bình thường lưu trong khối bộ nhớ, giúp mô hình có thêm ngữ cảnh trong quá trình tái tạo.

* **Kiến trúc và cơ chế hoạt động:**

Giống như encoder, decoder sử dụng mạng **LSTM nhiều tầng** để tái tạo lại chuỗi thời gian. Mỗi bước của decoder sẽ nhận một phần của thông tin từ z^\hat{z}z^ và trạng thái ẩn từ bước trước để tạo ra một bước trong chuỗi tái tạo.

Trong giai đoạn huấn luyện, kỹ thuật **teacher forcing** thường được sử dụng – tức là đầu vào cho mỗi bước decoder là giá trị thật tại bước trước đó (thay vì đầu ra dự đoán), giúp mô hình học nhanh và chính xác hơn.

* **Đầu ra và lỗi tái tạo:**

Kết quả cuối cùng của decoder là chuỗi tái tạo X^ = [x1,x2,...,xT]. Chuỗi này sau đó được so sánh với chuỗi đầu vào gốc để tính toán **mức sai số tái tạo (reconstruction error)** – là cơ sở chính để phát hiện bất thường.

Mô hình TSMAE không cố gắng tái tạo tốt mọi chuỗi, mà chỉ học tái tạo chính xác **các chuỗi bình thường**. Do đó, nếu một chuỗi bất thường được đưa vào, mô hình sẽ không thể khôi phục lại chính xác, và sai số tái tạo sẽ tăng cao – đây chính là tín hiệu để nhận diện dị thường

* **Vai trò của decoder trong phát hiện bất thường:**

Decoder giúp mô hình kiểm tra xem biểu diễn đã mã hóa có đủ thông tin để tái tạo dữ liệu ban đầu hay không. Khi mô hình hoạt động đúng, chỉ những biểu diễn của chuỗi bình thường mới được tái tạo tốt nhờ tham chiếu bộ nhớ. Các chuỗi bất thường thường tạo ra sai số lớn, do không khớp với bất kỳ mẫu bộ nhớ nào.

### Loss Function

Hàm mất mát (loss function) của TSMAE bao gồm **hai thành phần chính**:

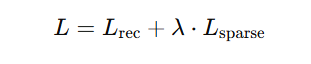
* **Mất mát tái tạo (Reconstruction Loss – LrecL\_{\text{rec}}Lrec**​**)**

Thành phần này đo lường độ chênh lệch giữa chuỗi đầu vào gốc và chuỗi được mô hình tái tạo lại. Cụ thể, mô hình sử dụng sai số bình phương trung bình (MSE – Mean Squared Error) để tối thiểu hóa sự khác biệt này, giúp mô hình học cách tái tạo chính xác dữ liệu bình thường.

* **Mất mát thưa hóa (Sparsity Loss – LsparseL\_{\text{sparse}}Lsparse**​**)**

Thành phần này khuyến khích **vector truy xuất bộ nhớ** (addressing vector) trở nên thưa hơn – nghĩa là chỉ tập trung vào một số mẫu bộ nhớ quan trọng nhất. Điều này giúp mô hình không sử dụng quá nhiều mục bộ nhớ để tái tạo dữ liệu bất thường, từ đó tăng độ nhạy trong phát hiện dị thường.

Hàm mất mát tổng thể được biểu diễn như sau:



Trong đó: λ là một **siêu tham số (hyperparameter)**, đóng vai trò cân bằng giữa độ chính xác của quá trình tái tạo và khả năng phát hiện bất thường thông qua độ thưa của vector truy xuất.

## Triển khai TSMAE bằng python

### Mô hình TSMAE

Mô hình TSMAE được xây dựng trên nền tảng **PyTorch** do đây là một thư viện học sâu mã nguồn mở mạnh mẽ, linh hoạt và dễ triển khai, đặc biệt phù hợp cho các bài toán xử lý chuỗi thời gian và mô hình mạng tuần tự như LSTM. PyTorch cung cấp các mô-đun tích hợp sẵn như nn.LSTM, nn.Module, và các hàm tối ưu hóa, giúp giảm đáng kể độ phức tạp trong việc xây dựng mô hình từ đầu. Ngoài ra, PyTorch hỗ trợ huấn luyện trên GPU, có khả năng tự động lan truyền đạo hàm (autograd) và quản lý bộ nhớ hiệu quả, từ đó đẩy nhanh quá trình huấn luyện và thử nghiệm. Với cú pháp gần gũi với Python thuần và khả năng dễ dàng gỡ lỗi, PyTorch là lựa chọn lý tưởng cho việc phát triển, kiểm thử và tối ưu mô hình TSMAE một cách nhanh chóng và đáng tin cậy3.3.1.Training setting.

Trong quá trình xây dựng và huấn luyện mô hình TSMAE, các siêu tham số (hyperparameters) được lựa chọn dựa trên tính chất của dữ liệu ECG5000 và mục tiêu phát hiện bất thường. Các giá trị này được điều chỉnh để đảm bảo mô hình đạt hiệu quả cao nhất về khả năng tái tạo và phân biệt dị thường.

* **Tham số mạng**
  + **input\_size = 1:** Mỗi bước thời gian chỉ có một đặc trưng (biên độ tín hiệu ECG), nên đầu vào mỗi time step là một giá trị thực.
  + **hidden\_size = 10:** Đây là số lượng neuron trong tầng ẩn (latent vector). Giá trị này được chọn đủ lớn để học đặc trưng phức tạp, nhưng đủ nhỏ để tránh quá khớp dữ liệu nhiễu.
  + **memory\_size = 20:** Số lượng mẫu bộ nhớ – đại diện cho các hình mẫu dữ liệu bình thường. Con số này giúp mô hình có khả năng ghi nhớ đa dạng các dạng sóng bình thường mà không làm tăng độ phức tạp quá mức.
* **Tham số sparsity và regularization**
  + **sparsity\_threshold = 0.05:** Ngưỡng thưa hóa dùng để loại bỏ các thành phần không quan trọng trong vector truy xuất bộ nhớ, giúp tăng độ phân biệt giữa dữ liệu thường và bất thường.
  + **sparsity\_factor = 0.001:** Hệ số điều chỉnh mức độ ảnh hưởng của mất mát sparsity trong hàm mất mát tổng. Giá trị này nhỏ để không làm lu mờ mục tiêu chính là tái tạo.
* **Tham số huấn luyện**
  + **batch\_size = 1:** Kích thước batch nhỏ giúp mô hình cập nhật thường xuyên hơn, phù hợp với dữ liệu không quá lớn như ECG5000 và kết quả sẽ chính xác hơn
  + **num\_epochs = 50:** Số epoch đủ để mô hình học ổn định nhưng vẫn tránh overfitting.
  + **learning\_rate = 1e-3:** Tốc độ học tiêu chuẩn cho mô hình dựa trên Adam. Có thể tinh chỉnh tùy theo tốc độ hội tụ.

### Data Preprocessing

Dữ liệu sử dụng trong đề tài là **ECG5000**, một tập dữ liệu phổ biến trong phát hiện bất thường sinh lý, đặc biệt là tín hiệu điện tim (ECG). Mục tiêu của bước tiền xử lý là **tách dữ liệu theo loại (bình thường / bất thường), định dạng lại dữ liệu thành tensor phù hợp**, và phục vụ trực tiếp cho quá trình huấn luyện mô hình TSMAE.

Toàn bộ quy trình được đóng gói trong lớp ECG5000, kế thừa từ torch.utils.data.Dataset, cho phép tích hợp trực tiếp với DataLoader của PyTorch.

#### Code cho phần data pre processing

class ECG5000(Dataset):

    def \_\_init\_\_(self, mode, split='train'):

        """

        mode: 'normal', 'anomaly', or 'all'.

              'all' means do not filter any samples (both normal and anomaly).

        split: 'train' to load training data; 'test' to load test data.

        """

        assert mode in ['normal', 'anomaly', 'all']

        assert split in ['train', 'test']

        # Select the file based on the split.

        if split == 'train':

            file\_path = 'ECG5000\_TRAIN.arff'

        else:

            file\_path = 'ECG5000\_TEST.arff'

        data, meta = arff.loadarff(file\_path)

        df = pd.DataFrame(data, columns=meta.names())

        # Rename the label column.

        new\_columns = list(df.columns)

        new\_columns[-1] = 'target'

        df.columns = new\_columns

        # Filter samples based on mode.

        if mode == 'normal':

            df = df[df.target == b'1'].drop(labels='target', axis=1)

        elif mode == 'anomaly':

            df = df[df.target != b'1'].drop(labels='target', axis=1)

        else:  # mode == 'all'

            df = df.drop(labels='target', axis=1)

        # Convert DataFrame to a numpy array of type float32.

        self.X = df.astype(np.float32).to\_numpy()

    def \_\_getitem\_\_(self, index):

        # Each sample is reshaped as (sequence\_length, 1)

        sample = torch.from\_numpy(self.X[index]).unsqueeze(-1)

        return sample

    def \_\_len\_\_(self):

        return self.X.shape[0]

    def get\_torch\_tensor(self):

        return torch.from\_numpy(self.X)

#### Giải thích code và tầm quan trọng của Data Preprocessing

Đoạn code định nghĩa lớp ECG5000 đóng vai trò là một module tiền xử lý dữ liệu đầu vào, được thiết kế để tích hợp trực tiếp với hệ thống huấn luyện mô hình trong PyTorch. Cụ thể, lớp này cho phép tải dữ liệu từ tập ECG5000 dưới định dạng .arff và tự động lựa chọn giữa tập huấn luyện (train) hoặc tập kiểm tra (test). Dựa trên chế độ được chỉ định (normal, anomaly, hoặc all), lớp sẽ lọc các mẫu tương ứng: chỉ giữ lại mẫu bình thường, mẫu bất thường, hoặc toàn bộ tập. Sau đó, cột nhãn (target) bị loại bỏ để đảm bảo chỉ còn lại phần dữ liệu đặc trưng đầu vào, và dữ liệu được chuyển đổi sang kiểu float32 để tăng hiệu quả tính toán. Toàn bộ tập dữ liệu được lưu dưới dạng numpy array và cung cấp lại dưới dạng tensor thông qua các phương thức \_\_getitem\_\_ và \_\_len\_\_, cho phép truy xuất từng chuỗi riêng lẻ dưới định dạng (sequence\_length, 1). Với cấu trúc này, lớp ECG5000 giúp đơn giản hóa quy trình trích xuất và chuẩn bị chuỗi thời gian cho mô hình TSMAE, đồng thời hỗ trợ các bước xử lý song song và tối ưu bộ nhớ trong quá trình huấn luyện.

### Memory Module implementation

**Khối bộ nhớ (Memory Module)** trong kiến trúc TSMAE đóng vai trò then chốt trong việc **lưu trữ các mẫu đại diện đặc trưng của dữ liệu chuỗi thời gian**. Thành phần này sử dụng một **vector truy xuất (addressing vector)** để tương tác với các mục trong bộ nhớ, từ đó tạo ra **biểu diễn ẩn tái hợp** – dùng để so sánh và phát hiện bất thường. Dưới đây là đoạn mã Python cài đặt khối bộ nhớ trong mô hình.

#### Code cho Memory Module

class MemoryModule(nn.Module):

    def \_\_init\_\_(self, memory\_size, hidden\_size, sparsity\_threshold=0.05):

        """

        memory\_size: Number of memory items.

        hidden\_size: Dimensionality of each memory item.

        sparsity\_threshold: Threshold for rectifying the addressing vector.

        """

        super(MemoryModule, self).\_\_init\_\_()

        self.memory\_size = memory\_size

        self.hidden\_size = hidden\_size

        self.sparsity\_threshold = sparsity\_threshold

        # Initialize learnable memory items.

        self.memory = nn.Parameter(torch.randn(memory\_size, hidden\_size))

    def forward(self, z):

        """

        z: latent representation from encoder with shape (batch, hidden\_size)

        Returns:

          z\_hat: recombined latent representation from memory.

          q: sparse addressing vector with shape (batch, memory\_size)

        """

        # Compute similarity scores between latent vector and memory items.

        sim = torch.matmul(z, self.memory.t())  # shape: (batch, memory\_size)

        # Softmax to obtain addressing weights.

        q = nn.functional.softmax(sim, dim=1)

        # Rectify: subtract threshold and zero out negatives.

        q = torch.max(q - self.sparsity\_threshold, torch.zeros\_like(q))

        # Normalize so that each row sums to 1.

        q = q / (q.sum(dim=1, keepdim=True) + 1e-8)

        # Recombine memory items.

        z\_hat = torch.matmul(q, self.memory)

        return z\_hat, q

#### Giải thích code và tầm quan trọng của Memory Module

Lớp MemoryModule kế thừa từ nn.Module trong PyTorch, được thiết kế để đóng vai trò như **bộ nhớ tăng cường** trong mô hình TSMAE, với mục tiêu hỗ trợ tái tạo biểu diễn ẩn của dữ liệu đầu vào một cách chọn lọc và có kiểm soát. Khi nhận một vector biểu diễn ẩn từ encoder, lớp này tính toán **độ tương đồng (similarity)** giữa vector đó và các mục bộ nhớ đã được học, thông qua phép nhân ma trận. Kết quả độ tương đồng được chuẩn hóa bằng hàm softmax để tạo ra **vector truy xuất bộ nhớ** (addressing vector) cho mỗi mẫu. Sau đó, vector này được **làm thưa (sparsify)** bằng cách trừ đi một ngưỡng định trước và loại bỏ các giá trị âm nhằm giới hạn số lượng mục bộ nhớ được sử dụng. Tiếp theo, nó được **chuẩn hóa** lại để đảm bảo tổng các giá trị bằng 1. Cuối cùng, biểu diễn ẩn mới z^ được tái hợp từ các mục bộ nhớ với trọng số là vector truy xuất, giúp mô hình tập trung vào các mẫu dữ liệu bình thường đặc trưng, từ đó tăng cường khả năng phát hiện các trường hợp bất thường qua sai số tái tạo.

### TSMAE implementation

Trong mô hình TSMAE, thành phần encoder và decoder đều được xây dựng dựa trên mô-đun nn.LSTM – một lớp tích hợp sẵn trong thư viện PyTorch để triển khai mạng LSTM (Long Short-Term Memory). Việc sử dụng nn.LSTM giúp đơn giản hóa quá trình xây dựng mô hình tuần tự, vì lớp này đã tối ưu sẵn các thao tác xử lý trạng thái ẩn, cổng vào/ra, lan truyền thời gian, cũng như hỗ trợ chế độ huấn luyện theo batch với cấu trúc đầu vào chuẩn (batch\_size, sequence\_length, input\_size). Ngoài ra, nn.LSTM còn hỗ trợ nhiều tầng (multi-layer), trạng thái hai chiều (bidirectional), và tùy chọn dropout, cho phép mở rộng kiến trúc dễ dàng khi cần nâng cao hiệu suất mô hình. Việc tận dụng lớp nn.LSTM có sẵn không chỉ tăng tính ổn định và hiệu quả, mà còn đảm bảo khả năng tương thích tốt với hệ sinh thái huấn luyện của PyTorch.

#### Code cho TSMAE

class TSMAE(nn.Module):

    def \_\_init\_\_(self, input\_size, hidden\_size, memory\_size, sparsity\_threshold=0.05, sparsity\_factor=0.001):

        """

        input\_size: Dimension of each time step (e.g., 1)

        hidden\_size: Dimension of the latent representation

        memory\_size: Number of memory items.

        sparsity\_threshold: Threshold used in the memory module.

        sparsity\_factor: Weight for the sparsity penalty in the loss.

        """

        super(TSMAE, self).\_\_init\_\_()

        self.input\_size = input\_size

        self.hidden\_size = hidden\_size

        self.memory\_size = memory\_size

        self.sparsity\_factor = sparsity\_factor

        # LSTM Encoder: encodes input sequence into a latent vector.

        self.encoder = nn.LSTM(input\_size, hidden\_size, batch\_first=True)

        # Memory Module: extracts typical normal patterns.

        self.memory\_module = MemoryModule(memory\_size, hidden\_size, sparsity\_threshold)

        # LSTM Decoder: decodes the latent representation back to sequence.

        self.decoder = nn.LSTM(hidden\_size, hidden\_size, batch\_first=True)

        # Final layer to project the LSTM decoder output to the input space.

        self.output\_layer = nn.Linear(hidden\_size, input\_size)

    def forward(self, x):

        """

        x: Input tensor of shape (batch, seq\_len, input\_size)

        Returns:

          x\_recon: Reconstructed sequence of shape (batch, seq\_len, input\_size)

          q: Sparse addressing vector from the memory module (batch, memory\_size)

          z: Latent representation from the encoder (batch, hidden\_size)

          z\_hat: Recombined latent representation from the memory module (batch, hidden\_size)

        """

        batch\_size, seq\_len, \_ = x.size()

        # Encode input sequence.

        enc\_out, (h\_n, c\_n) = self.encoder(x)

        z = h\_n[-1]  # Use the final hidden state; shape: (batch, hidden\_size)

        # Pass through memory module.

        z\_hat, q = self.memory\_module(z)

        # For decoding, repeat z\_hat across the sequence length.

        z\_hat\_seq = z\_hat.unsqueeze(1).repeat(1, seq\_len, 1)

        dec\_out, \_ = self.decoder(z\_hat\_seq)

        # Project decoder output back to input dimension.

        x\_recon = self.output\_layer(dec\_out)

        return x\_recon, q, z, z\_hat

    def loss\_function(self, x, x\_recon, q):

        # Mean Squared Error reconstruction loss.

        rec\_loss = torch.mean((x - x\_recon)\*\*2)

        # Sparsity loss to encourage a sparse addressing vector.

        sparsity\_loss = torch.mean(torch.log(1 + q\*\*2))

        loss = rec\_loss + self.sparsity\_factor \* sparsity\_loss

        return loss, rec\_loss, sparsity\_loss

#### Giải thích code về TSMAE và hàm loss

Mô hình **TSMAE** bao gồm ba thành phần chính: **encoder**, **bộ nhớ (memory module)** và **decoder**. Trong đó, **encoder** có nhiệm vụ xử lý chuỗi đầu vào và mã hóa nó thành một vector biểu diễn ẩn z, đại diện cho các đặc trưng tuần tự (temporal patterns) của dữ liệu. Khối bộ nhớ thực hiện tiếp việc lan truyền tiến như dã mô tả ở trên là lưu trữ các mẫu điển hình của dữ liệu bình thường, cho phép mô hình so sánh các đầu vào mới với những mẫu đã học trước đó. Bộ nhớ tính toán một **vector truy xuất q** dựa trên độ tương đồng giữa z và các mục bộ nhớ, sau đó áp dụng một **ngưỡng thưa hóa λ** để điều chỉnh qqq, đảm bảo chỉ một phần nhỏ các mục bộ nhớ được kích hoạt mạnh và tham gia đáng kể vào quá trình tái tạo.

Tiếp theo, **decoder** sẽ tái tạo lại chuỗi đầu vào X^ từ biểu diễn ẩn đã tái hợp z^ bằng cách kết hợp các mục bộ nhớ có trọng số là vector q. Quá trình này đảm bảo rằng chuỗi tái tạo vẫn giữ được các đặc trưng tuần tự quan trọng từ dữ liệu gốc.

Mô hình TSMAE được tối ưu hóa thông qua **hàm mất mát tùy chỉnh**, kết hợp giữa **mất mát tái tạo** và **mất mát sparsity**, nhằm nâng cao hiệu quả phát hiện bất thường. Cụ thể:

* **Mất mát tái tạo (Reconstruction Loss)** đo lường sai số giữa chuỗi đầu vào gốc X và chuỗi tái tạo X^ giúp mô hình học tái tạo chính xác. Hàm mất mát này được định nghĩa như sau:

A math equation with numbers and equal lines

Description automatically generated with medium confidence

* **Mất mát thưa hóa (Sparsity Loss)** nhằm khuyến khích vector truy xuất qqq trở nên thưa (chỉ có một số thành phần khác 0), từ đó tăng khả năng diễn giải và hiệu quả hoạt động của bộ nhớ. Công thức tính như sau:

A black symbol with black text

Description automatically generated

* **Hàm mất mát tổng** là tổ hợp tuyến tính giữa hai thành phần trên, với hệ số η\etaη điều chỉnh mức độ cân bằng giữa độ chính xác và độ thưa:

A black text on a white background

Description automatically generated

Nhờ vào hàm mất mát tổng quát này, TSMAE không chỉ học được các mẫu dữ liệu bình thường một cách hiệu quả, mà còn duy trì một không gian bộ nhớ tập trung và chọn lọc, từ đó trở nên đặc biệt phù hợp cho các bài toán phát hiện bất thường – nơi mà bất kỳ sự sai lệch nào so với các mẫu đã học đều có thể chỉ ra hành vi bất thường.

#### Kết quả sau khi training

A number and numbers on a white background

Description automatically generated

Hình : Traning Loss

A comparison of graphs with blue and orange lines

Description automatically generated

Hình : kết quả sau khi tái tạo cho dữ liệu bình thường và dữ liệu không bình thường

Tổng kết chương

Trong chương này, chúng ta đã trình bày chi tiết kiến trúc và cơ chế hoạt động của mô hình **TSMAE (Time-Series Memory-Augmented Autoencoder)** – một giải pháp hiệu quả cho bài toán phát hiện bất thường trong dữ liệu chuỗi thời gian.

TSMAE bao gồm ba thành phần chính: **encoder LSTM**, **khối bộ nhớ tăng cường**, và **decoder LSTM**. Encoder có nhiệm vụ trích xuất đặc trưng tuần tự từ dữ liệu đầu vào, trong khi khối bộ nhớ giúp mô hình so sánh biểu diễn ẩn với các mẫu điển hình đã học từ dữ liệu bình thường. Việc áp dụng cơ chế **truy xuất thưa hóa (sparse addressing)** giúp mô hình chỉ kích hoạt một số lượng nhỏ các mẫu bộ nhớ liên quan nhất, từ đó tăng khả năng phát hiện sai lệch. Cuối cùng, decoder sử dụng biểu diễn đã tái hợp từ bộ nhớ để tái tạo lại chuỗi đầu vào, và sai số tái tạo chính là tín hiệu chính để nhận biết bất thường.

Bên cạnh kiến trúc, chương này cũng mô tả quá trình hiện thực hóa mô hình bằng Python thông qua thư viện PyTorch, bao gồm xử lý dữ liệu đầu vào, cài đặt encoder–memory–decoder, và định nghĩa hàm mất mát tùy chỉnh kết hợp giữa **reconstruction loss** và **sparsity loss.**

Tổng thể, TSMAE không chỉ khai thác được sức mạnh của LSTM trong học đặc trưng theo thời gian mà còn tăng cường khả năng khái quát và phân tách thông qua cơ chế bộ nhớ có kiểm soát, giúp mô hình trở thành công cụ mạnh mẽ và đáng tin cậy trong các bài toán phát hiện bất thường trên dữ liệu tuần tự. Và đây là tiền đề để ta có thể lấy và phân tách mô hình để có thể triển khai trên phần cứng ở chương sau.

# XÂY DỰNG MÔ HÌNH TSMAE TRÊN PHẦN CỨNG

## Tách trọng số từ mô hình bằng Python

Việc xử lý và lấy trọng số từ mô hình chạy bằng python rất quan trọng trong việc xây dựng mô hình trên phần cứng điều đó giúp đảm bảo tính tiết kiệm thời gian cũng như là giúp cho việc thiết kế trở nên đơn giản hơn đồng thời làm cho việc triển khai mô hình trên phần cứng có kết quả chạy sát thực nhất so với trên phần mềm.

Tách trọng số từ mô hình bằng PyTorch:



Đây là cách để lấy toàn bộ trọng số của mô hình khi triển khai trên phần mềm bằng PyTorch tuy nhiên trong file tsmae\_weights.pth thì các trọng số mới chỉ được lưu dưới định dạng mà chỉ có PyTorch hiểu được chưa có sự phân tách ra thành từng neural hay là phân chia thành từng cổng để có thể triển khai thành phần cứng điều này đòi hỏi phải có một chương trình phân tách và trích xuất file trọng số này thành từng cổng từng neural riêng lẻ .

Xử lý file tsmae\_weights.pth:

import torch

import numpy as np

import os

# === THÔNG SỐ ===

INPUT\_FILE = 'tsmae\_weights.pth'

OUTPUT\_DIR = 'tsmae\_weights\_neurons'

HIDDEN\_SIZE = 10

GATES = ['input', 'forget', 'cell', 'output']

os.makedirs(OUTPUT\_DIR, exist\_ok=True)

# === Load mô hình

state\_dict = torch.load(INPUT\_FILE, map\_location='cpu')

# === Tách tensor LSTM đúng chiều và thứ tự

def split\_lstm\_tensor(param, base\_name):

arr = param.detach().cpu().numpy()

if arr.ndim == 2: # weight\_ih\_l0 hoặc weight\_hh\_l0: [4\*H, I]

for g, gate in enumerate(GATES):

gate\_matrix = arr[g \* HIDDEN\_SIZE : (g + 1) \* HIDDEN\_SIZE]

for i, row in enumerate(gate\_matrix):

fname = f"{base\_name}\_\_{gate}\_\_neuron\_{i}.txt"

np.savetxt(os.path.join(OUTPUT\_DIR, fname), row.reshape(1, -1), fmt="%.6f")

elif arr.ndim == 1: # bias\_ih\_l0 hoặc bias\_hh\_l0: [4\*H]

for g, gate in enumerate(GATES):

gate\_vector = arr[g \* HIDDEN\_SIZE : (g + 1) \* HIDDEN\_SIZE]

fname = f"{base\_name}\_\_{gate}.txt"

np.savetxt(os.path.join(OUTPUT\_DIR, fname), gate\_vector.reshape(1, -1), fmt="%.6f")

# === Tách tensor thường

def split\_general\_tensor(param, base\_name):

arr = param.detach().cpu().numpy()

if arr.ndim == 2:

for i, row in enumerate(arr):

fname = f"{base\_name}\_\_row\_{i}.txt"

np.savetxt(os.path.join(OUTPUT\_DIR, fname), row.reshape(1, -1), fmt="%.6f")

elif arr.ndim == 1:

fname = f"{base\_name}.txt"

np.savetxt(os.path.join(OUTPUT\_DIR, fname), arr.reshape(1, -1), fmt="%.6f")

else:

fname = f"{base\_name}.npy"

np.save(os.path.join(OUTPUT\_DIR, fname), arr)

# === Duyệt qua state\_dict

for name, param in state\_dict.items():

clean\_name = name.replace(".", "\_\_")

if 'weight\_ih\_l0' in name or 'weight\_hh\_l0' in name or 'bias\_ih\_l0' in name or 'bias\_hh\_l0' in name:

split\_lstm\_tensor(param, clean\_name)

else:

split\_general\_tensor(param, clean\_name)

Đoạn mã được thiết kế nhằm **trích xuất và lưu riêng từng trọng số** của mô hình TSMAE đã huấn luyện, đặc biệt là các tham số liên quan đến **LSTM layer** trong encoder hoặc decoder. Mục tiêu là **phân tách từng neuron** trong các cổng của LSTM (gồm: input gate, forget gate, cell gate, output gate), giúp người dùng dễ dàng phân tích hoặc sử dụng lại trọng số trong các môi trường khác như SystemC hoặc mô phỏng phần cứng.

Cụ thể, mã sẽ:

* Tải mô hình TSMAE đã lưu ở định dạng .pth,
* Duyệt toàn bộ state\_dict để kiểm tra tên và loại tham số,
* Nếu là các tensor của LSTM (weight\_ih\_l0, weight\_hh\_l0, bias\_ih\_l0, bias\_hh\_l0), hệ thống sẽ **tách riêng từng cổng LSTM** theo đúng thứ tự PyTorch: input → forget → cell → output, và lưu từng **neuron của mỗi cổng** thành file .txt,
* Nếu là tensor thường (ví dụ trọng số của decoder hoặc các lớp tuyến tính), chúng sẽ được lưu thành từng hàng hoặc vector riêng biệt,
* Tất cả kết quả được lưu có cấu trúc tên rõ ràng trong thư mục tsmae\_weights\_neurons/.

Việc trích xuất này không chỉ giúp dễ dàng kiểm thử mô hình trên phần cứng hoặc mã SystemC mà còn tạo điều kiện cho việc kiểm tra từng thành phần của mô hình theo cách trực quan và chi tiết hơn. Đây là một bước chuẩn bị quan trọng để tiến hành **mô phỏng phần cứng hóa mô hình học sâu**

## Kiến trúc của TSMAE khi dùng systemC

**A diagram of a computer

Description automatically generated**

Hình : Kiến trúc của TSMAE dùng systemC

Sơ đồ trên mô tả kiến trúc phần cứng của mô hình TSMAE khi triển khai theo pipeline đồng bộ hóa tín hiệu, gồm ba khối chính: LSTM Encoder, Memory Module và LSTM Decoder, được liên kết thông qua tín hiệu handshake để đảm bảo đồng bộ dữ liệu theo từng bước thời gian. Đầu vào X được xử lý bởi encoder để tạo ra biểu diễn ẩn last\_ht, sau đó được truyền sang Memory Module để tái hợp thành biểu diễn tăng cường Latent. Khối trung gian DRIVER lặp lại biểu diễn này thành chuỗi Latent\_series và truyền cho decoder để tái tạo dần chuỗi đầu ra. Output Layer tiếp nhận chuỗi trạng thái từ decoder và sinh ra X\_RECON, là chuỗi đã tái cấu trúc. Kiến trúc này giúp mô phỏng chính xác hoạt động của TSMAE trong môi trường phần cứng, đảm bảo tính tuần tự, đồng bộ và dễ triển khai trong các hệ thống logic như SystemC hoặc FPGA.

## LSTM\_neural

### Module LSTM\_neural

A white square with black text and blue dots

Description automatically generated

Hình : Module LSTM\_neural

Chức năng của các chân IO trong module:

Bảng : Chức năng các chân vào ra của LSTM\_neural

|  |  |  |
| --- | --- | --- |
| **Port Name** | **Type** | **Description** |
| Clk | input bool | Nhận xung đồng hồ |
| Rst | input bool | Reset các thanh ghi bên trong |
| X | input float | Dữ liệu vào theo dạng sequence |
| Ht[10] | output float | Giá trị của hidden theo dạng parallel |
| Ct[10] | output float | Giá trị của cell state theo dạng parallel |
| rdy\_to\_receiv | input bool | Tín hiệu handshake vào cho việc nhận |
| is\_rdy\_to\_receiv | output bool | Tín hiệu handshake ra cho việc nhận |
| rdy\_to\_send | input bool | Tín hiệu handshake vào cho việc truyền |
| is\_rdy\_to\_send | output bool | Tín hiệu handshake ra cho việc truyền |
| Rdy\_to\_output | input bool | Tín hiệu handshake vào cho việc tính output |
| is\_rdy\_to\_output | output bool | Tín hiệu handshake ra cho việc tính output |

Trong quá trình thiết kế và mô phỏng mô hình LSTM ở cấp độ phần cứng, việc xác định chính xác các cổng đầu vào và đầu ra của mô-đun lstm\_neural có ý nghĩa đặc biệt quan trọng. Đầu vào không chỉ bao gồm tín hiệu dữ liệu tuần tự (X) mà còn phải có các tín hiệu điều khiển như **clk, rst, rdy\_to\_receiv và rdy\_to\_send** nhằm đảm bảo hoạt động đồng bộ và luồng dữ liệu chính xác trong môi trường pipeline. Tương tự, các cổng đầu ra như Ht (trạng thái ẩn), Ct (cell state), **is\_rdy\_to\_receiv, is\_rdy\_to\_send và is\_rdy\_to\_output** đóng vai trò truyền tải kết quả trung gian cũng như báo hiệu trạng thái sẵn sàng cho các khối xử lý tiếp theo. Việc phân tách rõ ràng đầu vào ra không chỉ giúp quá trình triển khai SystemC hoặc synthesis lên FPGA trở nên mạch lạc và dễ kiểm soát, mà còn tăng độ tin cậy trong việc phối hợp giữa nhiều khối xử lý nối tiếp nhau trong kiến trúc tổng thể của TSMAE. Nếu lựa chọn sai hoặc không đầy đủ các cổng này, hệ thống có thể xảy ra lỗi timing, sai dữ liệu hoặc deadlock trong truyền nhận tín hiệu. Do đó, đầu vào/ra của lstm\_neural cần được thiết kế tỉ mỉ, đúng logic thời gian và phù hợp với yêu cầu mô hình toán học gốc.

Sau khi đã chọn được đầu vào và ra của module thì tiếp theo đó ta sẽ xây dựng diagram flow sao cho đúng với chức năng cũng như là các đầu vào ra mà ta đã thiết kế.

### LSTM\_neural diagram flow

A diagram of a flowchart

Description automatically generated

Hình : LSTM\_neural flow diagram

Mô-đun lstm\_neural được thiết kế theo nguyên lý xử lý tuần tự đầu vào nhưng xuất dữ liệu ra theo dạng song song, đảm bảo vừa đúng với logic của mạng LSTM, vừa thuận tiện cho tích hợp phần cứng. Việc nhận dữ liệu đầu vào chỉ diễn ra **khi nhận được tín hiệu handshake từ khối trước (rdy\_to\_receiv)**, nhằm đảm bảo rằng dữ liệu được truyền vào đúng thời điểm và không bị ghi đè. Sau mỗi lần tính toán, **dữ liệu đầu ra của LSTM (bao gồm Ht và Ct) sẽ được feedback trở lại như đầu vào cho bước kế tiếp**, mô phỏng chính xác cơ chế trạng thái lặp (recurrent) trong mạng LSTM.

Khi toàn bộ **140 mẫu dữ liệu trong một chuỗi (1 sequence)** đã được xử lý đầy đủ, mô-đun mới **phát tín hiệu handshake is\_rdy\_to\_send để báo hiệu sẵn sàng truyền dữ liệu ra ngoài**, đồng thời thiết lập lại các trạng thái tạm pre\_hidden và pre\_cell\_state về 0 để chuẩn bị cho chuỗi kế tiếp. Nếu có tín hiệu reset (rst), toàn bộ các thanh ghi bên trong mô-đun (trừ ROM chứa trọng số) sẽ được xóa về 0 để đảm bảo khởi tạo lại toàn trạng thái hệ thống. Ngoài ra, trong mỗi bước xử lý của chuỗi đầu vào, mô-đun sẽ phát tín hiệu is\_rdy\_to\_output cho output\_layer tương ứng với từng giá trị X\_in, giúp đồng bộ hóa việc lấy đầu ra tại tầng kế tiếp. Tất cả tín hiệu điều khiển được tổ chức chặt chẽ thông qua cơ chế handshake, đảm bảo pipeline hoạt động ổn định, tuần tự và không xung đột.

### Code cho LSTM\_neural

//Khai báo các chân vào ra và các biến nội bộ dùng trong LSTM\_neural:

#include <systemc.h>

#include <iostream>

SC\_MODULE(LSTM\_neural) {

sc\_in<bool> clk;

sc\_in<bool> rst;

sc\_in<bool> rdy\_to\_receiv;

sc\_in<bool> rdy\_to\_send;

sc\_in<bool> rdy\_to\_output;

sc\_in<float> x; // Single input

sc\_out<float> ht[10];

sc\_out<bool> is\_rdy\_to\_send;

sc\_out<bool> is\_rdy\_to\_output;

sc\_out<bool> is\_rdy\_to\_receiv;

sc\_signal<float> c\_feedback[10];

sc\_signal<float> h\_feedback[10];

float f\_bias\_reg[10], i\_bias\_reg[10], o\_bias\_reg[10], c\_bias\_reg[10];

float xf\_bias\_reg[10], xi\_bias\_reg[10], xo\_bias\_reg[10], xc\_bias\_reg[10];

float xf\_weight\_matrix[10], xi\_weight\_matrix[10];

float xo\_weight\_matrix[10], xc\_weight\_matrix[10];

float xf\_mul[10], xi\_mul[10], xo\_mul[10], xc\_mul[10];

float f\_weight\_matrix[10][10], i\_weight\_matrix[10][10];

float o\_weight\_matrix[10][10], c\_weight\_matrix[10][10];

float f\_mul[10], i\_mul[10], o\_mul[10], c\_mul[10];

float ht\_1\_reg[10], ct\_1\_reg[10];

float ht\_reg[10], ct\_reg[10];

int hidden\_count = 0;

//Cấu hình bên trong module và khai báo các hàm:

SC\_CTOR(LSTM\_neural) {

SC\_CTHREAD(lstm\_process, clk.pos());

async\_reset\_signal\_is(rst, true);

sensitive << clk.pos();

}

void lstm\_process();

float sigmoid(float x);

float tanh(float x);

float exp\_approx(float x);

};

*//LSTM\_neural process*

void LSTM\_neural::lstm\_process() {

while (true) {

if (rst.read()) { //reset

hidden\_count = 0;

for (int i = 0; i < 10; i++) {

xf\_mul[i] = xi\_mul[i] = xo\_mul[i] = xc\_mul[i] = 0;

f\_mul[i] = i\_mul[i] = o\_mul[i] = c\_mul[i] = 0;

ht\_1\_reg[i] = ct\_1\_reg[i] = ht\_reg[i] = ct\_reg[i] = 0;

ht[i].write(0);

h\_feedback[i].write(0);

c\_feedback[i].write(0);

wait();

}

} else {

is\_rdy\_to\_receiv.write(1);

do { wait(); } while (!rdy\_to\_receiv.read()); //tín hiệu handshake cho việc nhận x

is\_rdy\_to\_receiv.write(0);

float x\_val = x.read();

if (hidden\_count == 0) { //nếu đã chạy xong sequence thì cho ht\_1 và ct\_1 bằng 0

for (int i = 0; i < 10; i++) {

ht\_1\_reg[i] = 0;

ct\_1\_reg[i] = 0;

wait();

}

} else {

for (int i = 0; i < 10; i++) { //nhận ht\_1 và ct\_1

ht\_1\_reg[i] = h\_feedback[i].read();

ct\_1\_reg[i] = c\_feedback[i].read();

wait();

}

}

//nhân ma trận

for (int i = 0; i < 10; i++) {

xf\_mul[i] = x\_val \* xf\_weight\_matrix[i];

xi\_mul[i] = x\_val \* xi\_weight\_matrix[i];

xo\_mul[i] = x\_val \* xo\_weight\_matrix[i];

xc\_mul[i] = x\_val \* xc\_weight\_matrix[i];

wait();

}

for (int i = 0; i < 10; i++) {

f\_mul[i] = i\_mul[i] = o\_mul[i] = c\_mul[i] = 0;

wait();

}

for (int i = 0; i < 10; i++) {

for (int j = 0; j < 10; j++) {

f\_mul[i] += ht\_1\_reg[j] \* f\_weight\_matrix[j][i];

i\_mul[i] += ht\_1\_reg[j] \* i\_weight\_matrix[j][i];

o\_mul[i] += ht\_1\_reg[j] \* o\_weight\_matrix[j][i];

c\_mul[i] += ht\_1\_reg[j] \* c\_weight\_matrix[j][i];

wait();

}

}

for (int i = 0; i < 10; i++) {

f\_mul[i] += xf\_mul[i] + f\_bias\_reg[i] + xf\_bias\_reg[i];

i\_mul[i] += xi\_mul[i] + i\_bias\_reg[i] + xi\_bias\_reg[i];

o\_mul[i] += xo\_mul[i] + o\_bias\_reg[i] + xo\_bias\_reg[i];

c\_mul[i] += xc\_mul[i] + c\_bias\_reg[i] + xc\_bias\_reg[i];

wait();

}

//tính hàm kích hoạt

for (int i = 0; i < 10; i++) {

f\_mul[i] = sigmoid(f\_mul[i]);

i\_mul[i] = sigmoid(i\_mul[i]);

o\_mul[i] = sigmoid(o\_mul[i]);

c\_mul[i] = tanh(c\_mul[i]);

wait();

}

//tính Ct và Ht

for (int i = 0; i < 10; i++) {

ct\_reg[i] = f\_mul[i] \* ct\_1\_reg[i] + i\_mul[i] \* c\_mul[i];

ht\_reg[i] = o\_mul[i] \* tanh(ct\_reg[i]);

wait();

}

for (int i = 0; i < 10; i++) {

ht[i].write(ht\_reg[i]);

h\_feedback[i].write(ht\_reg[i]);

c\_feedback[i].write(ct\_reg[i]);

wait();

}

is\_rdy\_to\_output.write(1);

do { wait(); } while (!rdy\_to\_output.read());

is\_rdy\_to\_output.write(0);

hidden\_count++;

if (hidden\_count == 140) { //gửi Ht sau khi đã tính toán xong chuỗi

hidden\_count = 0;

is\_rdy\_to\_send.write(1);

do { wait(); } while (!rdy\_to\_send.read());

is\_rdy\_to\_send.write(0);

}

wait();

}

wait();

}

}

float LSTM\_neural::exp\_approx(float x) { //tính toán hàm ex dựa trên khai triển taylor

float result = 1.0, term = 1.0;

for (int i = 1; i <= 10; ++i) {

term \*= x / i;

result += term;

}

return result;

}

float LSTM\_neural::sigmoid(float x) { //hàm sigmoid

return 1.0f / (1.0f + exp\_approx(-x));

}

float LSTM\_neural::tanh(float x) { //hàm tanh

return (exp\_approx(x) - exp\_approx(-x)) / (exp\_approx(x) + exp\_approx(-x));

}

## Memory\_module

### Module Memory\_module

A white square with black text

Description automatically generated

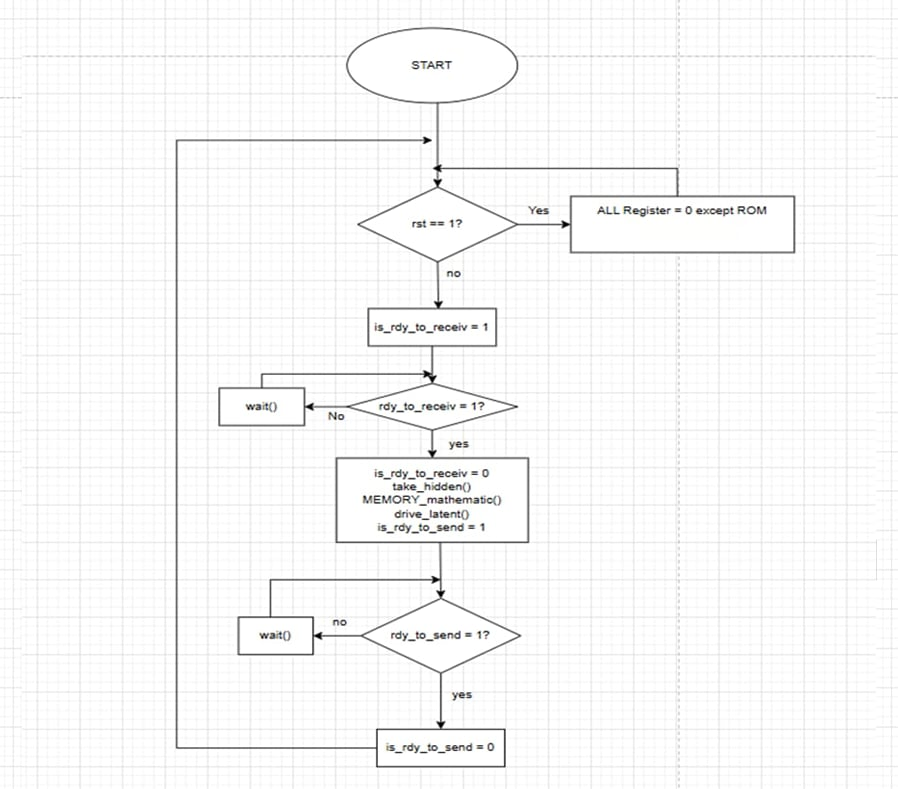
Hình : Module MEMORY\_module

Trong đó chức năng các chân:

Bảng : Các cổng vào ra của Memory\_module

|  |  |  |
| --- | --- | --- |
| **PORT NAME** | **TYPE** | **DESCRIPTION** |
| clk | Input bool | Nhận xung đồng hồ |
| rst | Input bool | reset |
| Last\_ht[10] | Input float | Hidden cuối cùng của 1 data sequence |
| Rdy\_to\_send | Input bool | Tín hiệu handshake vào cho việc truyền |
| Is\_rdy\_to\_send | Output bool | Tín hiệu handshake ra cho việc truyền |
| Rdy\_to\_receiv | Input bool | Tín hiệu handshake vào cho việc nhận |
| Is\_rdy\_to\_receiv | Output bool | Tín hiệu handshake ra cho việc nhận |
| Latent | Output float | Dữ liệu ra của memory |

### MEMORY\_module flow diagram

**

Hình : MEMORY\_module flow diagram

Sơ đồ trên mô tả chi tiết **luồng điều khiển của khối Memory Module** trong kiến trúc TSMAE. Quá trình bắt đầu từ trạng thái khởi tạo, trong đó tất cả các thanh ghi nội bộ ngoại trừ ROM được đặt về giá trị 0. Khi hệ thống thoát khỏi trạng thái reset (**rst** = 0), tín hiệu **is\_rdy\_to\_receiv** được bật để cho phép truyền dữ liệu từ khối trước.

Khối Memory sẽ liên tục kiểm tra tín hiệu **rdy\_to\_receiv**. Khi tín hiệu này được kích hoạt (**rdy\_to\_receiv** = 1), đồng nghĩa khối trước đã sẵn sàng gửi dữ liệu, hệ thống sẽ tắt tín hiệu **is\_rdy\_to\_receiv** để thực hiện ba thao tác chính:

1. **take\_hidden()** – nhận vector trạng thái ẩn từ encoder,
2. **MEMORY\_mathematic()** – thực hiện phép truy xuất bộ nhớ (ví dụ: so sánh với các mẫu đã học),
3. **drive\_latent()** – sinh ra biểu diễn ẩn đã tái hợp (latent vector) làm đầu vào cho decoder

Sau khi xử lý xong, tín hiệu **is\_rdy\_to\_send** được kích hoạt để báo hiệu dữ liệu đầu ra đã sẵn sàng truyền. Khối Memory tiếp tục chờ đến khi khối sau gửi lại tín hiệu rdy\_to\_send = 1 (báo hiệu đã sẵn sàng nhận). Khi đó, quá trình kết thúc và **is\_rdy\_to\_send** được hạ về 0 để chuẩn bị cho chu kỳ tiếp theo và lại chờ đời tín hiệu **rdy\_to\_receiv**

Toàn bộ quy trình được điều phối thông qua cơ chế tín hiệu handshake kết hợp cùng các điểm dừng wait() nhằm đảm bảo đồng bộ tuyệt đối và tránh xung đột truyền dữ liệu giữa các khối trong pipeline phần cứng. Luồng điều khiển này đảm bảo Memory Module chỉ hoạt động đúng thời điểm, chính xác về dữ liệu và giữ được tính tuyến tính của hệ thống LSTM.

### Code cho Memory\_module

#include <systemc.h>

#include <iostream>

SC\_MODULE(MemoryModule) {

    sc\_in<bool> clk;

    sc\_in<bool> rst;

    sc\_in<bool> rdy\_to\_receiv;

    sc\_in<bool> rdy\_to\_send;

    sc\_in<float> last\_ht[10]; // z input

    sc\_out<float> latent[10]; // output vector

    sc\_out<bool> is\_rdy\_to\_receiv;

    sc\_out<bool> is\_rdy\_to\_send;

    float mem\_item[20][10];       // memory items: m\_i

    float last\_ht\_reg[10];        // copy of input z

    float sim[20];                // zᵗm\_i

    float q[20];                  // softmax weights

    float latent\_reg[10];         // output latent vector

    SC\_CTOR(MemoryModule) {

        SC\_CTHREAD(process, clk.pos());

        async\_reset\_signal\_is(rst, true);

    }

    void process();

    float exp\_approx(float x);

};

//Memory\_process

void MemoryModule::process() {

    while (true) {

        if (rst.read()) {

            for (int i = 0; i < 10; i++) {

                latent[i].write(0.0f);

                wait(); // mỗi ghi = 1 chu kỳ

            }

        } else {

            // --- chờ sẵn sàng nhận

            is\_rdy\_to\_receiv.write(1);

            do {

                wait();

            } while (!rdy\_to\_receiv.read());

            is\_rdy\_to\_receiv.write(0);

            // --- đọc z vào last\_ht\_reg

            for (int i = 0; i < 10; i++) {

                last\_ht\_reg[i] = last\_ht[i].read();

                wait(); // mỗi read = 1 chu kỳ

            }

            // --- tính zᵗ m\_i và exp(sim)

            float sum\_exp = 0.0f;

            for (int i = 0; i < 20; i++) {

                sim[i] = 0.0f;

                wait(); // gán = 0.0 = 1 chu kỳ

                for (int j = 0; j < 10; j++) {

                    sim[i] += last\_ht\_reg[j] \* mem\_item[i][j];

                    wait(); // mỗi phép nhân + cộng = 1 chu kỳ

                }

                sim[i] = exp\_approx(sim[i]);

                wait(); // tính exp = 1 chu kỳ

                sum\_exp += sim[i];

                wait(); // cộng vào tổng = 1 chu kỳ

            }

            // --- softmax

            for (int i = 0; i < 20; i++) {

                q[i] = sim[i] / sum\_exp;

                wait(); // mỗi phép chia = 1 chu kỳ

            }

            // --- recombine latent = ∑ q\_i \* m\_i

            for (int j = 0; j < 10; j++) {

                latent\_reg[j] = 0.0f;

                wait(); // khởi tạo = 1 chu kỳ

                for (int i = 0; i < 20; i++) {

                    latent\_reg[j] += q[i] \* mem\_item[i][j];

                    wait(); // mỗi phép nhân + cộng = 1 chu kỳ

                }

                latent[j].write(latent\_reg[j]);

                wait(); // mỗi write = 1 chu kỳ

            }

            // --- chờ gửi

            is\_rdy\_to\_send.write(1);

            do {

                wait();

            } while (!rdy\_to\_send.read());

            is\_rdy\_to\_send.write(0);

        }

        wait(); // kết thúc vòng lặp

    }

}

float MemoryModule::exp\_approx(float x) { // tính hàm e^x bằng khai triển Taylor

    float result = 1.0f;

    float term = 1.0f;

    for (int i = 1; i <= 10; ++i) {

        term = term \* x / i;

        result += term;

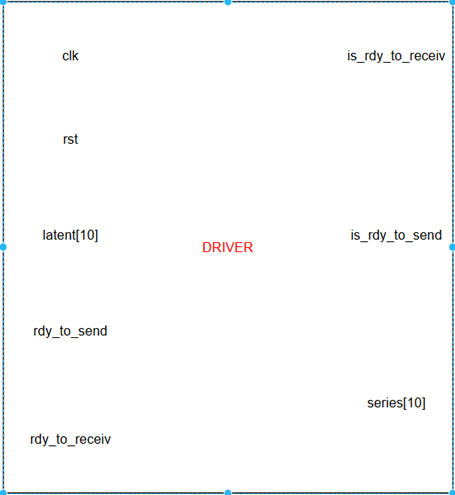
    }

    return result;

}

## Driver

### Module Driver



Hình : Module Driver

Trong đó các cổng vào ra của module Driver:

Bảng : Các cổng vào ra của module Driver

|  |  |  |
| --- | --- | --- |
| **PORT NAME** | **TYPE** | **DESCRIPTION** |
| Clk | Input bool | Xung clock |
| Rst | Input bool | Reset |
| Latent | Input float | Latent in |
| Rdy\_to\_receiv | Input bool | Tín hiệu handshake vào cho việc nhận |
| Is\_rdy\_to\_receiv | Output bool | Tín hiệu handshake ra cho việc nhận |
| Rdy\_to\_send | Input bool | Tín hiệu handshake vào cho việc truyền |
| Is\_rdy\_to\_send | Output bool | Tín hiệu handshake ra cho việc truyền |
| series | Output float | Lặp lại latent 32 lần |

### Driver flow diagram

A diagram of a flowchart

Description automatically generated

Hình : Driver flow diagram

Sơ đồ trên thể hiện **luồng điều khiển (FSM)** của khối **Driver**, hay còn gọi là **Latent Repeater**, với nhiệm vụ lặp lại vector latent đầu vào 32 lần để tạo thành chuỗi đầu vào cho decoder trong mô hình TSMAE. Khi hệ thống khởi động hoặc nhận tín hiệu reset (rst = 1), tất cả các thanh ghi được đưa về 0 (ngoại trừ ROM), và hệ thống chuyển sang trạng thái chờ nhận tín hiệu đầu vào.

Khi is\_rdy\_to\_receiv = 1 và rdy\_to\_receiv = 1 (có yêu cầu nhận và sẵn sàng từ khối trước), mô-đun sẽ thực hiện thao tác **take\_latent()** để nhận vector latent, đồng thời khởi tạo biến đếm count = 0. Sau đó, hệ thống đi vào vòng lặp lặp lại giá trị latent: nếu count < 140, mô-đun sẽ phát tín hiệu is\_rdy\_to\_send = 1 để báo sẵn sàng truyền dữ liệu ra, và đồng thời thực hiện **drive\_latent()** – gửi một bản sao của latent ra cổng series. Mỗi lần truyền thành công, count được tăng lên 1.

Tại mỗi vòng lặp, hệ thống kiểm tra tín hiệu rdy\_to\_send. Khi phía nhận xác nhận đã sẵn sàng (rdy\_to\_send = 1), hệ thống hạ is\_rdy\_to\_send = 0 để kết thúc một lần truyền, và tiếp tục đến lần lặp tiếp theo. Quá trình này được lặp lại cho đến khi đủ 140 lần, hoàn tất chuỗi đầu ra.

Toàn bộ điều khiển được thực hiện thông qua cơ chế **handshake hai chiều** (rdy\_to\_\* và is\_rdy\_to\_\*), đảm bảo truyền dữ liệu tuần tự, chính xác và không bị ghi đè hoặc thiếu bước. Đây là khối trung gian quan trọng giúp mô hình TSMAE hoạt động đúng logic tuần tự khi triển khai trên nền tảng phần cứng như SystemC hoặc FPGA.

### Code cho module Driver

#include <systemc.h>

#include <iostream>

SC\_MODULE(Mem\_de\_driver) {

    sc\_in<float> latent[10];

    sc\_in<bool> clk;

    sc\_in<bool> rst;

    sc\_in<bool> rdy\_to\_receiv;

    sc\_in<bool> rdy\_to\_send;

    sc\_out<bool> is\_rdy\_to\_send;

    sc\_out<bool> is\_rdy\_to\_receiv;

    sc\_out<float> latent\_series[10];

    float buffer[10];

    int count;

    int index;

    SC\_CTOR(Mem\_de\_driver) {

        SC\_CTHREAD(driver\_process, clk.pos());

        async\_reset\_signal\_is(rst, true);

    }

    void driver\_process();

};

// ========== Implementation phần tách ra ngoài ==========

void Mem\_de\_driver::driver\_process() {

    while (true) {

        if (rst.read()) {

            count = 0;

            index = 0;

            for (int i = 0; i < 10; i++) {

                buffer[i] = 0;

                latent\_series[i].write(0);

                wait();  // mỗi bước reset = 1 chu kỳ

            }

        } else {

            count = 0;

            is\_rdy\_to\_receiv.write(1);

            do {

                wait();  // chờ module trước sẵn sàng

            } while (!rdy\_to\_receiv.read());

            is\_rdy\_to\_receiv.write(0);

            // Đọc latent vào buffer từng phần tử

            for (int i = 0; i < 10; i++) {

                buffer[i] = latent[i].read();

                wait();  // mỗi read = 1 chu kỳ

            }

            while (count < 140) {

                for (index = 0; index < 10; index++) {

                    latent\_series[index].write(buffer[index]);

                    wait();  // mỗi write = 1 chu kỳ

                }

                is\_rdy\_to\_send.write(1);

                do {

                    wait();  // chờ bên nhận đọc xong

                } while (!rdy\_to\_send.read());

                is\_rdy\_to\_send.write(0);

                wait();  // giữ tín hiệu ổn định 1 chu kỳ

                count = count + 1;

                wait();  // mỗi lần phát xong +1 = 1 chu kỳ

            }

        }

        wait();  // cuối vòng lặp

    }

}

## Output\_Layer

### Module Output\_Layer

A screenshot of a computer

Description automatically generated

Hình : Module Output\_layer

Trong đó các công cụ vào ra:

Bảng : Các cổng vào ra của module Driver

|  |  |  |
| --- | --- | --- |
| **PORT NAME** | **TYPE** | **DESCRIPTION** |
| Clk | Input bool | Xung clock |
| Rst | Input bool | Reset |
| Ht[10] | Input float | Dữ liệu hidden vào |
| Rdy\_to\_receiv | Input bool | Tín hiệu handshake vào cho việc nhận |
| Is\_rdy\_to\_receiv | Output bool | Tín hiệu handshake ra cho việc nhận |
| Rdy\_to\_send | Output bool | Tín hiệu handshake vào cho việc truyền |
| Is\_rdy\_to\_send | Output bool | Tín hiệu handshake ra cho việc truyền |
| X\_recon | Output float | Dữ liệu tái tạo từ hidden |

### Output\_layer flow diagram

A diagram of a program

Description automatically generated

Hình : Output\_layer flow diagram

Sơ đồ trên thể hiện **luồng điều khiển (Finite State Machine – FSM)** của **khối Output Layer** trong mô hình TSMAE, nơi thực hiện việc xử lý đầu ra cuối cùng từ Ht để tái tạo lại chuỗi dữ liệu gốc (X\_recon). Quá trình bắt đầu từ trạng thái khởi tạo, nơi hệ thống kiểm tra tín hiệu rst. Nếu **rst** = 1, tất cả các thanh ghi ngoại trừ ROM sẽ được reset về 0, sau đó chuyển sang trạng thái sẵn sàng nhận dữ liệu (**is\_rdy\_to\_receiv** = 1).

Khi tín hiệu **rdy\_to\_receiv** đều được kích hoạt, khối sẽ tiến hành ba bước quan trọng:

1. **take\_hidden()**: nhận dữ liệu trạng thái ẩn (Ht) từ LSTM decoder.
2. **OUTPUT\_mathematic()**: thực hiện tính toán tuyến tính để ánh xạ từ không gian ẩn về không gian đầu ra gốc.
3. **drive\_x\_recon()**: tạo đầu ra cuối cùng X\_recon.

Sau đó, tín hiệu **is\_rdy\_to\_send** = 1 được kích hoạt để báo hiệu dữ liệu đã sẵn sàng gửi đi. Hệ thống sẽ chờ tín hiệu xác nhận từ phía nhận (**rdy\_to\_send** = 1) trước khi hạ **is\_rdy\_to\_send** = 0 và quay lại trạng thái ban đầu, sẵn sàng cho chu kỳ xử lý kế tiếp.

Cơ chế handshake hai chiều được duy trì xuyên suốt (**rdy\_to\_\*** và **is\_rdy\_to\_\*)** cùng với các điểm dừng wait() tại các nút kiểm tra trạng thái, giúp đảm bảo truyền nhận dữ liệu một cách đồng bộ, ổn định và tránh xung đột tín hiệu trong pipeline xử lý của mô hình. Khối này đóng vai trò hoàn tất chuỗi hoạt động TSMAE bằng cách đưa ra tín hiệu đầu ra X\_recon dùng cho đánh giá bất thường.

### Code cho Output\_layer

#include <systemc.h>

#include <iostream>

SC\_MODULE(Output\_layer) {

    sc\_in<bool> rst;

    sc\_in<bool> clk;

    sc\_in<float> ht[10];

    sc\_out<float> x\_recon;

    sc\_in<bool> rdy\_to\_receiv;

    sc\_in<bool> rdy\_to\_send;

    sc\_out<bool> is\_rdy\_to\_send;

    sc\_out<bool> is\_rdy\_to\_receiv;

    float output\_weight[10];

    float output\_bias;

    float mul;

    float ht\_reg[10];

    SC\_CTOR(Output\_layer) {

        SC\_CTHREAD(process, clk.pos());

        async\_reset\_signal\_is(rst, true);

    }

    void process();

};

// ========== Implementation ==========

void Output\_layer::process() {

    while (true) {

        if (rst.read()) {

            wait(); // thiết lập tín hiệu = 1 chu kỳ

            for (int i = 0; i < 10; i++) {

                ht\_reg[i] = 0;

                wait(); // reset từng phần tử = 1 chu kỳ

            }

            mul = 0;

            wait(); // reset tích đầu ra

        } else {

            is\_rdy\_to\_receiv.write(1);

            wait(); // báo sẵn sàng nhận

            do {

                wait(); // chờ module trước cho phép

            } while (!rdy\_to\_receiv.read());

            is\_rdy\_to\_receiv.write(0);

            wait(); // clear lại tín hiệu

            for (int i = 0; i < 10; i++) {

                ht\_reg[i] = ht[i].read();

                wait(); // đọc từng phần tử ht = 1 chu kỳ

            }

            mul = 0;

            wait(); // reset tích đầu ra

            for (int i = 0; i < 10; i++) {

                mul += ht\_reg[i] \* output\_weight[i];

                wait(); // mỗi nhân + cộng = 1 chu kỳ

            }

            mul += output\_bias;

            wait(); // cộng bias = 1 chu kỳ

            x\_recon.write(mul);

            wait(); // ghi output = 1 chu kỳ

            is\_rdy\_to\_send.write(1);

            wait(); // báo ready = 1 chu kỳ

            do {

                wait(); // chờ module sau sẵn sàng nhận

            } while (!rdy\_to\_send.read());

            is\_rdy\_to\_send.write(0);

            wait(); // clear lại tín hiệu

        }

        wait(); // cuối vòng lặp

    }

}

## TSMAE

Sau khi xây dựng các module nhỏ lẻ thì cần một module lớn TSMAE kết nối các module nhỏ đó lại đồng thời cũng tạo ra một hàm load\_weights() để có thể nạp các trọng số vào bên trong các biến nội bộ được coi như là ROM của các module.

### Code TSMAE

#ifndef TSMAE\_H

#define TSMAE\_H

#include "systemc.h"

#include <iostream>

#include "LSTM\_neural.h"

#include "MemoryModule.h"

#include "Mem\_de\_driver.h"

#include "Output\_layer.h"

#include "decoder.h"

#include "weight.h"

SC\_MODULE(TSMAE) {

sc\_in<bool> clk;

sc\_in<bool> rst;

sc\_in<float> x;

sc\_in<bool> rdy\_to\_receiv;

sc\_in<bool> rdy\_to\_send;

sc\_in<bool> rdy\_to\_output;

sc\_out<bool> is\_rdy\_to\_output;

sc\_out<bool> is\_rdy\_to\_send;

sc\_out<bool> is\_rdy\_to\_receiv;

sc\_out<float> x\_recon;

// Modules

LSTM\_neural \*encoder;

Decoder \*decoder;

MemoryModule \*memory\_module;

Mem\_de\_driver \*driver;

Output\_layer \*output\_layer;

// Internal registers

float x\_recon\_reg[32];

float loss;

int count;

// Signals

sc\_signal<bool> output\_done\_ques, output\_done\_ans;

sc\_signal<bool> en\_mem\_ques, en\_mem\_ans;

sc\_signal<bool> mem\_driver\_ques, mem\_driver\_ans;

sc\_signal<bool> driver\_de\_ques, driver\_de\_ans;

sc\_signal<bool> de\_out\_ques, de\_out\_ans;

sc\_signal<bool> queue\_read\_en;

sc\_signal<bool> output\_en, for\_free, en\_for\_free, de\_for\_free;

sc\_signal<float> x\_recon\_wire;

sc\_signal<float> queue\_out;

sc\_signal<float> en\_mem\_hidden[10];

sc\_signal<float> de\_out\_hidden[10];

sc\_signal<float> mem\_driver\_latent[10];

sc\_signal<float> driver\_de\_series[10];

SC\_CTOR(TSMAE) {

output\_en.write(1);

// Module init

encoder = new LSTM\_neural("encoder");

decoder = new Decoder("decoder");

memory\_module = new MemoryModule("memory\_module");

output\_layer = new Output\_layer("output\_layer");

driver = new Mem\_de\_driver("driver");

// Encoder

encoder->clk(clk); encoder->rst(rst);

encoder->rdy\_to\_receiv(rdy\_to\_receiv);

encoder->is\_rdy\_to\_receiv(is\_rdy\_to\_receiv);

encoder->rdy\_to\_output(rdy\_to\_output);

encoder->is\_rdy\_to\_output(is\_rdy\_to\_output);

encoder->x(x);

encoder->rdy\_to\_send(en\_mem\_ans);

encoder->is\_rdy\_to\_send(en\_mem\_ques);

for(int i = 0; i < 10; i++) encoder->ht[i](en\_mem\_hidden[i]);

// Memory

memory\_module->clk(clk); memory\_module->rst(rst);

memory\_module->rdy\_to\_receiv(en\_mem\_ques);

memory\_module->is\_rdy\_to\_receiv(en\_mem\_ans);

for (int i = 0; i < 10 ; i++) {

memory\_module->last\_ht[i](en\_mem\_hidden[i]);

memory\_module->latent[i](mem\_driver\_latent[i]);

}

memory\_module->rdy\_to\_send(mem\_driver\_ans);

memory\_module->is\_rdy\_to\_send(mem\_driver\_ques);

// Driver

driver->clk(clk); driver->rst(rst);

for (int i = 0; i < 10; i++) {

driver->latent[i](mem\_driver\_latent[i]);

driver->latent\_series[i](driver\_de\_series[i]);

}

driver->is\_rdy\_to\_receiv(mem\_driver\_ans);

driver->rdy\_to\_receiv(mem\_driver\_ques);

driver->is\_rdy\_to\_send(driver\_de\_ques);

driver->rdy\_to\_send(driver\_de\_ans);

// Decoder

decoder->clk(clk); decoder->rst(rst);

decoder->rdy\_to\_receiv(driver\_de\_ques);

decoder->is\_rdy\_to\_receiv(driver\_de\_ans);

decoder->rdy\_to\_output(de\_out\_ans);

decoder->is\_rdy\_to\_output(de\_out\_ques);

decoder->rdy\_to\_send(de\_for\_free);

decoder->is\_rdy\_to\_send(for\_free);

for(int i = 0; i < 10; i++) {

decoder->ht[i](de\_out\_hidden[i]);

decoder->x[i](driver\_de\_series[i]);

}

// Output layer

output\_layer->clk(clk); output\_layer->rst(rst);

output\_layer->rdy\_to\_receiv(de\_out\_ques);

output\_layer->is\_rdy\_to\_receiv(de\_out\_ans);

output\_layer->rdy\_to\_send(rdy\_to\_send);

output\_layer->is\_rdy\_to\_send(is\_rdy\_to\_send);

for(int i = 0; i < 10 ; i++) output\_layer->ht[i](de\_out\_hidden[i]);

output\_layer->x\_recon(x\_recon);

}

void load\_weights();

float pow(float x);

};

// ============================================

// ========== IMPLEMENTATION BELOW ============

// ============================================

void TSMAE::load\_weights() {

output\_layer->output\_bias = output\_bias;

for (int i = 0; i < 10; i++) {

decoder->c\_bias\_reg[i] = decoder\_bias\_hh\_cell[i];

decoder->f\_bias\_reg[i] = decoder\_bias\_hh\_forget[i];

decoder->i\_bias\_reg[i] = decoder\_bias\_hh\_input[i];

decoder->o\_bias\_reg[i] = decoder\_bias\_hh\_output[i];

decoder->xc\_bias\_reg[i] = decoder\_bias\_ih\_cell[i];

decoder->xf\_bias\_reg[i] = decoder\_bias\_ih\_forget[i];

decoder->xi\_bias\_reg[i] = decoder\_bias\_ih\_input[i];

decoder->xo\_bias\_reg[i] = decoder\_bias\_ih\_output[i];

encoder->c\_bias\_reg[i] = encoder\_bias\_hh\_cell[i];

encoder->f\_bias\_reg[i] = encoder\_bias\_hh\_forget[i];

encoder->i\_bias\_reg[i] = encoder\_bias\_hh\_input[i];

encoder->o\_bias\_reg[i] = encoder\_bias\_hh\_output[i];

encoder->xc\_bias\_reg[i] = encoder\_bias\_ih\_cell[i];

encoder->xf\_bias\_reg[i] = encoder\_bias\_ih\_forget[i];

encoder->xi\_bias\_reg[i] = encoder\_bias\_ih\_input[i];

encoder->xo\_bias\_reg[i] = encoder\_bias\_ih\_output[i];

encoder->xc\_weight\_matrix[i] = encoder\_weight\_ih\_cell[i];

encoder->xf\_weight\_matrix[i] = encoder\_weight\_ih\_forget[i];

encoder->xi\_weight\_matrix[i] = encoder\_weight\_ih\_input[i];

encoder->xo\_weight\_matrix[i] = encoder\_weight\_ih\_output[i];

output\_layer->output\_weight[i] = output\_layer\_weight\_row\_0[i];

for (int j = 0; j < 10; j++) {

decoder->c\_weight\_matrix[i][j] = decoder\_weight\_hh\_cell[i][j];

decoder->f\_weight\_matrix[i][j] = decoder\_weight\_hh\_forget[i][j];

decoder->i\_weight\_matrix[i][j] = decoder\_weight\_hh\_input[i][j];

decoder->o\_weight\_matrix[i][j] = decoder\_weight\_hh\_output[i][j];

decoder->xc\_weight\_matrix[i][j] = decoder\_weight\_ih\_cell[i][j];

decoder->xf\_weight\_matrix[i][j] = decoder\_weight\_ih\_forget[i][j];

decoder->xi\_weight\_matrix[i][j] = decoder\_weight\_ih\_input[i][j];

decoder->xo\_weight\_matrix[i][j] = decoder\_weight\_ih\_output[i][j];

encoder->c\_weight\_matrix[i][j] = encoder\_weight\_hh\_cell[i][j];

encoder->f\_weight\_matrix[i][j] = encoder\_weight\_hh\_forget[i][j];

encoder->i\_weight\_matrix[i][j] = encoder\_weight\_hh\_input[i][j];

encoder->o\_weight\_matrix[i][j] = encoder\_weight\_hh\_output[i][j];

}

}

for (int i = 0; i < 20; i++) {

for (int j = 0; j < 10; j++) {

memory\_module->mem\_item[i][j] = memory\_matrix[i][j];

}

}

}

float TSMAE::pow(float x) {

return x \* x;

}

#endif // TSMAE\_H

### Giải thích code

Đoạn mã trên định nghĩa mô-đun chính TSMAE trong SystemC, là sự hiện thực hóa toàn bộ pipeline của mô hình **Time-Series Memory-Augmented Autoencoder (TSMAE)** ở cấp độ phần cứng. Mô hình bao gồm các khối xử lý tương ứng với từng giai đoạn trong pipeline của mô hình học sâu: **LSTM Encoder**, **Memory Module**, **Latent Driver**, **LSTM Decoder**, và **Output Layer**.

**1. Cổng vào/ra và tín hiệu điều khiển**

Mô-đun khai báo các cổng chuẩn để tương tác với testbench hoặc môi trường bên ngoài:

* x: tín hiệu đầu vào là chuỗi thời gian,
* x\_recon: đầu ra tái tạo từ mô hình,
* Các tín hiệu handshake: rdy\_to\_receiv, is\_rdy\_to\_receiv, rdy\_to\_send, is\_rdy\_to\_send, rdy\_to\_output, is\_rdy\_to\_output để đảm bảo đồng bộ giữa các khối.

Ngoài ra, hệ thống còn sử dụng một loạt **sc\_signal nội bộ** để kết nối giữa các mô-đun thành phần, chia nhỏ tín hiệu vector thành các dòng dữ liệu song song ([10]) để phục vụ truyền từng giá trị giữa các tầng.

**2. Khởi tạo và kết nối các mô-đun**

Trong constructor SC\_CTOR(TSMAE), từng khối được khởi tạo và kết nối với nhau thông qua các cổng và tín hiệu:

* **LSTM Encoder** nhận x và sinh ra trạng thái ẩn ht[10], truyền đến khối Memory.
* **Memory Module** nhận ht[10] và xuất latent[10].
* **Driver** lặp lại latent[10] thành chuỗi series[10] để tạo chuỗi cho decoder.
* **LSTM Decoder** sử dụng chuỗi series để sinh ra trạng thái ẩn tại từng bước.
* **Output Layer** lấy ht[10] từ decoder để tính ra giá trị tái tạo x\_recon.

Việc kết nối giữa các khối đều tuân theo logic handshake để đảm bảo tính đồng bộ trong môi trường xung nhịp thực tế.

**3. Nạp trọng số mô hình**

Phương thức load\_weights() trong mô-đun thực hiện việc nạp toàn bộ trọng số và bias đã được huấn luyện từ các mảng C tĩnh (\*\_weight\_matrix, \*\_bias\_reg, v.v.). Quá trình nạp bao gồm:

* Trọng số và bias cho cả encoder và decoder theo từng cổng LSTM (input, forget, output, cell),
* Trọng số của lớp tuyến tính cuối (output layer),
* Ma trận bộ nhớ mem\_item chứa các mẫu điển hình trong Memory Module.

Việc nạp trọng số tách riêng như vậy cho phép mô hình học sâu có thể được huấn luyện bằng Python (PyTorch) và sau đó xuất sang các file .h hoặc .txt để sử dụng lại trong SystemC, phù hợp với luồng thiết kế mô hình AI phần cứng.

**4. Tính năng bổ sung**

Mô-đun có thêm hàm float pow(float x) dùng để tính bình phương, đơn giản hóa khi cần đo sai số hoặc hỗ trợ các phép toán trong loss function nếu được mở rộng.

## Tổng kết chương

Trong chương này, mô hình TSMAE đã được hiện thực hóa ở cấp độ mô-đun phần cứng bằng ngôn ngữ **SystemC**, với cấu trúc rõ ràng gồm năm thành phần chính: **LSTM Encoder**, **Memory Module**, **Latent Driver**, **LSTM Decoder** và **Output Layer**. Mỗi khối được thiết kế độc lập theo dạng SC\_MODULE, đảm nhận một vai trò tương ứng với pipeline logic trong mô hình học sâu gốc, đồng thời kết nối thông qua cơ chế **handshake tín hiệu hai chiều** để đảm bảo đồng bộ dữ liệu trong môi trường tuần tự theo xung nhịp.

Đặc biệt, các khối LSTM encoder và decoder được tổ chức đầy đủ theo đúng cấu trúc cổng vào/ra và trọng số như trong mô hình huấn luyện ban đầu, cho phép dễ dàng **nạp trọng số từ bên ngoài** (đã được huấn luyện bằng PyTorch). Khối Memory Module sử dụng logic so khớp để truy xuất vector latent từ bộ nhớ đã học, trong khi Driver và Output Layer đảm nhiệm việc tái tạo chuỗi đầu ra.

Tất cả các mô-đun đều đã được xây dựng hoàn chỉnh, hỗ trợ reset, kết nối clock và quản lý trạng thái nội bộ bằng hệ thống thanh ghi riêng biệt. Tuy nhiên, tại thời điểm này, **việc kiểm thử toàn hệ thống vẫn chưa được tiến hành** vì chưa tích hợp testbench. Do đó, chương tiếp theo sẽ tập trung vào việc thiết kế **testbench SystemC** cũng như là kiểm thử so với code chạy bằng python, tổ chức đầu vào, mô phỏng tín hiệu, và xác thực tính đúng đắn của toàn bộ pipeline TSMAE dưới dạng phần cứng.

# KIỂM TRA VÀ MÔ PHỎNG HỆ THỐNG

## Testbench cho systemC

### Code cho testbench

#ifndef TSMAE\_H

#define TSMAE\_H

#include "systemc.h"

#include <iostream>

#include "LSTM\_neural.h"

#include "MemoryModule.h"

#include "Mem\_de\_driver.h"

#include "Output\_layer.h"

#include "decoder.h"

#include "weight.h"

SC\_MODULE(TSMAE) {

sc\_in<bool> clk;

sc\_in<bool> rst;

sc\_in<float> x;

sc\_in<bool> rdy\_to\_receiv;

sc\_in<bool> rdy\_to\_send;

sc\_in<bool> rdy\_to\_output;

sc\_out<bool> is\_rdy\_to\_output;

sc\_out<bool> is\_rdy\_to\_send;

sc\_out<bool> is\_rdy\_to\_receiv;

sc\_out<float> x\_recon;

// Modules

LSTM\_neural \*encoder;

Decoder \*decoder;

MemoryModule \*memory\_module;

Mem\_de\_driver \*driver;

Output\_layer \*output\_layer;

// Internal registers

float x\_recon\_reg[32];

float loss;

int count;

// Signals

sc\_signal<bool> output\_done\_ques, output\_done\_ans;

sc\_signal<bool> en\_mem\_ques, en\_mem\_ans;

sc\_signal<bool> mem\_driver\_ques, mem\_driver\_ans;

sc\_signal<bool> driver\_de\_ques, driver\_de\_ans;

sc\_signal<bool> de\_out\_ques, de\_out\_ans;

sc\_signal<bool> queue\_read\_en;

sc\_signal<bool> output\_en, for\_free, en\_for\_free, de\_for\_free;

sc\_signal<float> x\_recon\_wire;

sc\_signal<float> queue\_out;

sc\_signal<float> en\_mem\_hidden[10];

sc\_signal<float> de\_out\_hidden[10];

sc\_signal<float> mem\_driver\_latent[10];

sc\_signal<float> driver\_de\_series[10];

SC\_CTOR(TSMAE) {

output\_en.write(1);

// Module init

encoder = new LSTM\_neural("encoder");

decoder = new Decoder("decoder");

memory\_module = new MemoryModule("memory\_module");

output\_layer = new Output\_layer("output\_layer");

driver = new Mem\_de\_driver("driver");

// Encoder

encoder->clk(clk); encoder->rst(rst);

encoder->rdy\_to\_receiv(rdy\_to\_receiv);

encoder->is\_rdy\_to\_receiv(is\_rdy\_to\_receiv);

encoder->rdy\_to\_output(rdy\_to\_output);

encoder->is\_rdy\_to\_output(is\_rdy\_to\_output);

encoder->x(x);

encoder->rdy\_to\_send(en\_mem\_ans);

encoder->is\_rdy\_to\_send(en\_mem\_ques);

for(int i = 0; i < 10; i++) encoder->ht[i](en\_mem\_hidden[i]);

// Memory

memory\_module->clk(clk); memory\_module->rst(rst);

memory\_module->rdy\_to\_receiv(en\_mem\_ques);

memory\_module->is\_rdy\_to\_receiv(en\_mem\_ans);

for (int i = 0; i < 10 ; i++) {

memory\_module->last\_ht[i](en\_mem\_hidden[i]);

memory\_module->latent[i](mem\_driver\_latent[i]);

}

memory\_module->rdy\_to\_send(mem\_driver\_ans);

memory\_module->is\_rdy\_to\_send(mem\_driver\_ques);

// Driver

driver->clk(clk); driver->rst(rst);

for (int i = 0; i < 10; i++) {

driver->latent[i](mem\_driver\_latent[i]);

driver->latent\_series[i](driver\_de\_series[i]);

}

driver->is\_rdy\_to\_receiv(mem\_driver\_ans);

driver->rdy\_to\_receiv(mem\_driver\_ques);

driver->is\_rdy\_to\_send(driver\_de\_ques);

driver->rdy\_to\_send(driver\_de\_ans);

// Decoder

decoder->clk(clk); decoder->rst(rst);

decoder->rdy\_to\_receiv(driver\_de\_ques);

decoder->is\_rdy\_to\_receiv(driver\_de\_ans);

decoder->rdy\_to\_output(de\_out\_ans);

decoder->is\_rdy\_to\_output(de\_out\_ques);

decoder->rdy\_to\_send(de\_for\_free);

decoder->is\_rdy\_to\_send(for\_free);

for(int i = 0; i < 10; i++) {

decoder->ht[i](de\_out\_hidden[i]);

decoder->x[i](driver\_de\_series[i]);

}

// Output layer

output\_layer->clk(clk); output\_layer->rst(rst);

output\_layer->rdy\_to\_receiv(de\_out\_ques);

output\_layer->is\_rdy\_to\_receiv(de\_out\_ans);

output\_layer->rdy\_to\_send(rdy\_to\_send);

output\_layer->is\_rdy\_to\_send(is\_rdy\_to\_send);

for(int i = 0; i < 10 ; i++) output\_layer->ht[i](de\_out\_hidden[i]);

output\_layer->x\_recon(x\_recon\_wire);

}

void load\_weights();

float pow(float x);

};

// ============================================

// ========== IMPLEMENTATION BELOW ============

// ============================================

void TSMAE::load\_weights() {

output\_layer->output\_bias = output\_bias;

for (int i = 0; i < 10; i++) {

decoder->c\_bias\_reg[i] = decoder\_bias\_hh\_cell[i];

decoder->f\_bias\_reg[i] = decoder\_bias\_hh\_forget[i];

decoder->i\_bias\_reg[i] = decoder\_bias\_hh\_input[i];

decoder->o\_bias\_reg[i] = decoder\_bias\_hh\_output[i];

decoder->xc\_bias\_reg[i] = decoder\_bias\_ih\_cell[i];

decoder->xf\_bias\_reg[i] = decoder\_bias\_ih\_forget[i];

decoder->xi\_bias\_reg[i] = decoder\_bias\_ih\_input[i];

decoder->xo\_bias\_reg[i] = decoder\_bias\_ih\_output[i];

encoder->c\_bias\_reg[i] = encoder\_bias\_hh\_cell[i];

encoder->f\_bias\_reg[i] = encoder\_bias\_hh\_forget[i];

encoder->i\_bias\_reg[i] = encoder\_bias\_hh\_input[i];

encoder->o\_bias\_reg[i] = encoder\_bias\_hh\_output[i];

encoder->xc\_bias\_reg[i] = encoder\_bias\_ih\_cell[i];

encoder->xf\_bias\_reg[i] = encoder\_bias\_ih\_forget[i];

encoder->xi\_bias\_reg[i] = encoder\_bias\_ih\_input[i];

encoder->xo\_bias\_reg[i] = encoder\_bias\_ih\_output[i];

encoder->xc\_weight\_matrix[i] = encoder\_weight\_ih\_cell[i];

encoder->xf\_weight\_matrix[i] = encoder\_weight\_ih\_forget[i];

encoder->xi\_weight\_matrix[i] = encoder\_weight\_ih\_input[i];

encoder->xo\_weight\_matrix[i] = encoder\_weight\_ih\_output[i];

output\_layer->output\_weight[i] = output\_layer\_weight\_row\_0[i];

for (int j = 0; j < 10; j++) {

decoder->c\_weight\_matrix[i][j] = decoder\_weight\_hh\_cell[i][j];

decoder->f\_weight\_matrix[i][j] = decoder\_weight\_hh\_forget[i][j];

decoder->i\_weight\_matrix[i][j] = decoder\_weight\_hh\_input[i][j];

decoder->o\_weight\_matrix[i][j] = decoder\_weight\_hh\_output[i][j];

decoder->xc\_weight\_matrix[i][j] = decoder\_weight\_ih\_cell[i][j];

decoder->xf\_weight\_matrix[i][j] = decoder\_weight\_ih\_forget[i][j];

decoder->xi\_weight\_matrix[i][j] = decoder\_weight\_ih\_input[i][j];

decoder->xo\_weight\_matrix[i][j] = decoder\_weight\_ih\_output[i][j];

encoder->c\_weight\_matrix[i][j] = encoder\_weight\_hh\_cell[i][j];

encoder->f\_weight\_matrix[i][j] = encoder\_weight\_hh\_forget[i][j];

encoder->i\_weight\_matrix[i][j] = encoder\_weight\_hh\_input[i][j];

encoder->o\_weight\_matrix[i][j] = encoder\_weight\_hh\_output[i][j];

}

}

for (int i = 0; i < 20; i++) {

for (int j = 0; j < 10; j++) {

memory\_module->mem\_item[i][j] = memory\_matrix[i][j];

}

}

}

float TSMAE::pow(float x) {

return x \* x;

}

#endif // TSMAE\_H

### Giải thích code

Đoạn mã trên định nghĩa mô-đun **TestbenchTSMAE** trong SystemC, dùng để **kiểm thử và mô phỏng hoạt động của mô hình TSMAE** đã được cài đặt dưới dạng phần cứng (DUT – Device Under Test). Mô hình TSMAE được đóng gói trong đối tượng TSMAE\* tsmae, với đầy đủ các cổng vào/ra và cơ chế handshake phục vụ cho pipeline đồng bộ.

**1. Cấu trúc kết nối và tín hiệu**

Mô-đun testbench khai báo một số lượng lớn tín hiệu SystemC (sc\_signal) để thực hiện giao tiếp với khối TSMAE:

* Các cổng đồng hồ và reset: clk, rst
* Cổng dữ liệu: x\_sig (dữ liệu đầu vào), x\_recon\_sig (dữ liệu tái tạo)
* Các cổng handshake: rdy\_to\_receiv, is\_rdy\_to\_receiv, rdy\_to\_send, is\_rdy\_to\_send, rdy\_to\_output, is\_rdy\_to\_output

Đặc biệt, kiểu binding handshake được áp dụng là **hoán đổi chiều điều khiển**: tín hiệu rdy\_to\_send từ testbench sẽ nối với is\_rdy\_to\_receiv của DUT, tạo ra cơ chế xác nhận qua lại giữa hai khối theo nguyên tắc “sẵn sàng gửi” và “sẵn sàng nhận”.

**2. Khởi tạo mô hình và tải dữ liệu**

Trong phần constructor (SC\_CTOR), mô hình TSMAE được tạo ra và các cổng được liên kết với testbench. Trước khi chạy mô phỏng, mô hình gọi hàm load\_weights() để nạp trọng số từ tệp tin. Dữ liệu đầu vào là một chuỗi thực (std::vector<float> seq) được đọc từ file "normal\_sample0.txt" – tương ứng với một chuỗi thời gian đầu vào.

**3. Quy trình truyền và nhận dữ liệu**

Hai tiến trình độc lập send\_process() và receiv\_process() được khởi tạo bằng SC\_THREAD, mô phỏng **giao tiếp hai chiều song song** giữa testbench và mô hình.

* send\_process():
  + Ban đầu thực hiện reset hệ thống trong 2 chu kỳ xung nhịp (clk).
  + Sau đó, lần lượt gửi từng phần tử trong chuỗi seq vào cổng x\_sig, kết hợp bật tín hiệu is\_rdy\_to\_send và is\_rdy\_to\_output.
  + Testbench sẽ chờ tín hiệu rdy\_to\_send từ DUT phản hồi trước khi tiếp tục gửi bước tiếp theo.
* receiv\_process():
  + Đồng thời với quá trình gửi, tiến trình này chờ kết quả từ mô hình.
  + Với mỗi bước, testbench bật is\_rdy\_to\_receiv và đợi đến khi DUT phát tín hiệu rdy\_to\_receiv.
  + Khi có kết quả, giá trị x\_recon\_sig (đầu ra tái tạo) sẽ được in ra cùng với timestamp.

Kết thúc toàn bộ quá trình gửi và nhận, lệnh sc\_stop() được gọi để dừng mô phỏng.

### Kết quả sau khi mô phỏng

Log dưới đây là kết quả sau khi tái tạo lại chuỗi dữ liệu:

[421260 ns] x\_recon[125] = -2.26981

[423980 ns] x\_recon[126] = -1.46036

[423990 ns] x\_recon[127] = -1.46036

[426710 ns] x\_recon[128] = -1.04884

[426720 ns] x\_recon[129] = -1.04884

[429440 ns] x\_recon[130] = -0.869434

[429450 ns] x\_recon[131] = -0.869434

[432170 ns] x\_recon[132] = -0.632553

[432180 ns] x\_recon[133] = -0.632553

[434900 ns] x\_recon[134] = -0.37465

[434910 ns] x\_recon[135] = -0.37465

[437630 ns] x\_recon[136] = -0.102807

[437640 ns] x\_recon[137] = -0.102807

[440360 ns] x\_recon[138] = 0.187466

[440370 ns] x\_recon[139] = 0.187466

## Testbench cho python

### Code cho testbench

#!/usr/bin/env python3

import torch

import numpy as np

from torch.utils.data import DataLoader, TensorDataset

# Import your TSMAE class definition

from TSMAE import TSMAE

def test\_tsmae():

# File paths (hard-coded)

input\_file = 'normal\_sample0\_column.txt'

checkpoint = 'tsmae\_merged.pth'

output\_file = 'reconstructed.txt'

# Hyperparameters must match training

input\_size = 1

hidden\_size = 10

memory\_size = 20

sparsity\_factor = 0.001

# Select device

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

print(f"Using device: {device}")

# 1) Load model and weights

model = TSMAE(input\_size, hidden\_size, memory\_size, sparsity\_factor)

state = torch.load(checkpoint, map\_location=device)

model.load\_state\_dict(state)

model.to(device).eval()

# 2) Read input sequence

values = np.loadtxt(input\_file, dtype=np.float32) # shape (seq\_len,)

seq\_len = values.shape[0]

# Create tensor of shape (1, seq\_len, 1)

x = torch.from\_numpy(values).unsqueeze(0).unsqueeze(-1).to(device)

# 3) Forward pass

with torch.no\_grad():

x\_recon, q, z, z\_hat = model(x)

# 4) Save reconstructed values

recon = x\_recon.cpu().squeeze().numpy() # shape (seq\_len,)

np.savetxt(output\_file, recon, fmt="%.6f")

print(f"Saved reconstructed sequence ({seq\_len} points) to {output\_file}")

if \_\_name\_\_ == "\_\_main\_\_":

test\_tsmae()

### Giải thích Code

Đoạn mã trên là một **chương trình kiểm thử mô hình TSMAE sau huấn luyện** bằng ngôn ngữ Python, sử dụng thư viện PyTorch. Mục tiêu của đoạn mã là **tải mô hình đã được huấn luyện, thực hiện suy diễn (inference) trên một chuỗi đầu vào**, và ghi lại kết quả tái tạo để so sánh với dữ liệu gốc. Đây là bước quan trọng nhằm xác thực mô hình phần mềm trước khi chuyển sang mô phỏng phần cứng.

**1. Thiết lập và khởi tạo**

Chương trình bắt đầu bằng cách **định nghĩa các siêu tham số chính** như input\_size, hidden\_size, memory\_size, và sparsity\_factor, đảm bảo khớp với mô hình đã huấn luyện. Đường dẫn file được cố định gồm:

* normal\_sample0\_column.txt: chứa chuỗi đầu vào cần tái tạo,
* tsmae\_merged.pth: checkpoint lưu trọng số mô hình,
* reconstructed.txt: nơi lưu kết quả tái tạo.

**2. Tải mô hình**

Mô hình TSMAE được khởi tạo với các siêu tham số nêu trên. Trọng số đã huấn luyện được nạp thông qua load\_state\_dict, và mô hình được chuyển sang chế độ eval() để đảm bảo không có dropout hoặc batch norm.

**3. Chuẩn bị dữ liệu đầu vào**

Dữ liệu đầu vào là một chuỗi 1 chiều được đọc từ file văn bản và chuyển thành tensor có kích thước (1, seq\_len, 1) – đúng định dạng yêu cầu của mô hình TSMAE: 1 batch, chuỗi thời gian, và 1 đặc trưng tại mỗi bước.

**4. Suy diễn (forward pass)**

Mô hình thực hiện tái tạo chuỗi thông qua hàm model(x). Quá trình này không yêu cầu lan truyền ngược, do đó được đặt trong khối with torch.no\_grad() để tiết kiệm bộ nhớ và tăng tốc. Kết quả trả về gồm:

* x\_recon: chuỗi tái tạo,
* q: vector truy xuất bộ nhớ,
* z, z\_hat: biểu diễn ẩn trước và sau tăng cường.

**5. Xuất kết quả**

Sau khi suy diễn xong, dữ liệu x\_recon được chuyển về dạng numpy và ghi ra file reconstructed.txt. File này sau đó có thể dùng để:

* so sánh với dữ liệu gốc,
* tính sai số tái tạo,

đưa vào testbench phần cứng để đối chiếu đầu ra mô phỏng.

### Kết quả sau khi mô phỏng

Log dưới đây là kết quả sau khi tái tạo lại chuỗi dữ liệu

[132] -0.058808

[133] -0.158048

[134] -0.326958

[135] -0.482794

[136] -0.602048

[137] -0.694354

[138] -0.768951

[139] -0.830802

## So sánh kết quả giữa python và systemC

Để có thể kiểm tra mô hình TSMAE chạy bằng systemC đã hoạt động đúng hay chưa thì ta cần phải so sánh đồ thị của chuỗi ban đầu; chuỗi được tái tạo bởi Python và chuỗi được tái tạo bởi SystemC.

Dưới đây là chuỗi ban đầu:

A graph with a line

AI-generated content may be incorrect.

Hình : Chuỗi dữ liệu ban đầu

Chuỗi được tái tạo bằng Python:

A graph with a line

AI-generated content may be incorrect.

Hình : Chuỗi dữ liệu được tái tạo bằng Python

Chuỗi được tái tạo bằng SystemC: A graph with a line graph

AI-generated content may be incorrect.

Hình : Chuỗi dữ liệu được tái tạo bằng SystemC

## So sánh lượng tài nguyên sử dụng

### Tài nguyên yêu cầu cho Python

#### Hệ điều hành

* **Linux** (Ubuntu 18.04 trở lên, CentOS 7+)
* **Windows** 10 (64-bit)
* **macOS** 10.14 trở lên

#### Python

* **Phiên bản**: 3.7, 3.8, 3.9 hoặc 3.10
* Cài đặt qua **Anaconda/Miniconda** hoặc **pip**:

#### Phần cứng CPU-only

* **CPU**: x86\_64 (Intel/AMD) hoặc AArch64 (ARM64)
* **RAM**: tối thiểu 4 GB (8 GB trở lên khuyến nghị cho workloads nhỏ)
* **Disk**: ~1 GB trống cho bộ cài và cache gói

#### Nếu cần GPU acceleration

* **GPU NVIDIA** với kiến trúc Pascal (compute capability ≥6.0) trở lên.
* **Driver NVIDIA**: ≥450.x
* **CUDA Toolkit**:
  + CUDA 10.2 (tương thích PyTorch 1.8)
  + CUDA 11.3/11.6 (tương thích PyTorch 1.10+)
* **CUDNN**: tương ứng với phiên bản CUDA.

Thời gian mô hình chạy một chuỗi

A black and white screen with white text

AI-generated content may be incorrect.

Hình : Thời gian và tài nguyên máy tính khi chạy một chuỗi

### Tài nguyên cho SystemC

Sử dụng công cụ Xilinx Vitis để có thể tổng hợp được ngôn ngữ HLS tuy nhiên do Vitis không hỗ trợ synthesis cả hệ thống pipeline mà chỉ có thể synthesi từng module đơn lẻ sau đó ta sẽ tự tổng hợp lại tất cả lượng tài nguyên cần sử dụng

Bảng 8: Lượng phần c cần thiết cho mô hình TSMAE

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Module | Cycles | LUT | FF | DSP | BRAM |
| Encoder | 193 | 19524 | 21644 | 80 | 5 |
| Memory\_module | 525 | 8991 | 18507 | 17 | 2 |
| Driver | 11 | 460 | 277 | 0 | 0 |
| Decoder | 234 | 30250 | 26382 | 97 | 21 |
| Output | 35 | 1387 | 1604 | 7 | 0 |
| Tổng cộng | -- | 60612 | 68414 | 201 | 28 |

Đây là số tài nguyên phải bỏ ra trên mỗi Module còn về số chy kỳ thì chỉ là số chu kỳ cho 1 lần tính của module đó tức là để tính hết 1 chuỗi đầu vào với encoder thì sẽ phải cần 140 lần tính với chuỗi của ECG5000 dataset

Thời gian tính toán xong một chuỗi

[437620 ns] x\_recon[136] = -0.102807  
[437630 ns] x\_recon[137] = -0.102807  
[440350 ns] x\_recon[138] = 0.187466  
[440360 ns] x\_recon[139] = 0.187466  
Processed 140 samples in 44035 cycles (0.00044035 s)

## Kết luận

Như vậy là sau khi so sánh mô hình chạy bằng SystemC và Python ta có thể thấy được rằng là tốc độ tính toán của systemC có phần nhỉnh hơn hoàn toàn có thể đáp ứng tính Realtime (44035 cycles) mặc dù điều này còn phụ thuộc vào FPGA. Tuy rằng lượng tài nguyên mà FPGA cần phải dành cho TSMAE tương đối lớn đòi hỏi một FPGA tương đối khoẻ tuy nhiên nếu được đưa vào trong chu trình của ASIC và công nghệ hiện nay thì ta hoàn toàn có thể tạo ra các chip với kích thước nhỏ gọn để có thể mang bên người do không cần phải đáp ứng các yêu cầu khắt khe như một máy tính thông thường phải có đủ các thành phần cũng như là phải có hệ điều hành.

Tuy rằng kết quả có sự sai lệch khá lớn có lẽ là do việc trích xuất trọng số cũng như là trình biên dịch của systemC và Python có sự khác biệt nhỏ điều này làm sai lệch kết quả sau nhiều lần tính toán do việc tính toán khá là nặng và mang tính lan truyền. Tuy nhiên điều này vẫn cho thấy việc triển khai một mô hình học máy lên FPGA hay là một chip ASIC là hoàn toàn khả thi.

CHƯƠNG 5. KẾT LUẬN

Trong khuôn khổ đồ án này, mô hình **TSMAE (Temporal-Sparse Memory-Augmented Autoencoder)** đã được nghiên cứu, hiện thực và triển khai hoàn chỉnh từ cấp độ **thuật toán học sâu** đến **mô phỏng phần cứng**. TSMAE được lựa chọn vì khả năng phát hiện bất thường vượt trội trên dữ liệu chuỗi thời gian, nhờ vào cơ chế kết hợp giữa bộ nhớ học mẫu và mã hóa tuần tự bằng LSTM. Mô hình không chỉ tái hiện được các đặc trưng thời gian phức tạp mà còn có khả năng làm nổi bật sai lệch thông qua tái cấu trúc có điều hướng bộ nhớ.

Về mặt lý thuyết, báo cáo đã trình bày hệ thống kiến thức nền về mạng neural, RNN và LSTM, đặt cơ sở cho việc hiểu sâu kiến trúc TSMAE. Ở cấp độ triển khai phần mềm, mô hình được hiện thực đầy đủ bằng PyTorch và đã trải qua các bước huấn luyện, kiểm thử, đánh giá tái tạo với bộ dữ liệu ECG5000. Kết quả mô phỏng cho thấy TSMAE có thể học và phát hiện bất thường trong chuỗi một cách chính xác.

Điểm nổi bật của đồ án nằm ở việc **mô phỏng chi tiết toàn bộ pipeline mô hình bằng SystemC**, trong đó từng khối như Encoder, Memory Module, Decoder và Output Layer đều được xây dựng dưới dạng mô-đun phần cứng độc lập, giao tiếp qua cơ chế handshake để đảm bảo tính đồng bộ. Ngoài ra, trọng số huấn luyện từ PyTorch cũng được chuyển đổi và tích hợp thành công vào SystemC, giúp mô hình giữ nguyên tính logic khi chuyển sang môi trường phần cứng.

Tuy kết quả khi mô phỏng bằng systemC không đạt được như mong muốn điều đó có thể là do việc trích xuất trọng số sai do em chưa hiểu rõ về thư viện Pytorch của python. Nhưng mô hình chạy bằng SystemC vẫn có thể chạy và tính toán ra được chuỗi dữ liệu dù cho có phần không chính xác, điều này cho thấy mô hình hoàn toàn có thể triển khai được trên phần cứng. Và việc triển khai trên phần cứng trong thực tế vẫn chưa thể thực hiện được do thiếu sót về mặt công cụ vì công cụ synthesis từ code SystemC sang HDL là một bộ công cụ phải trả phí và chỉ được cung cấp cho bên doanh nghiệp.

Tổng thể, đồ án không chỉ chứng minh được khả năng xây dựng mô hình AI hiệu quả cho bài toán anomaly detection, mà còn cho thấy hướng tiếp cận mô phỏng phần cứng hoàn toàn khả thi, mở đường cho việc triển khai TSMAE trên FPGA hoặc các kiến trúc nhúng hiệu năng cao trong tương lai. Đây là bước quan trọng trong xu hướng tích hợp AI trực tiếp lên phần cứng để tối ưu tốc độ và hiệu quả năng lượng trong các ứng dụng thực tế như giám sát thiết bị, tài chính, hoặc chăm sóc sức khỏe.

TÀI LIỆU THAM KHẢO

1. M.J. et al, “The internet of things: Mapping the value beyond the hype”.https:[//https://www.mckinsey.com/~/media/mckinsey/industries/technology%](http://www.mckinsey.com/~/media/mckinsey/industries/technology%25)20media%20and%20telecommunications/high%20tech/our%20insights/the%20internet%20of%20things%20the%20value%20of%20digitizing%20the%20physical%20world/unlocking\_the\_potential\_of\_the\_internet\_of\_things\_executive\_summary.pdf, 2015.
2. D. S. Pham, S. Venkatesh, M. Lazarescu, *et al.*, “Anomaly detection in large-scale data stream networks,” *Data Mining and Knowledge Discovery*, vol. 28, pp. 145– 189, 2014.
3. M. Hasan, M. M. Islam, M. I. I. Zarif, and M. Hashem, “Attack and anomaly detection in iot sensors in iot sites using machine learning approaches,” *Internet of Things*, vol. 7, p. 100059, 2019
4. McCulloch, Walter Pitts, “a logical calculus of the ideas immanent in nervous activity” mathematical biophysics volume 5, 1943
5. Honghao Gao , Senior Member, IEEE, Binyang Qiu, Ramón J. Durán Barroso , Walayat Hussain , Yueshen Xu , Member, IEEE, and Xinheng Wang , Senior Member, IEE “TSMAE: A Novel Anomaly Detection Approach for Internet of Things Time Series Data Using Memory-Augmented Autoencoder”, IEEE transactions on network science and engineering, vol. 10, no. 5, september/october 2023