ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH

oOo

Báo cáo môn học Cơ Sơ Toán cho Khoa Học Máy Tính

NGHIÊN CỨU TỔNG QUAN VỀ LÝ THUYẾT CỦA ĐẠI SỐ TUYẾN TÍNH TRONG KHOA HỌC MÁY TÍNH

(An Overview Study of the Theory of Linear Algebra in Computer Science)

GIẢNG VIÊN HƯỚNG DẪN
TS. Nguyễn An Khương
TS. Trần Tuấn Anh
PGS. TS. Lê Hồng Trang

HỌC VIÊN
2570485 - Lê Nguyễn Nhật Quang
2570483 - Đàm Quang Phục
2470304 - Nguyễn Quang Mạnh
2570443 - Trần Lê Hoàng Long

LỜI MỞ ĐẦU

Trong suốt quá trình học học phần Cơ sở Toán cho Khoa học Máy tính, nhóm chúng em đã nhận được sự hướng dẫn tận tình và những kiến thức vô cùng quý báu từ Thầy TS. Nguyễn An Khương, Thầy TS. Trần Tuấn Anh và Thầy PGS.TS. Lê Hồng Trang. Những bài giảng, gợi mở và chỉ dẫn của Quý Thầy đã giúp chúng em hình thành nền tảng vững chắc trong việc hiểu, phân biệt và vận dụng các kiến thức toán học vào lĩnh vực khoa học máy tính.

Nhóm xin bày tỏ lòng biết ơn chân thành và sâu sắc tới Quý Thầy vì sự tận tâm trong giảng dạy cùng những định hướng ý nghĩa dành cho chúng em trong suốt quá trình học tập. Những kiến thức, kinh nghiệm và lời khuyên của Quý Thầy không chỉ hỗ trợ chúng em trong học tập và nghiên cứu hiện tại, mà còn là hành trang lâu dài cho con đường nghề nghiệp và cuộc sống sau này.

Cuối cùng, nhóm chúng em kính chúc Quý Thầy luôn dồi dào sức khỏe, nhiều niềm vui và giữ vững ngọn lửa nhiệt huyết để tiếp tục cống hiến cho sự nghiệp giáo dục, truyền đạt tri thức đến các thế hệ sinh viên tương lai.

TP. Hồ Chí Minh, Ngày 11 Tháng 12 Năm 2025
Nhóm học viên thực hiện

Đại diện nhóm báo cáo
Học viên. Lê Nguyễn Nhật Quang

MỤC LỤC

Chương I. TỔNG QUAN

1.1. Bối cảnh và động lực nghiên cứu

Linear Algebra (Đại số tuyến tính) là một ngành nền tảng của toán học hiện đại, với vai trò trung tâm trong việc mô tả và xử lý các hệ tuyến tính, không gian vectơ và các phép biến đổi ma trận. Trong nhiều thập kỷ qua, vai trò của đại số tuyến tính ngày càng trở nên nổi bật khi các ngành khoa học dữ liệu, trí tuệ nhân tạo và tính toán hiệu năng cao phát triển mạnh mẽ. Những cấu trúc như phân tích ma trận, không gian vectơ, cơ sở, hạng, số đặc trưng và phân tích trị riêng đã trở thành nền tảng trong các hệ thống phân tích dữ liệu và mô hình học máy [1]. Các công trình về tính toán ma trận như *Matrix Computations* của Golub và Van Loan [2] khẳng định tầm quan trọng của việc xây dựng các thuật toán vừa ổn định về mặt số học vừa hiệu quả về mặt tính toán.

Nhu cầu ngày càng tăng trong việc xây dựng các mô hình hiệu quả hơn, ổn định hơn và có khả năng mở rộng tốt đã thúc đẩy nhiều nghiên cứu tập trung vào việc hiểu sâu bản chất tuyến tính của các phép biến đổi trong mạng nơ-ron. Cụ thể, các kỹ thuật như phân rã ma trận, chuẩn hóa, hoặc khai thác cấu trúc đại số của tham số mô hình đang được ứng dụng rộng rãi để tăng tốc huấn luyện và cải thiện năng lực mô hình [3].

Về mặt ứng dụng, Linear Algebra là trụ cột đối với các thuật toán học máy và phân tích dữ liệu, hồi quy tuyến tính, phân cụm dựa trên trị riêng và phương pháp bình phương tối thiểu. Các nghiên cứu và báo cáo ứng dụng gần đây trong khoa học dữ liệu cho thấy rằng các phép biến đổi và phân rã ma trận đóng vai trò quan trọng xuyên suốt từ tiền xử lý dữ liệu đến xây dựng mô hình dự đoán. Điều này tạo nên động lực mạnh mẽ cho việc nghiên cứu sâu hơn các thuật toán đại số tuyến tính, không chỉ ở khía cạnh lý thuyết mà còn về mặt triển khai thực nghiệm và hiệu năng tính toán.

Từ các phân tích trên, đề tài "Nghiên cứu các thuật toán và lý thuyết, thực tiễn trong Linear Algebra" được hình thành từ nhu cầu của cả lý thuyết và ứng dụng của các ngành khoa học dữ liệu hiện đại. Việc hiểu sâu bản chất toán học của các phép biến đổi tuyến tính giúp xây dựng nền tảng lý luận vững chắc cho các mô hình hiện đại.

1.2. Mục tiêu nghiên cứu

Mục tiêu của nghiên cứu này là phân tích và hệ thống hóa một cách toàn diện các khái niệm nền tảng của **đại số tuyến tính** – lĩnh vực toán học đóng vai trò cốt lõi trong việc xây dựng, mô tả và tối ưu hóa các mô hình học sâu hiện đại. Nghiên cứu tập trung khai thác khung lý thuyết nhằm làm rõ bản chất toán học, hành vi và ý nghĩa ứng dụng của các cấu trúc tuyến tính trong quá trình huấn

luyện mô hình. Qua đó, nghiên cứu hướng tới việc tạo dựng nền tảng lý thuyết vững chắc, hỗ trợ người học và nhà nghiên cứu hiểu sâu các thuật toán và kiến trúc trong học sâu. Cụ thể, nghiên cứu tập trung hướng đến các mục tiêu sau:

- **Phân tích các khái niệm cốt lõi** của đại số tuyến tính bao gồm vectơ, ma trận, chuẩn, tích vô hướng, tính độc lập tuyến tính và hạng ma trận, nhằm chỉ ra vai trò trung tâm của chúng trong việc biểu diễn dữ liệu và tham số mô hình trong học sâu.
- **Làm rõ bản chất của các phép biến đổi tuyến tính**, bao gồm nhân ma trận và ánh xạ tuyến tính, từ đó giải thích vì sao các kiến trúc mạng nơ-ron, từ perceptron đến mạng nhiều lớp, đều dựa trên các phép toán này để biến đổi và học biểu diễn.
- **Đánh giá vai trò của đại số tuyến tính trong các thuật toán tối ưu**, đặc biệt là trong tính toán gradient, truyền ngược, và các điều kiện hội tụ của mô hình, qua đó cho thấy mối liên hệ giữa cấu trúc đại số và hành vi huấn luyện.
- **Xác định các ưu điểm và hạn chế** của từng khái niệm đại số khi được triển khai trong môi trường học sâu, chẳng hạn như vấn đề số lớn — số nhỏ, điều kiện hóa ma trận, và tác động của chúng đến độ ổn định của quá trình tối ưu.
- **Khái quát hóa mối quan hệ giữa đại số tuyến tính và năng lực biểu diễn của mô hình**, giúp định hướng cách lựa chọn cấu trúc mạng, phép biến đổi và cách chuẩn hóa dữ liệu sao cho phù hợp với từng bài toán học sâu.

Thông qua các mục tiêu trên, nghiên cứu kỳ vọng mang đến một bức tranh nhất quán và có giá trị thực tiễn về vai trò thiết yếu của đại số tuyến tính trong học sâu, qua đó hỗ trợ người học xây dựng nền tảng toán học vững chắc, góp phần nâng cao năng lực triển khai và thiết kế các mô hình hiện đại.

1.3. Phạm vi và giới hạn

Các nội dung trọng tâm bao gồm:

- Khái niệm về vectơ và ma trận
- Các phép toán cơ bản (cộng, nhân, tích vô hướng, chuẩn)
- Biến đổi tuyến tính
- Hạng ma trận và tính độc lập tuyến tính

- Các cấu trúc đại số liên quan đến học sâu

Nghiên cứu **không mở rộng** sang các chủ đề cao cấp hơn như phân tích trị riêng, phân rã ma trận nâng cao, tối ưu hóa lồi hoặc các ứng dụng ngoài phạm vi học sâu. Bên cạnh đó, nghiên cứu tập trung vào khía cạnh lý thuyết và diễn giải, không nhằm mục tiêu triển khai thuật toán thực nghiệm quy mô lớn.

Chương II. KIẾN THỨC CƠ SỞ

2.1. Scalars

Trong đại số tuyến tính, scalar (vô hướng) là một giá trị số đơn lẻ, không mang thông tin về hướng hay cấu trúc nhiều chiều, mà chỉ biểu diễn độ lớn. Thông thường, khi nói về linear algebra trong machine learning, ta làm việc với các scalar là số thực, ký hiệu là $\mathbb{R}$. Khi đó, ta viết $a \in \mathbb{R}$ để chỉ một vô hướng a. Nếu cần làm việc với số phức, ta dùng $\mathbb{C}$ thay cho $\mathbb{R}$ [3], [4]. Scalar đóng vai trò là "hệ số" trong các phép biến đổi tuyến tính: ta dùng chúng để nhân với vector, ma trận, hay tensor nhằm phóng to, thu nhỏ, hoặc đổi dấu các đối tượng đó. Chẳng hạn, nếu $\alpha \in \mathbb{R}$ là một scalar và x là một vector, thì αx là vector được nhân đều từng phần tử bởi α. Về mặt ký hiệu, scalar thường được viết bằng chữ thường như $a, b, c, \lambda, \alpha, \beta$, còn các tập hợp scalar được viết bằng chữ in hoa kép như $\mathbb{R}$(real numbers), $\mathbb{Z}$(integers) [3]. Trong học máy, các đại lượng vô hướng đóng vai trò quan trọng trong việc tinh chỉnh mô hình, đo lường hiệu suất và với các giá trị cơ bản để xây dựng vectơ, ma trận và tensor bậc cao hơn. Trong học máy có thể kể đến là tốc độ học (Learning rate), độ lệch đơn vị (Bias values), hệ số điều chuẩn (Regularization coefficients) và giá trị tổn thất riêng lẻ (Loss values) [1].

2.2. Vectors

Vector là một dãy có thứ tự các scalar, có thể hiểu như một "cột số liệu" hoặc một điểm trong không gian nhiều chiều. Nếu ta có n thành phần, một vector thường được viết dưới dạng cột như

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^n,$$

trong đó mỗi x_i là một scalar [3], và $\mathbb{R}^n$ là không gian vector n-chiều các dãy số thực độ dài n [3], [4]. Về mặt hình học, x có thể được xem là một mũi tên xuất phát từ gốc tọa độ tới điểm $(x_1, \dots, x_n)$. Trong bối cảnh dữ liệu, mỗi vector thường biểu diễn một mẫu dữ liệu: ví dụ, một bệnh nhân có thể được mô tả bởi vector $x =$ (tuổi, chiều cao, cân nặng, ...). Các phép toán cơ bản trên vector đều được định nghĩa theo từng thành phần. Nếu $x, y \in \mathbb{R}^n$, ta có

$$x + y = \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \vdots \\ x_n + y_n \end{bmatrix}, \alpha x = \begin{bmatrix} \alpha x_1 \\ \alpha x_2 \\ \vdots \\ \alpha x_n \end{bmatrix}$$

với $\alpha \in \mathbb{R}$ là một scalar. Chính các phép cộng vector và nhân scalar này là nền tảng để định nghĩa cấu trúc không gian vector và các khái niệm như tổ hợp tuyến tính, cơ sở, hay hạng (rank) sau này [1], [6].

2.3. Matrices

Ma trận là một mảng chữ nhật gồm các scalar, sắp xếp theo hàng và cột. Nếu một ma trận có m hàng và n cột, ta viết $A \in \mathbb{R}^{m \times n}$ và biểu diễn dưới dạng

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix},$$

Trong đó a_{ij} là phần tử nằm ở hàng i, cột j [3], [4]. Ta có thể nhìn Anhư một tập các vector cột (mỗi cột là một vector trong $\mathbb{R}^m$) hoặc tập các vector hàng (mỗi hàng là một vector trong $\mathbb{R}^n$). Trong machine learning, ma trận thường được dùng để lưu cả một batch dữ liệu: mỗi hàng là một mẫu, mỗi cột là một đặc trưng (feature) [1], [6]. Ở góc nhìn biến đổi tuyến tính, một ma trận $A \in \mathbb{R}^{m \times n}$ mô tả một ánh xạ tuyến tính từ $\mathbb{R}^n$ sang $\mathbb{R}^m$, gửi một vector đầu vào $x \in \mathbb{R}^n$ tới vector đầu ra $y = Ax \in \mathbb{R}^m$ [3], [4]. Một số ma trận đặc biệt thường gặp gồm có ma trận vuông (khi $m = n$), ma trận đường chéo (chỉ có phần tử khác 0 trên đường chéo chính), và ma trận đơn vị $I_n \in \mathbb{R}^{n \times n}$ với

$$\left(I_n \right)_{ij} = \begin{cases} 1, & \text{nếu } i = j, \\ 0, & \text{nếu } i \neq j, \end{cases}$$

đóng vai trò "phần tử trung hòa" cho phép nhân ma trận vì $I_n x = x$ và $I_n A = A$ khi kích thước tương thích [3].

Trong học máy, ma trận đóng vai trò là cấu trúc dữ liệu trung tâm để biểu diễn và xử lý thông tin. Một tập dữ liệu thông thường được biểu diễn dưới dạng ma trận, trong đó mỗi hàng tương ứng với một mẫu dữ liệu và mỗi cột đại diện cho một đặc trưng. Cách tổ chức này cho phép các thuật toán khai thác các phép toán đại số tuyến tính để xử lý đồng thời nhiều mẫu dữ liệu một cách hiệu quả [1], [3], [6]. Trong mạng nơ-ron, các lớp tuyến tính (fully connected layers) cũng được mô tả thông qua các phép nhân ma trận: đầu vào là một vectơ x, trọng số của lớp được lưu trữ trong một ma trận W, và đầu ra của lớp được tính bằng phép nhân $W^\top x$. Khi nhiều lớp được kết nối tuần tự, quá trình lan truyền thông tin qua mạng nơ-ron thực chất là chuỗi các phép nhân ma trận và biến đổi tuyến tính liên tiếp [1], [4]. Do đó, đại số tuyến tính, đặc biệt là các thao tác với ma trận đã trở

thành nền tảng toán học quan trọng trong việc xây dựng, huấn luyện và tối ưu hóa các mô hình học máy hiện đại [1], [4], [6].

2.4. Tensors

Tensor là sự khái quát hóa của scalar, vector và ma trận sang số chiều cao hơn. Thay vì chỉ làm việc với một số đơn lẻ (scalar), một dãy một chiều (vector) hay một bảng hai chiều (ma trận), tensor cho phép ta mô tả dữ liệu trong không gian nhiều chiều hơn một cách có hệ thống. Về mặt bậc, ta có thể coi scalar là tensor bậc 0 (0-order tensor), vector là tensor bậc 1 (1st-order tensor), ma trận là tensor bậc 2 (2nd-order tensor), và khi số chiều tăng lên ta thu được các tensor bậc cao hơn (3rd-order, 4th-order, …) [1], [6]. Nói chung, một tensor bậc k(hay bậc N) có shape

$$\text{shape} = (d_1, d_2, \dots, d_k),$$

trong đó $d_i$là kích thước trên trục thứ i. Nếu X là một tensor bậc k với phần tử là số thực, ta ký hiệu

$$X \in \mathbb{R}^{d_1 \times d_2 \times \cdots \times d_k},$$

và một phần tử cụ thể được đánh chỉ số

$$x_{i_1 i_2 \dots i_k}, 1 \le i_j \le d_j \text{ với mọi } j = 1, \dots, k.$$

Hình dung trực quan, tensor bậc 3 là một "khối" 3D các số; tensor bậc 4 có thể được xem như một tập các khối 3D xếp chồng theo một trục nữa, và cứ thế mở rộng [4], [6].

Trong học máy và đặc biệt là học sâu, tensor xuất hiện khắp nơi vì hầu như mọi dạng dữ liệu và tham số mô hình đều được biểu diễn dưới dạng tensor đa chiều. Một ảnh màu thường được biểu diễn bởi tensor 3 chiều với các trục chiều cao, chiều rộng và số kênh (height×width×channels). Khi làm việc với nhiều ảnh đồng thời trong quá trình huấn luyện, ta lại dùng tensor 4 chiều để lưu trữ một batch dữ liệu, ví dụ shape (batch, channels, height, width) hoặc (batch, height, width, channels) tùy framework [1], [6]. Với dữ liệu chuỗi hoặc thời gian như âm thanh, tín hiệu cảm biến hay chuỗi embedding của văn bản, biểu diễn phổ biến là tensor 3 chiều (batch,time,features) [1]. Các mạng nơ-ron tích chập (CNN) hoạt động trực tiếp trên những cấu trúc này thông qua các phép tích chập đa chiều; các mô hình xử lý chuỗi (RNN, Transformer) cũng coi mỗi mini-batch chuỗi như một tensor bậc cao thay vì từng vector riêng lẻ [1], [6].

Không chỉ dữ liệu, các tham số của mô hình trong deep learning cũng là tensor: trọng số của lớp tích chập là các tensor nhiều chiều (ví dụ (out_channels, in_channels, kernel_height, kernel_width)), ma trận embedding là tensor 2 chiều (vocab_size,embedding_dim), còn các bộ lọc, kernel, weight trong nhiều kiến trúc khác đều có thể được xem là tensor với bậc phù hợp. Việc trừu tượng hóa scalar, vector, ma trận thành tensor giúp ta có một framework thống nhất để mô tả dữ liệu nhiều chiều, các phép toán và kiến trúc mạng nơ-ron, đồng thời tận dụng tốt khả năng song song hóa của GPU/TPU, vốn được thiết kế để thao tác trên các tensor lớn một cách hiệu quả [1], [6].

2.5. Các tính chất cơ bản của phép toán trên tensor

Các phép toán cơ bản trên tensor (cộng, trừ, nhân với vô hướng) được định nghĩa theo từng phần tử và thừa hưởng những tính chất quen thuộc từ số thực và vector [2], [3], [4]. Giả sử ta có hai tensor cùng shape $\mathcal{X}, \mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_k}$. Phép cộng tensor được định nghĩa bằng cách cộng từng phần tử tương ứng:

$$(\mathcal{X} + \mathcal{Y})_{i_1 \ldots i_k} = x_{i_1 \ldots i_k} + y_{i_1 \ldots i_k}.$$

Tương tự, nếu $\alpha \in \mathbb{R}$ là một scalar, phép nhân scalar–tensor là

$$(\alpha \mathcal{X})_{i_1 \ldots i_k} = \alpha\, x_{i_1 \ldots i_k}.$$

Các phép toán này tuân theo những tính chất đại số cơ bản như giao hoán $(\mathcal{X} + \mathcal{Y} = \mathcal{Y} + \mathcal{X})$, kết hợp $((\mathcal{X} + \mathcal{Y}) + \mathcal{Z} = \mathcal{X} + (\mathcal{Y} + \mathcal{Z}))$, và phân phối $\alpha(\mathcal{X} + \mathcal{Y}) = \alpha\mathcal{X} + \alpha\mathcal{Y}$ [3], [4]. Trong các thư viện tính toán như NumPy, PyTorch, MXNet, các phép toán này còn đi kèm với cơ chế broadcasting, cho phép cộng hoặc nhân các tensor có shape khác nhau nhưng "tương thích", ví dụ cộng một tensor với một scalar, hoặc với một vector được lặp lại theo một chiều [1], [6]. Điều này giúp việc biểu diễn các phép toán trên nhiều mẫu (batch) trở nên gọn gàng, đồng thời vẫn bám sát các quy tắc đại số trên từng phần tử.

2.6. Reduction (Phép rút gọn)

Reduction là nhóm các phép toán biến một tensor thành một đối tượng có ít chiều hơn (thậm chí là một scalar), bằng cách "gộp" các giá trị dọc theo một hoặc nhiều trục (axis). Các phép reduction phổ biến bao gồm tổng (sum), trung bình (mean), giá trị lớn nhất (max), nhỏ nhất (min),… Nếu $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_k}$, ta có thể lấy tổng trên toàn bộ các phần tử:

$$sum(\mathcal{X}) = \sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} \cdots \sum_{i_k=1}^{n_k} x_{i_1 \ldots i_k},$$

kết quả là một scalar [2], [3]. Ta cũng có thể reduce theo một trục cụ thể. Ví dụ, với ma trận $A \in \mathbb{R}^{m \times n}$, tổng theo hàng (giữ lại chiều cột) được viết:

$$s_j = \sum_{i=1}^{m} a_{ij}, j = 1, \ldots, n,$$

và khi đó $s \in \mathbb{R}^n$. Tương tự, nếu ta reduce theo trục cột, ta thu được vector độ dài m. Trong bối cảnh học máy, reduction thường được dùng để tổng hợp thông tin: tính tổng loss trên batch, tính trung bình xác suất, chuẩn hóa dữ liệu theo feature, hay thu được một vector đặc trưng gọn hơn

từ một tensor nhiều chiều [1], [6]. Các hàm như sum(axis=...), mean(axis=...), max(axis=...) trong code chính là các phép reduction được định nghĩa một cách tổng quát trên tensor [6].

2.7. Non-Reduction Sum (Tổng không rút gọn chiều)

Khác với reduction (làm giảm số chiều), non-reduction sum là thao tác cộng các tensor mà vẫn giữ nguyên shape đầu ra. Về bản chất, đây chính là phép cộng theo từng phần tử (element-wise) giữa các tensor có cùng kích thước, hoặc các tensor tương thích dưới cơ chế broadcasting [1], [6]. Giả sử ta có $k$tensor $\mathcal{X}^{(1)}, \dots, \mathcal{X}^{(k)} \in \mathbb{R}^{n_1 \times \cdots \times n_m}$ cùng shape. Ta định nghĩa tổng không rút gọn:

$$\mathcal{S} = \sum_{t=1}^{k} \mathcal{X}^{(t)}, \mathcal{S} \in \mathbb{R}^{n_1 \times \cdots \times n_m},$$

với từng phần tử

$$s_{i_1 \dots i_m} = \sum_{t=1}^{k} x^{(t)}_{i_1 \dots i_m}.$$

Kết quả $\mathcal{S}$vẫn là một tensor m-chiều với cùng shape như các tensor ban đầu, tức là ta "gộp" thông tin từ nhiều nguồn nhưng không làm mất chiều nào cả. Trong thực hành, non-reduction sum xuất hiện rất nhiều: ta cộng output của nhiều lớp (residual connection trong ResNet), cộng embedding của nhiều feature, hoặc cộng từng dự đoán để lấy tổng dự đoán theo từng phần tử [1]. Khi thư viện như NumPy/PyTorch nói A + B, nếu A và B có shape tương thích, đó chính là một non-reduction sum: mỗi phần tử của kết quả là tổng của các phần tử tương ứng, nhưng cấu trúc hình học (shape) của đối tượng vẫn được bảo toàn.

2.8. Dot Products

Dot product (tích vô hướng, tích trong) là một phép toán lấy hai vector và trả về một scalar, đo mức "gần gũi" giữa chúng [2], [3], [4]. Với hai vector $x, y \in \mathbb{R}^n$, tích vô hướng được định nghĩa là

$$x \cdot y = \sum_{i=1}^{n} x_i \, y_i.$$

Nếu ta viết x dưới dạng vector hàng còn y là vector cột, thì dot product chính là phép nhân ma trận $1 \times n$với $n \times 1$:

$$x^\top y = [x_1 \quad x_2 \quad \dots \quad x_n] \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^{n} x_i\, y_i.$$

Về mặt hình học, dot product còn liên hệ với góc giữa hai vector. Nếu θlà góc giữa x và y, ta có

$$x \cdot y = \| x \| \ \ \| y \| \cos\ \theta$$

Trong đó $\| x \|$là norm (độ dài) của x [3], [4]. Từ đó, nếu dot product dương thì góc giữa hai vector nhỏ hơn 90°(chúng "cùng hướng"), nếu bằng 0 thì hai vector vuông góc (orthogonal), nếu âm thì chúng "ngược hướng". Trong machine learning, dot product là thành phần cốt lõi của rất nhiều thuật toán: từ logistic regression, linear regression, tới attention trong transformer, cosine similarity,… [1], [6] Tất cả đều dựa trên việc nhân các vector rồi cộng các tích phần tử tương ứng – chính là dot product.

2.9. Matrix – Vector Products (Tích Ma Trận – Vector)

Tích ma trận–vector mô tả cách một ma trận tác động lên một vector để tạo ra một vector mới [2], [3], [4]. Cho ma trận $A \in \mathbb{R}^{m\times n}$ và vector cột $x \in \mathbb{R}^n$. Tích $y = Ax$ là một vector cột $y \in \mathbb{R}^m$. Nếu ký hiệu dòng thứ i của A là $a_i^\top$, thì phần tử thứ i của y chính là tích vô hướng giữa dòng đó và x:

$$y_i = a_i^\top x = \sum_{j=1}^{n} a_{ij}\, x_j, i = 1, \dots, m.$$

Như vậy, tích giữa một ma trận A kích thước $m \times n$ và một vector n-chiều tạo ra một vector cột dài m, trong đó phần tử thứ i là tích vô hướng $a_i^\top x$. Ta có thể coi phép nhân với ma trận Alà một phép biến đổi tuyến tính

$$A\colon \mathbb{R}^n \to \mathbb{R}^m, x \mapsto Ax,$$

tức là một phép chiếu hoặc biến đổi các vector từ không gian $\mathbb{R}^n$ sang không gian $\mathbb{R}^m$.

Nhiều phép biến đổi hình học quen thuộc có thể được biểu diễn dưới dạng tích ma trận–vector. Ví dụ, phép quay trong mặt phẳng một góc θ quanh gốc tọa độ được mô tả bởi ma trận quay

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\ \theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$

Với một điểm $x = (x, y)^\top$, tọa độ sau khi quay là $x' = (x', y')^\top$ thỏa

$$x' = R(\theta)x = \begin{bmatrix} cos\,\theta & -sin\ \theta \\ sin\,\theta & cos\,\theta \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}.$$

Ở đây, mỗi vector trong $\mathbb{R}^2$ được ánh xạ sang một vector khác trong $\mathbb{R}^2$ bằng cùng một ma trận quay.

Tích ma trận–vector cũng chính là phép tính then chốt khi ta tính đầu ra của mỗi lớp tuyến tính trong mạng nơ-ron dựa trên đầu ra của lớp trước đó [1], [6]. Nếu $x \in \mathbb{R}^n$ là vector đầu vào của lớp, $W \in \mathbb{R}^{m\times n}$ là ma trận trọng số và $b \in \mathbb{R}^m$ là vector bias, thì đầu ra $z \in \mathbb{R}^m$ được tính bằng

$$z = Wx + b.$$

Ví dụ, với một lớp có 4 đầu vào và 2 neuron, ta có

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}, z = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = Wx + b.$$

Mỗi phần tử z_i ở đây lại là một tích vô hướng giữa dòng thứ i của W với vector x, cộng thêm bias tương ứng [1].

Lưu ý rằng kích thước cột của ma trận A (số cột, hay độ dài dọc theo trục 1) phải bằng với độ dài của vector x; nếu không, tích Ax không được định nghĩa [2], [3]. Trong code, tích ma trận–vector thường được viết ngắn gọn bằng toán tử nhân ma trận, chẳng hạn như A @ x trong NumPy, hoặc torch.mv(A, x) hay cũng chỉ đơn giản là A @ x trong PyTorch, miễn là kích thước hai toán hạng tương thích như trên.

2.10. Matrix – Matrix Multiplication

Matrix–matrix multiplication mô tả cách kết hợp hai phép biến đổi tuyến tính liên tiếp thành một phép biến đổi tuyến tính duy nhất [2], [3], [4]. Cho hai ma trận $A \in \mathbb{R}^{m\times n}$và $B \in \mathbb{R}^{n\times p}$. Khi đó tích $C = AB$ được định nghĩa là một ma trận $C \in \mathbb{R}^{m\times p}$sao cho mỗi phần tử c_{ij} là dot product giữa hàng thứ i của A và cột thứ j của B:

$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj}, i = 1, \dots, m,\ j = 1, \dots, p.$$

Viết dạng ma trận, ta có

$$C = AB = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}\begin{bmatrix} b_{11} & \dots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{np} \end{bmatrix}.$$

Một cách nhìn quan trọng là "AB" tương đương với việc trước hết áp dụng phép biến đổi tuyến tính B lên vector đầu vào, rồi sau đó áp dụng tiếp A. Nếu xem $B: \mathbb{R}^p \to \mathbb{R}^n$ và $A: \mathbb{R}^n \to \mathbb{R}^m$, thì tích AB chính là ánh xạ hợp $A \circ B: \mathbb{R}^p \to \mathbb{R}^m$. Từ đó, matrix–matrix multiplication có tính kết hợp: $(AB)C = A(BC)$ khi kích thước tương thích, nhưng không giao hoán nói chung: $AB \neq BA$ trong đa số trường hợp [3], [4].

Trong triển khai thực tế của mạng nơ-ron, khi ta xử lý một batch gồm bmẫu dữ liệu, ta thường gom chúng thành một ma trận input $X \in \mathbb{R}^{b \times n}$ (mỗi hàng là một vector đặc trưng), rồi nhân với ma trận trọng số $W \in \mathbb{R}^{n \times m}$ để thu được ma trận output $Z = XW \in \mathbb{R}^{b \times m}$ [1], [6]. Ở đây, mỗi hàng của Z tương ứng với kết quả của phép nhân ma trận–vector cho một mẫu riêng lẻ, và toàn bộ phép tính được gộp thành một matrix–matrix multiplication duy nhất để tận dụng tối đa khả năng song song của phần cứng.

2.11. Norms

Norm là một hàm đo "độ lớn" hoặc "độ dài" của một vector (hoặc ma trận), thỏa mãn một số tính chất toán học nhất định, tương tự như cách ta đo độ dài của một đoạn thẳng trong hình học [2], [3]. Một norm trên không gian vector V là một ánh xạ $\|\cdot\|: V \to \mathbb{R}_{\geq 0}$ sao cho, với mọi vector $x, y \in V$và mọi scalar α, ta có:

(1) $\| x \| \geq 0$và $\| x \| = 0$ khi và chỉ khi $x = 0$ (không âm và tách điểm)

(2) $\| \alpha x \| = | \alpha | \ \| x \|$(đồng nhất bậc 1 theo vô hướng)

(3) $\| x + y \| \leq \| x \| + \| y \|$(bất đẳng thức tam giác).

Đối với vector $x = (x_1, \dots, x_n)^\top \in \mathbb{R}^n$, một lớp norm quan trọng là p-norm:

$$\| x \|_p = \left(\sum_{i=1}^{n} | x_i |^p \right)^{1/p}, p \geq 1.$$

Có các trường hợp Norms đặc biệt thường dùng bao gồm:

1. **Norm ℓ_1 (khoảng cách Manhattan)**

Norm ℓ_1 đo tổng độ lệch tuyệt đối của các thành phần trong vector, giống như khi ta đi trong thành phố kiểu Manhattan: chỉ được đi theo hai trục ngang và dọc, mỗi bước là một "block" [6]. Ý nghĩa của nó là đo tổng "độ lệch" trên từng chiều, nên phù hợp khi ta quan tâm đến độ chênh lệch tuyến tính và muốn bớt nhạy với bình phương sai số. Trong bài toán tối ưu, norm $\ell_1$thường được dùng để khuyến khích nghiệm thưa (nhiều thành phần bằng 0), ví dụ trong L1-regularization hay Lasso, vì "quả bóng" $\{x: \| x \|_1 \leq 1\}$ có các góc nhọn trên trục tọa độ khiến nghiệm tối ưu dễ rơi vào các điểm có nhiều tọa độ bằng 0 [3], [4]. Với $x = (x_1, \dots, x_n)^\top$, ta có công thức tính ℓ_1-norm:

$$\| x \|_1 = \sum_{i=1}^{n} | x_i |.$$

2. Norm ℓ_2 (khoảng cách Euclid)

Norm ℓ_2 đo độ dài "thẳng" của vector từ gốc tọa độ tới điểm đó, chính là khoảng cách Euclid quen thuộc trong hình học [6]. Nó khuếch đại các sai số lớn (do dùng bình phương) nên thường được dùng khi ta muốn phạt mạnh các thành phần có giá trị quá lớn. Norm ℓ_2 cũng có nhiều tính chất tốt: trơn, dễ lấy đạo hàm, bất biến dưới các phép quay trực chuẩn (nếu Q trực chuẩn thì $\| Qx \|_2 = \| x \|_2$). Trong machine learning, ℓ_2-norm được dùng để đo độ lớn của vector trọng số, đo khoảng cách $\| x - y \|_2$ giữa hai điểm, và làm L2-regularization (weight decay) để "kéo" trọng số về gần 0 nhưng hiếm khi đẩy chúng về đúng 0 [1], [3]. Công thức của Norm ℓ_2 là:

$$\| x \|_2 = \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2}.$$

3. Frobenius norm (cho ma trận)

Frobenius norm mở rộng ý tưởng norm ℓ_2 từ vector sang ma trận: ta gom tất cả phần tử của ma trận thành một vector dài, rồi đo norm ℓ_2 của vector đó [6]. Ý nghĩa của nó là đo "kích thước tổng thể" của ma trận, xem tổng bình phương mọi phần tử lớn đến mức nào. Vì vậy, Frobenius norm là thước đo tự nhiên cho sự khác biệt giữa hai ma trận $\| A - B \|_F$, và thường được dùng để regularization cho toàn bộ ma trận trọng số của một lớp trong mạng nơ-ron, giúp giới hạn độ lớn chung của các trọng số [1]. Với $A \in \mathbb{R}^{m \times n}$, ta có:

$$\| A \|_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2 \right)^{1/2}.$$

Chương III. GIẢI THUẬT ĐẠI SỐ TUYẾN TÍNH

1. Prove that the transpose of the transpose of a matrix is the matrix itself: $(A^\top)^\top = A$.

Ta xét ma trận $A \in \mathbb{R}^{m \times n}$ với phần tử hàng i, cột j là a_{ij}. Chuyển vị của A, ký hiệu $A^\top \in \mathbb{R}^{n \times m}$, có phần tử

$$(A^\top)_{ij} = a_{ji}.$$

Bây giờ lấy chuyển vị lần nữa, ta được

$$((A^\top)^\top)_{ij} = (A^\top)_{ji} = a_{ij}.$$

Vì điều này đúng với mọi cặp chỉ số i, j, nên ma trận $(A^\top)^\top$ có đúng cùng các phần tử như A. Do đó

$$(A^\top)^\top = A.$$

2. **Given two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, show that sum and transposition commute: $\boldsymbol{A}^\top + \boldsymbol{B}^\top = (\boldsymbol{A} + \boldsymbol{B})^\top$.**

Giả sử $A, B \in \mathbb{R}^{m \times n}$, với phần tử tương ứng là a_{ij}, b_{ij}. Ma trận tổng $A + B$ có phần tử

$$(A + B)_{ij} = a_{ij} + b_{ij}.$$

Chuyển vị tổng:

$$\big((A + B)^\top\big)_{ij} = (A + B)_{ji} = a_{ji} + b_{ji}.$$

Mặt khác,

$$\big(A^\top + B^\top\big)_{ij} = (A^\top)_{ij} + (B^\top)_{ij} = a_{ji} + b_{ji}.$$

Hai biểu thức cho cùng một phần tử trùng nhau với mọi i, j, vì vậy

$$(A + B)^\top = A^\top + B^\top.$$

3. **Given any square matrix $\boldsymbol{A}$, is $\boldsymbol{A} + \boldsymbol{A}^\top$ always symmetric? Can you prove the result using only the results of the previous two exercises?**

Xét ma trận vuông $A \in \mathbb{R}^{n \times n}$ và đặt $S = A + A^\top$. Ta cần chứng minh S là đối xứng, tức là $S^\top = S$. Ta tính

$$S^\top = (A + A^\top)^\top.$$

Dùng kết quả câu 2, chuyển vị của tổng bằng tổng các chuyển vị:

$$S^\top = A^\top + (A^\top)^\top.$$

Dùng kết quả câu 1, $(A^\top)^\top = A$, nên

$$S^\top = A^\top + A = A + A^\top = S.$$

Vậy $A + A^\top$ luôn là ma trận đối xứng với mọi ma trận vuông A.

4. **We defined the tensor $\boldsymbol{X}$ of shape (2,3,4) in this section. What is the output of len$(\boldsymbol{X})$?**

Trong các thư viện như NumPy, PyTorch, khi áp dụng len(·) lên một tensor nhiều chiều, kết quả là độ dài của trục đầu tiên (axis 0). Với shape (2,3,4), trục 0 có độ dài 2. Do đó

$$\text{len}(X) = 2.$$

5. For a tensor X of arbitrary shape, does len(X) always correspond to the length of a certain axis of X? What is that axis?

Giả sử X có shape $(n_1, n_2, \dots, n_k)$. Việc gọi len(·) cho ta số "phần tử ở cấp trên cùng" – tức là số "khối con" khi ta duyệt "for item in X". Mỗi khối này tương ứng với một lát cắt dọc theo trục đầu tiên. Vì vậy $\text{len}(X) = n_1$ chính là độ dài của axis 0 (trục đầu tiên).

6. Run A/A.sum(axis=1) and see what happens. Can you analyze the results?

Xét ma trận $A \in \mathbb{R}^{m \times n}$ với phần tử a_{ij}. Tổng theo hàng là

$$s_i = \sum_{j=1}^{n} a_{ij}, i = 1, \dots, m.$$

Vector $s = (s_1, \dots, s_m)^\top$ tương ứng với A.sum(axis=1). Khi ta tính

$$B = \frac{A}{A.\text{sum(axis=1)}}$$

trong các thư viện có broadcasting, mỗi hàng thứ i của A được chia cho cùng một scalar s_i. Phần tử của B là

$$b_{ij} = \frac{a_{ij}}{s_i}$$

với giả thiết $s_i \neq 0$. Khi đó tổng trên hàng i của B là

$$\sum_{j=1}^{n} b_{ij} = \sum_{j=1}^{n} \frac{a_{ij}}{s_i} = \frac{1}{s_i} \sum_{j=1}^{n} a_{ij} = \frac{s_i}{s_i} = 1.$$

Như vậy, phép toán này chuẩn hóa từng hàng của A sao cho mỗi hàng có tổng bằng 1 (nếu các phần tử không âm, mỗi hàng trở thành một phân phối xác suất).

7. **When traveling between two points in downtown Manhattan, what is the distance that you need to cover in terms of the coordinates, i.e., in terms of avenues and streets? Can you travel diagonally?**

Giả sử hệ trục tọa độ biểu diễn avenues và streets, điểm xuất phát là (x_1, y_1) và điểm đích là (x_2, y_2). Trên lưới Manhattan, ta chỉ được đi song song với các trục (xuống–lên, trái–phải), không được đi chéo xuyên qua block. Để tới điểm đích, ta phải đi $| x_2 - x_1 |$ block theo phương x và $| y_2 - y_1 |$ block theo phương y. Tổng số block phải đi là

$$d_{\text{Manhattan}}((x_1, y_1), (x_2, y_2)) = | x_2 - x_1 | + | y_2 - y_1 |.$$

Vì không được phép đi chéo, ta không dùng khoảng cách Euclid $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$mà phải dùng khoảng cách Manhattan ở trên; và câu trả lời cho "Can you travel diagonally?" là không.

8. **Consider a tensor of shape $(2, 3, 4)$. What are the shapes of the summation outputs along axes 0, 1, and 2?**

Tensor X có shape $(2,3,4)$, tức là:

- trục 0: độ dài 2
- trục 1: độ dài 3
- trục 2: độ dài 4

Khi lấy tổng dọc theo một trục (và không dùng keepdims=True), ta "xóa" trục đó.

- Tổng theo axis 0: cộng 2 lát 3×4 với nhau, còn lại trục 1 và 2, nên shape là $(3,4)$.
- Tổng theo axis 1: cộng 3 phần tử theo trục 1, với mỗi cặp (trục 0, trục 2) ta có một giá trị, nên shape là $(2,4)$.
- Tổng theo axis 2: cộng 4 phần tử theo trục 2, còn lại trục 0 và 1, nên shape là $(2,3)$.

Tóm lại:

$$\begin{aligned} \text{sum}(X, \text{axis} = 0) &\in \mathbb{R}^{3\times4}, \\ \text{sum}(X, \text{axis} = 1) &\in \mathbb{R}^{2\times4}, \\ \text{sum}(X, \text{axis} = 2) &\in \mathbb{R}^{2\times3}. \end{aligned}$$

9. **Feed a tensor with three or more axes to the linalg.norm function and observe its output. What does this function compute for tensors of arbitrary shape?**

Khi không chỉ định trục (axis), các hàm như linalg.norm xem toàn bộ phần tử của tensor như một vector 1 chiều, rồi tính chuẩn Euclid (ℓ_2-norm) của vector đó. Nếu X có shape $(n_1, \ldots, n_k)$ với phần tử $x_{i_1 \ldots i_k}$, ta có

$$\text{linalg.norm}(X) = \sqrt{\sum_{i_1=1}^{n_1} \cdots \sum_{i_k=1}^{n_k} x_{i_1 \ldots i_k}^2}.$$

Khi X là ma trận, đây chính là Frobenius norm; nói chung, đây là ℓ_2-norm của toàn bộ tensor khi "trải phẳng" nó thành một vector.

10. Consider three large matrices, say $\boldsymbol{A} \in \mathbb{R}^{2^{10} \times 2^{16}}$, $\boldsymbol{B} \in \mathbb{R}^{2^{16} \times 2^{5}}$ and $\boldsymbol{C} \in \mathbb{R}^{2^{5} \times 2^{14}}$, initialized with Gaussian random variables. You want to compute the product $\boldsymbol{ABC}$. Is there any difference in memory footprint and speed, depending on whether you compute $(\boldsymbol{AB})\boldsymbol{C}$ or $\boldsymbol{A}(\boldsymbol{BC})$? Why?

Để so sánh hai cách nhân $(AB)C$ và $A(BC)$, ta dùng quy tắc: nhân một ma trận kích thước $m \times n$ với một ma trận kích thước $n \times p$ tốn khoảng mnp phép nhân–cộng và tạo ra một ma trận trung gian kích thước $m \times p$. Các ma trận ở đây có kích thước rất lớn vì đều là lũy thừa của 2, nên khác biệt về bậc sẽ ảnh hưởng mạnh đến cả thời gian chạy lẫn bộ nhớ.

Trước hết xét cách nhân $(AB)C$. Bước 1, ta nhân A (kích thước $2^{10} \times 2^{16}$) với B (kích thước $2^{16} \times 2^{5}$). Kết quả AB có kích thước $2^{10} \times 2^{5}$. Số phép tính cần cho bước này là:

$$2^{10} \cdot 2^{16} \cdot 2^{5} = 2^{31}.$$

Ma trận trung gian AB chứa $2^{10+5} = 2^{15}$ phần tử, nên dung lượng nhớ để chứa nó vẫn còn tương đối nhỏ so với kích thước các ma trận ban đầu. Bước 2, ta nhân ma trận trung gian AB (kích thước $2^{10} \times 2^{5}$) với C (kích thước $2^{5} \times 2^{14}$) để thu được ABC có kích thước $2^{10} \times 2^{14}$. Chi phí tính toán của bước này là:

$$2^{10} \cdot 2^{5} \cdot 2^{14} = 2^{29}.$$

Tổng số phép tính cho cả hai bước xấp xỉ $2^{31} + 2^{29}$, và ma trận trung gian lớn nhất chỉ có 2^{15} phần tử. Như vậy cả chi phí tính toán lẫn bộ nhớ trung gian đều ở mức "vừa phải" so với kích thước bài toán.

Bây giờ xét cách nhân $A(BC)$. Bước 1, ta nhân B (kích thước $2^{16} \times 2^{5}$) với C (kích thước $2^{5} \times 2^{14}$). Kết quả BC có kích thước $2^{16} \times 2^{14}$. Số phép tính cần cho bước này là:

$$2^{16} \cdot 2^{5} \cdot 2^{14} = 2^{35}.$$

Ma trận trung gian BC lúc này có $2^{16+14} = 2^{30}$ phần tử, tức là lớn hơn ma trận trung gian AB ở cách trước tới $2^{30}/2^{15} = 2^{15}$ lần. Đây là một ma trận khổng lồ, đòi hỏi lượng bộ nhớ rất lớn. Bước 2, ta nhân A (kích thước $2^{10} \times 2^{16}$) với BC (kích thước $2^{16} \times 2^{14}$) để thu được ABC (kích thước $2^{10} \times 2^{14}$). Chi phí tính toán cho bước này là:

$$2^{10} \cdot 2^{16} \cdot 2^{14} = 2^{40}.$$

Tổng số phép tính là khoảng $2^{35} + 2^{40}$, lớn hơn rất nhiều so với $2^{31} + 2^{29}$ ở cách $(AB)C$. Đồng thời, trong suốt quá trình, ta phải giữ một ma trận trung gian 2^{30} phần tử trong bộ nhớ, tốn gấp hàng chục nghìn lần so với ma trận trung gian 2^{15} phần tử ở cách đầu tiên.

Từ phân tích trên, ta thấy mặc dù phép nhân ma trận là kết hợp về mặt đại số (tức là $(AB)C = A(BC)$ về kết quả), nhưng thứ tự nhân lại có ảnh hưởng cực lớn đến chi phí tính toán và bộ nhớ. Trong ví dụ này, nhân theo cách $(AB)C$ vừa dùng ít phép tính hơn (giảm từ bậc 2^{40}xuống bậc 2^{31}) vừa cần ma trận trung gian rất nhỏ so với cách $A(BC)$. Vì vậy, cách nhân $(AB)C$ rõ ràng tiết kiệm cả thời gian lẫn bộ nhớ hơn hẳn so với nhân theo thứ tự $A(BC)$.

11. Consider three large matrices, say $\boldsymbol{A} \in \mathbb{R}^{2^{10} \times 2^{16}}$, $\boldsymbol{B} \in \mathbb{R}^{2^{16} \times 2^{5}}$ and $\boldsymbol{C} \in \mathbb{R}^{2^{5} \times 2^{16}}$. Is there any difference in speed depending on whether you compute $\boldsymbol{AB}$ or $\boldsymbol{AC}^{\top}$? Why? What changes if you initialize $\boldsymbol{C} = \boldsymbol{B}^{\top}$without cloning memory? Why?

Trước hết, ta phân tích kích thước và số phép tính cho hai phép nhân. Ma trận A có kích thước $2^{10} \times 2^{16}$, ma trận B có kích thước $2^{16} \times 2^{5}$. Khi nhân AB, ta nhân ma trận $2^{10} \times 2^{16}$với ma trận $2^{16} \times 2^{5}$, nên kết quả có kích thước $2^{10} \times 2^{5}$. Số phép nhân–cộng vào cỡ

$$2^{10} \cdot 2^{16} \cdot 2^{5} = 2^{31}.$$

Đối với $AC^{\top}$, ta lưu ý rằng $C \in \mathbb{R}^{2^{5} \times 2^{16}}$nên $C^{\top} \in \mathbb{R}^{2^{16} \times 2^{5}}$. Khi đó phép nhân $AC^{\top}$ cũng là nhân ma trận $2^{10} \times 2^{16}$với ma trận $2^{16} \times 2^{5}$, nên kết quả cũng có kích thước $2^{10} \times 2^{5}$và số phép tính vẫn là

$$2^{10} \cdot 2^{16} \cdot 2^{5} = 2^{31}.$$

Về mặt lý thuyết, hai phép nhân AB và $AC^{\top}$ có cùng số phép toán và cùng kích thước đầu vào, vì vậy độ phức tạp thời gian là như nhau. Bất kỳ chênh lệch nhỏ nào về tốc độ (nếu có) thường đến từ chi tiết cài đặt: việc tạo ra $C^{\top}$ có được thực hiện như một view (chỉ đổi cách đọc bộ nhớ) hay là phải copy dữ liệu sang một vùng nhớ mới; dữ liệu có nằm liên tiếp trong bộ nhớ (contiguous) để tối ưu truy cập vector hóa hay không; thư viện BLAS sử dụng kernel nào cho trường hợp dữ liệu đã transpose sẵn, v.v.

Phần thứ hai của câu hỏi: chuyện gì xảy ra nếu ta khởi tạo $C = B^{\top}$ mà không clone bộ nhớ, tức là C chỉ là một view của $B^{\top}$? Khi đó C và $B^{\top}$ thực chất chia sẻ cùng một vùng dữ liệu, chỉ khác cách đánh chỉ số. Ta có

$$C = B^{\top} \Rightarrow C^{\top} = (B^{\top})^{\top}.$$

Theo bài toán 1 đã chứng minh, $(B^{\top})^{\top} = B$. Suy ra

$$C^\top = B.$$

Thay vào biểu thức, ta được

$$AC^\top = A(B^\top)^\top = AB.$$

Nghĩa là về mặt toán học, hai phép nhân bây giờ hoàn toàn giống nhau: cả hai đều là AB với cùng ma trận A và B. Nếu việc gán $C = B^\top$ chỉ tạo ra một view (không copy dữ liệu), thư viện tuyến tính sẽ chỉ cần biết rằng đối với $AC^\top$ nó đang thực hiện phép nhân với đúng ma trận B ban đầu. Khi đó chi phí tính toán, mô hình truy cập bộ nhớ và tốc độ trên thực tế sẽ gần như trùng với phép nhân AB. Sự khác biệt chỉ còn nằm ở mức độ cài đặt: thư viện có tối ưu tốt cho trường hợp các toán hạng là view transpose hay không; nhưng về bản chất, không có lý do gì để một trong hai phép nhân chậm hơn phép còn lại.

12. Consider three matrices, say $A, B, C \in \mathbb{R}^{100\times200}$. Construct a tensor with three axes by stacking $[A, B, C]$. What is the dimensionality? Slice out the second coordinate of the third axis to recover B. Check that your answer is correct.

Ta cần xây dựng một tensor ba chiều từ ba ma trận A, B, C, mỗi ma trận đều có cùng kích thước 100×200. "Stacking $[A, B, C]$" có thể hiểu là tạo thêm một trục mới, trên trục đó ta đặt lần lượt A, B, C theo thứ tự. Để trục mới này thực sự trở thành "trục thứ ba" (axis 2, đánh số từ 0) như đề bài nói, ta chọn cách xếp chồng theo chiều cuối cùng.

Cụ thể, ta định nghĩa một tensor T sao cho với mọi chỉ số hàng $i = 1, \dots, 100$ và cột $j = 1, \dots, 200$:

$$T_{ij0} = A_{ij}, T_{ij1} = B_{ij}, T_{ij2} = C_{ij}.$$

Như vậy, với mỗi cặp (i, j), trục thứ ba (axis 2) của T chứa một vector ba phần tử (A_{ij}, B_{ij}, C_{ij}). Điều này có nghĩa là T có:

- 100 giá trị theo trục 0 (hàng),
- 200 giá trị theo trục 1 (cột),
- 3 giá trị theo trục 2 (chọn A, B hay C).

Do đó, dimensionality (shape) của tensor là

$$T \in \mathbb{R}^{100\times200\times3}.$$

Bây giờ, để "slice out the second coordinate of the third axis to recover B", ta cố định chỉ số của trục thứ ba bằng 1 (second coordinate nếu đánh số từ 0) và giữ nguyên hai trục còn lại. Tensor con thu được có phần tử

$$(T_{ij1})_{1\le i\le100,\ 1\le j\le200}.$$

Theo cách ta xây dựng ở trên, $T_{ij1} = B_{ij}$ với mọi i, j. Vì vậy lát cắt này chính là ma trận B, với shape là 100×200, trùng khớp với kích thước ban đầu của B. Điều này kiểm tra lại được rằng cách ta xếp chồng và cắt (slice) là đúng theo yêu cầu của đề bài.

Chương IV. THỬ NGHIỆM

4.1. Mục tiêu thử nghiệm

Ở phần thử nghiệm về đại số tuyến tính, mục tiêu tổng quát là kiểm tra tính đúng đắn của các tính chất cơ bản (như các đẳng thức về transpose, cộng, nhân ma trận, len (X), cách hoạt động của linalg.norm, khoảng cách Manhattan so với Euclid, ảnh hưởng của thứ tự nhân ma trận đến chi phí tính toán, ví dụ $(AB)C$ so với $A(BC)$) trên một tập nghiệm nhỏ nhưng có cấu trúc, đồng thời rút ra trực giác hình học và ý nghĩa trong bối cảnh khoa học máy tính và học máy. Thay vì chứng minh thuần túy trên ký hiệu, các thử nghiệm được thiết kế sao cho mỗi tính chất lý thuyết đều được "chiếu" xuống các tensor cụ thể, từ đó giúp người đọc thấy rõ điều gì luôn đúng, điều gì phụ thuộc vào shape và cách triển khai, và vì sao những kết quả đó quan trọng khi cài đặt mô hình trên máy tính.

Ở cụm A (Objects & Shape), phần thử nghiệm nhằm làm rõ sự khác nhau giữa bốn loại đối tượng cơ bản trong đại số tuyến tính: scalar, vector, matrix và tensor bậc cao, dưới góc nhìn số chiều (rank), hình dạng (shape) và cách diễn giải trong bối cảnh dữ liệu thật. Thông qua các ví dụ cụ thể, người đọc sẽ thấy rằng scalar, vector và matrix thực chất chỉ là các trường hợp đặc biệt của tensor với số chiều lần lượt là 0, 1và 2, đồng thời hiểu rõ hơn các thao tác cơ bản như truy cập phần tử, reshape, stack, len (X)và slice theo trục để thu lại một ma trận con (ví dụ bài toán stack $[A, B, C]$ rồi trích lại B). Bên cạnh đó, cụm này cũng xây dựng một tập nghiệm nhỏ nhưng đủ phong phú để có thể tái sử dụng ở các phần sau (phép toán tensor, dot product, norm…), đồng thời vẫn đơn giản đến mức có thể kiểm tra trực quan hoặc tính tay được.

Dựa trên tập nghiệm đó, cụm B (Arithmetic & Operations trên tensor) đặt mục tiêu minh họa và kiểm chứng bằng thực nghiệm các phép toán và tính chất đại số cơ bản trên tensor: cộng, trừ và nhân theo phần tử; các tính chất giao hoán, kết hợp, phân phối trong giới hạn cho phép; các phép reduction (như sum, mean) so với non-reduction sum và broadcasting; cùng với các phép toán tuyến tính cốt lõi như dot product, matrix–vector product và matrix–matrix product. Cụm này cũng kiểm tra trực tiếp các đẳng thức về transpose, chẳng hạn $(A^\top)^\top = A$, $(A + B)^\top = A^\top + B^\top$ và việc $A + A^\top$ luôn đối xứng, cũng như các hiện tượng tính toán như A/A.sum(axis $= 1$) tạo ra ma trận được chuẩn hóa theo hàng, hay việc đổi thứ tự nhân $(AB)C$ so với $A(BC)$ dẫn tới chênh lệch rất lớn về số phép toán và kích thước ma trận trung gian. Thông qua việc áp dụng cùng một tập scalar, vector, matrix và tensor đã định nghĩa ở cụm A, phần này cho thấy cách những phép toán đó hoạt động trên dữ liệu thật, đồng thời liên hệ chúng với các thao tác quen thuộc trong khoa học máy tính và học máy như biến đổi tuyến tính, weighted sum, tính toán đầu ra của một lớp tuyến tính trong mạng nơ-ron và tối ưu hóa hiệu năng khi nhân ma trận kích thước lớn.

Tiếp nối A và B, cụm C (Norms & Distances) hướng tới việc khảo sát thực nghiệm các khái niệm "độ lớn" và "khoảng cách" của vector và matrix thông qua các loại norm: ℓ_2-norm (Euclid), ℓ_1-norm (Manhattan), họ ℓ_p-norm tổng quát và Frobenius norm cho ma trận, cũng như cách hàm linalg.norm xử lý các tensor nhiều chiều. Mục tiêu là làm rõ cách các norm này được tính toán trên cùng tập nghiệm đã dùng ở hai cụm trước, cách chúng phản ứng với việc scale vector, cách so sánh "dài/ngắn" giữa các vector khác nhau (bao gồm cả outlier có norm lớn), và cách diễn giải khoảng cách Manhattan dưới dạng đường đi trên lưới so với khoảng cách Euclid. Từ các kết quả thực nghiệm này, người đọc có thể thấy rõ vai trò của norm trong việc đo lỗi, đo khoảng cách, chuẩn hóa trọng số và xây dựng hàm mất mát trong các bài toán tối ưu hóa và học sâu, cũng như hiểu vì sao lựa chọn norm khác nhau ℓ_1, ℓ_2, Frobenius sẽ dẫn đến những hành vi regularization khác nhau của mô hình.

4.2. Tập nghiệm

Tập nghiệm của cụm A được thiết kế có kích thước vừa phải nhưng đa dạng, và được khởi tạo trực tiếp trong code, sao cho có thể tái sử dụng xuyên suốt cả ba cụm A, B, C. Mục tiêu là mỗi nhóm đối tượng (scalar, vector, matrix, tensor) vừa đủ đơn giản để quan sát trực quan, vừa đủ phong phú để kiểm tra hầu hết các tính chất đã nêu, bao gồm shape, broadcasting, reduction, transpose, nhân ma trận, dot product, norms và khoảng cách.

Với scalars, ta chọn năm giá trị $\alpha_1, \dots, \alpha_5 \in \mathbb{R}$thể hiện đủ các trường hợp âm, dương, nhỏ, lớn và bằng không, để sau này có thể minh họa hiệu ứng co giãn khi nhân vào vector hoặc matrix, cũng như kiểm tra tính chất $\| \alpha x \| = | \alpha | \ \| x \|$trong cụm C. Cụ thể trong code:

```
alpha1 = torch.tensor(-2.0)
alpha2 = torch.tensor(-0.5)
alpha3 = torch.tensor(0.0)
alpha4 = torch.tensor(0.5)
alpha5 = torch.tensor(2.0)
scalars = [alpha1, alpha2, alpha3, alpha4, alpha5]
```

Tập scalar này sẽ được dùng ở cụm B để thử nghiệm nhân theo phần tử và broadcasting với tensor, và ở cụm C để khảo sát tác động của phép scale lên các loại norm khác nhau.

Với vectors, ta sử dụng năm vector 3 chiều trong $\mathbb{R}^3$ với các hướng và độ lớn khác nhau, trong đó v_5được cố ý chọn có norm lớn (outlier) để dùng ở phần norm và khoảng cách sau này. Các vector được định nghĩa như sau:

```
v1 = torch.tensor([1., 0., 0.])
v2 = torch.tensor([1., 2., 3.])
v3 = torch.tensor([-1., 1., 0.])
v4 = torch.tensor([0.5, 0.5, 0.5])
v5 = torch.tensor([10., 0., 0.])
vectors = [v1, v2, v3, v4, v5]
```

Trong cụm A, các vector này giúp minh họa shape, rank và cách diễn giải hình học trong $\mathbb{R}^3$. Trong cụm B, chúng được dùng để kiểm tra dot product, matrix–vector product (khi nhân với các ma trận ở dưới) và các phép cộng/trừ theo phần tử. Trong cụm C, cùng một tập vectors sẽ được dùng để so sánh ℓ_1-norm, ℓ_2-norm, ℓ_∞-norm, khảo sát ảnh hưởng của việc scale và minh họa khoảng cách Euclid so với khoảng cách Manhattan giữa hai điểm trong mặt phẳng.

Đối với matrices, ta chuẩn bị một nhóm ma trận nhỏ kích thước 2×3 và 3×2 để biểu diễn tình huống "số mẫu × số đặc trưng", cùng với các ma trận đặc biệt như identity, diagonal và một ma trận 2×2 "random" với giá trị dễ nhìn. Cụ thể:

Code	Definition
`A1 = torch.tensor([[1., 2., 0.], [3., 1., 1.]])`	2×3, minh họa "2 mẫu × 3 đặc trưng"
`A2 = torch.tensor([[1., 0.], [0., 1.], [2., -1.]])`	3×2, dùng cho phép nhân `A1 @ A2`
`A3 = torch.eye(2)`	identity 2×2
`A4 = torch.diag(torch.tensor([2., 0.5]))`	ma trận diagonal, scale khác nhau theo từng trục
`A5 = torch.tensor([[4., -1.], [2., 3.]])`	ma trận 2×2 tổng quát

Nhóm ma trận này cho phép cụm A minh họa rõ các shape kiểu $(\text{số mẫu}, \text{số đặc trưng})$và mối quan hệ giữa các chiều khi nhân ma trận–vector hoặc ma trận–ma trận. Trong cụm B, chúng được dùng để kiểm tra các tính chất liên quan đến transpose, ví dụ $(A^\top)^\top = A$, $(A + B)^\top = A^\top + B^\top$, xây dựng ma trận đối xứng $A + A^\top$, cũng như so sánh chi phí và shape của các phép nhân $(AB)C$ so với $A(BC)$ trong các ví dụ được scale lên kích thước lớn. Đồng thời, các ma trận 2×2 và 3×2 cũng là đầu vào tự nhiên cho các thử nghiệm liên quan đến Frobenius norm và linalg.norm ở cụm C.

Cuối cùng, ta tạo ít nhất một tensor 3D kích thước $5 \times 3 \times 4$ biểu diễn 5mẫu, mỗi mẫu là chuỗi 3 bước thời gian với 4 đặc trưng, và một tensor 4D kích thước $2 \times 1 \times 2 \times 3$ để minh họa cho batch ảnh nhỏ theo format $(\text{batch}, \text{channel}, \text{height}, \text{width})$:

```
T_seq = torch.arange(5 * 3 * 4, dtype=torch.float32).reshape(5, 3, 4)
T_img = torch.arange(2 * 1 * 2 * 3, dtype=torch.float32).reshape(2, 1, 2, 3)
```

Tensor T_seq được dùng trong cụm A để minh họa các shape bậc 3 và thao tác slice theo batch, theo thời gian hoặc theo đặc trưng; trong cụm B để kiểm tra reduction theo từng axis (sum, mean…), broadcasting với vector đặc trưng và cách linalg.norm xử lý tensor nhiều trục; và trong cụm C để kiểm tra rằng linalg.norm(T) thực chất tính $\sqrt{\sum x_{i_1 i_2 i_3}^2}$. Tensor T_img giúp liên hệ trực

tiếp với bối cảnh xử lý ảnh trong học sâu, nơi các phép biến đổi như cộng bias theo channel, nhân theo phần tử với kernel hoặc chuẩn hóa theo batch đều dựa trên cùng các cơ chế broadcasting và reduction đã nêu.

Toàn bộ các đối tượng này được thiết kế để có thể tái sử dụng ở cả ba cụm A, B, C (objects & shapes, arithmetic & operations, norms & distances), giúp phần thực nghiệm nhất quán, không rời rạc và cho phép kiểm tra từ hình học cơ bản đến các tính chất đại số và độ lớn, khoảng cách trong không gian vector.

4.3. Phương thức thử nghiệm và kết quả

4.3.1. Cụm A – Objects & Shapes

4.3.1.a. Phương thức thử nghiệm

Dựa trên tập nghiệm đã xây dựng ở mục trước (các biến α_1–α_5, v_1–v_5, A_1–A_5, T_{seq}, T_{img}), phần này tập trung vào việc quan sát và phân tích thuộc tính hình dạng của từng loại đối tượng. Mục tiêu là kiểm tra thực nghiệm rằng scalar đúng là tensor bậc 0, vector là tensor bậc 1, matrix là tensor bậc 2, và các tensor bậc cao hơn chỉ đơn giản là mở rộng thêm trục, đồng thời xác nhận rằng $\text{len}(X)$luôn trả về kích thước trên trục đầu tiên.

Trước hết, với scalars, ta kiểm tra rằng mỗi biến α_i đúng là một tensor 0 chiều bằng cách in ra shape, số chiều và giá trị tương ứng, đồng thời dùng thêm hàm len để thấy rằng $\text{len}(\alpha_i)$ không được định nghĩa (scalar không có trục nào để đo chiều dài):

```
for i, a in enumerate(scalars, start=1):
    print(f"Scalar alpha{i}: value = {a.item():>4},  shape = {a.shape},  dim
= {a.dim()}")
```

Tiếp theo, với vectors, ta sử dụng danh sách v_i đã tạo để minh họa tensor 1 chiều. Ta in vector, shape, dim, đồng thời thử một vài thao tác indexing cơ bản để cho thấy mỗi phần tử có thể được truy cập như một "feature"; ngoài ra, ta dùng len để khẳng định rằng $\text{len}(v_i)$ đúng bằng kích thước trên trục duy nhất của vector:

```
for i, v in enumerate(vectors, start=1):
    print(f"Vector v{i} = {v},  shape = {v.shape},  dim = {v.dim()},  len =
{len(v)}")

print("v2[0]  =", v2[0].item())
print("v2[1:] =", v2[1:])
```

Đối với matrices, ta duyệt qua các ma trận trong danh sách matrices và ghi nhận shape, dim cùng tổng số phần tử. Điều này giúp làm rõ rằng matrix là tensor 2 chiều rất phù hợp để biểu diễn bảng dữ liệu, trong đó trục 0 thường được hiểu là số mẫu và trục 1 là số đặc trưng. Đồng thời, ta dùng len để xác nhận rằng $\text{len}(A)$ bằng số hàng của ma trận:

```
for i, M in enumerate(matrices, start=1):
```

```
    print(f"Matrix A{i} =\n{M}")
    print("  shape =", M.shape,
          ", dim =", M.dim(),
          ", len =", len(M),
          ", tổng số phần tử =", M.numel())
```

Để nhấn mạnh cách diễn giải "hàng = mẫu, cột = đặc trưng", ta thao tác cụ thể trên A_1:

```
print("A1 =\n", A1)
print("Hàng thứ 0 (mẫu 1)  :", A1[0])
print("Hàng thứ 1 (mẫu 2)  :", A1[1])
print("Cột thứ 0 (feature 1):", A1[:, 0])
print("Cột thứ 2 (feature 3):", A1[:, 2])
```

Cuối cùng, với tensors bậc cao, ta sử dụng trực tiếp T_{seq} và T_{img} đã được định nghĩa để minh họa tensor 3 chiều và 4 chiều. Ta quan sát shape, số chiều, len và một vài lát cắt điển hình: theo mẫu, theo bước thời gian, theo kênh ảnh. Việc in len cho thấy rõ rằng len (T_{seq}) bằng số mẫu trong batch chuỗi, còn len (T_{img}) bằng số ảnh trong batch:

```
print("T_seq shape =", T_seq.shape, ", dim =", T_seq.dim(), ", len =",
len(T_seq))
print("T_seq[0]       -> mẫu 0, shape", T_seq[0].shape)
print("T_seq[0, 1]    -> mẫu 0, time step 1, shape", T_seq[0, 1].shape)
print("T_seq[:, :, 0] -> toàn bộ mẫu & time, feature 0, shape", T_seq[:, :,
0].shape)

print("T_img shape =", T_img.shape, ", dim =", T_img.dim(), ", len =",
len(T_img))
print("T_img[0]      -> ảnh thứ 1, shape", T_img[0].shape)
print("T_img[0, 0]   -> kênh 0 của ảnh 1, shape", T_img[0, 0].shape)
```

Bổ sung thêm, ta có thể kiểm tra nhanh một ví dụ stack để kết nối với bài toán $[;A,B,C;]$: chẳng hạn stack A_3, A_4, A_5 theo một trục mới để thu được tensor 3 chiều và kiểm tra việc slice lại từng ma trận thành phần:

```
A_stack = torch.stack([A3, A4, A5], dim=2)
print("A_stack shape =", A_stack.shape)
print("Slice lại A4 từ A_stack:\n", A_stack[:, :, 1])
```

Song song với các thao tác trên, ta diễn giải ý nghĩa của từng loại tensor trong bối cảnh dữ liệu: scalar là một giá trị đo lường đơn lẻ; vector là một điểm dữ liệu hoặc một embedding trong không gian đặc trưng; matrix là bảng dữ liệu nhiều mẫu, nhiều đặc trưng; tensor 3 chiều như T_{seq} biểu diễn tập các chuỗi thời gian; tensor 4 chiều như T_{img} biểu diễn batch ảnh với nhiều kênh. Qua chuỗi bước này, người đọc thấy rõ rằng việc tăng số chiều chỉ là thêm trục để chứa thêm cấu trúc dữ liệu, và tất cả các đối tượng trên đều thống nhất trong khái niệm chung là tensor với shape cụ thể.

4.3.1.b. Kết quả thử nghiệm

Khi ra output, ta thấy toàn bộ các giá trị vô hướng α_1–α_5 đều có `shape = torch.Size([])` và `dim = 0`. Điều này khẳng định chúng là các tensor 0 chiều, hay chính là scalars trong ngôn ngữ đại số tuyến tính. Mỗi scalar chỉ mang một giá trị số đơn lẻ nhưng vẫn có thể tham gia các phép toán với vector và matrix ở các phần sau, chẳng hạn như nhân vào một vector vđể quan sát hiệu ứng co giãn độ dài $\| \alpha v \|$.

Đối với vector v_2, output cho thấy `shape = torch.Size([3])`, `dim = 1` và `len(v2) = 3`, đúng với trực giác rằng đây là một tensor 1 chiều gồm 3 phần tử. Các lệnh `v2[0]` và `v2[1:]` lần lượt trả về phần tử đầu tiên và hai phần tử còn lại, minh họa việc coi mỗi phần tử của vector là một đặc trưng (feature) của một điểm dữ liệu trong không gian $\mathbb{R}^3$. Các vector khác v_1, v_3, v_4, v_5 cũng đều có `dim = 1` và `len = 3`, củng cố nhận định rằng vector trong thử nghiệm này luôn là tensor một chiều với ba tọa độ.

Ma trận A_1được in ra với `shape = torch.Size([2, 3])` và `dim = 2`, nghĩa là 2 hàng × 3 cột, đồng thời `len(A1) = 2` cho thấy hàm `len` trả về số hàng (kích thước trên trục 0). Khi truy cập `A1[0]` và `A1[1]`, ta thu được hai vector chiều dài 3, tương ứng với hai mẫu dữ liệu khác nhau. Ngược lại, các biểu thức `A1[:, 0]` và `A1[:, 2]` trả về các vector chứa giá trị của đặc trưng thứ 1 và thứ 3 trên toàn bộ mẫu. Điều này xác nhận cách diễn giải quen thuộc "hàng = mẫu, cột = đặc trưng" khi dùng matrix để biểu diễn dữ liệu dạng bảng, và cho thấy matrix chính là tensor 2 chiều trong đó trục 0 đóng vai trò batch các mẫu, trục 1 là các feature.

Với tensor 3 chiều T_{seq}, ta quan sát được `shape = torch.Size([5, 3, 4])`, `dim = 3` và `len(T_seq) = 5`, cho thấy cấu trúc 5 mẫu × 3 time steps × 4 features. Lát cắt `T_seq[0]` là một tensor 2D 3×4, mô tả toàn bộ chuỗi thời gian của mẫu thứ nhất. Lát cắt nhỏ hơn `T_seq[0, 1]` là một vector 4 chiều, tương ứng với bước thời gian thứ hai của mẫu đó. Còn `T_seq[:, :, 0]` giữ lại toàn bộ mẫu và toàn bộ bước thời gian nhưng chỉ lấy feature thứ 0, nên có shape 5×3. Những lát cắt này minh họa rõ cách tensor 3D gói nhiều trục thông tin (mẫu, thời gian, đặc trưng) vào cùng một cấu trúc thống nhất.

Tương tự, tensor 4D T_{img} có `shape = torch.Size([2, 1, 2, 3])`, `dim = 4` và `len(T_img) = 2`, thể hiện 2 ảnh, 1 kênh, kích thước 2×3 pixel. Lát cắt `T_img[0]` trả về một tensor $1 \times 2 \times 3$, đại diện cho toàn bộ ảnh thứ nhất (với 1 kênh). Khi đi sâu thêm một mức đến `T_img[0, 0]`, ta thu được một ma trận 2×3 chứa các giá trị pixel của kênh duy nhất của ảnh này. Đây chính là cấu trúc quen thuộc khi xử lý ảnh trong deep learning với format $(\text{batch}, \text{channel}, \text{height}, \text{width})$.

Từ các quan sát trên, ta rút ra rằng scalar, vector và matrix lần lượt là các tensor 0D, 1D, 2D; còn các tensor 3D, 4D chỉ là sự mở rộng tự nhiên khi ta "xếp chồng" thêm trục để mô tả cấu trúc phức tạp hơn như chuỗi thời gian hay ảnh. Đồng thời, việc sử dụng chung một tập nghiệm gồm các α_i, v_i, A_j, T_{seq} và T_{img} ở cụm A cho thấy chúng là nền tảng tốt để tiếp tục dùng trong các cụm tiếp theo (tensor arithmetic, dot product, matrix–vector, matrix–matrix, norms), giúp toàn bộ phần thực nghiệm của báo cáo giữ được tính nhất quán và dễ theo dõi.

4.3.2. Cụm B – Arithmetic & Operations

4.3.2.a. Phương thức thử nghiệm

Trong cụm B, tất cả các phép toán được thực hiện trực tiếp trên tập nghiệm đã xây dựng ở cụm A, bao gồm các scalar $\alpha_1, \dots, \alpha_5$, các vector $v_1, \dots, v_5$, các ma trận $A_1, \dots, A_5$ và hai tensor bậc cao $T_{\text{seq}}, T_{\text{img}}$. Khi cần minh họa thêm cho matrix–matrix multiplication, broadcasting hoặc transpose, ta chỉ bổ sung thêm một vài vector, ma trận nhỏ có kích thước phù hợp, nhưng vẫn giữ tinh thần dùng chung một khung dữ liệu thống nhất xuyên suốt. Mỗi khái niệm trong cụm B được kiểm tra bằng nhiều ví dụ ngắn, với các đoạn mã cụ thể để người đọc có thể chạy lại và quan sát.

Trước hết, với các tính chất cơ bản của phép toán trên tensor, ta tập trung vào việc kiểm tra giao hoán, kết hợp và phân phối. Đối với vector, ta chọn ba vector 3D v_2, v_3, v_4 và dùng đoạn mã sau để so sánh các biểu thức cộng tương ứng:

```
x, y, z = v2, v3, v4

print("x + y      =", x + y)
print("y + x      =", y + x)
print("Equal?     =", torch.allclose(x + y, y + x))

print("(x + y) + z =", (x + y) + z)
print("x + (y + z) =", x + (y + z))
print("Equal?      =", torch.allclose((x + y) + z, x + (y + z)))
```

Kết quả cho thấy $x + y = y + x$ và $(x + y) + z = x + (y + z)$ với sai số số học bằng không trong `torch.allclose`, minh họa tính giao hoán và kết hợp của phép cộng vector.

Để kiểm tra tính phân phối của nhân với scalar, ta dùng một scalar như $\alpha_5 = 2.0$ và so sánh hai cách viết $\alpha(x + y)$ và $\alpha x + \alpha y$:

```
alpha = alpha5
print("alpha * (x + y)     =", alpha * (x + y))
print("alpha * x + alpha*y =", alpha * x + alpha * y)
print("Equal?              =", torch.allclose(alpha * (x + y),
                                              alpha * x + alpha * y))
```

Kết quả thu được cho thấy hai vế trùng khớp, khẳng định tính phân phối của nhân với scalar trên vector. Với ma trận, ta áp dụng cùng kiểu kiểm tra trên A_1, A_5 để xác nhận rằng $\alpha(A + B) = \alpha A + \alpha B$ và $A + B = B + A$ vẫn đúng ở cấp ma trận.

Song song đó, ta đặt cạnh nhau phép nhân theo phần tử (Hadamard) và phép cộng trên vector để nhấn mạnh rằng đây là hai phép toán khác biệt nhưng dùng cùng shape:

```
print("x * y (Hadamard) =", x * y)
print("x + y            =", x + y)
```

Trong phần reduction, mục tiêu là xem các hàm như `sum`, `mean`, `max` được áp dụng như thế nào trên các chiều khác nhau của tensor. Trên vector v_2, ta dùng đoạn mã:

```
print("v2        =", v2)
print("v2.sum() =", v2.sum())
```

```
print("v2.mean()=", v2.mean())
print("v2.max() =", v2.max())
```

Kết quả cho thấy `v2.sum()` đúng bằng tổng các phần tử, `v2.mean()` bằng trung bình, và `v2.max()` là phần tử lớn nhất, phù hợp với định nghĩa reduction trên tensor một chiều.

Đối với ma trận A_1, ta tính tổng theo cột và theo hàng bằng:

```
print("A1 =\n", A1)
print("Sum over rows   (dim=0):", A1.sum(dim=0))
print("Sum over columns(dim=1):", A1.sum(dim=1))
```

Sau đó, ta dùng thêm hai biến `sum_no_keep` và `sum_keep` để so sánh shape khi có và không sử dụng `keepdim`:

```
sum_no_keep = A1.sum(dim=1)
sum_keep    = A1.sum(dim=1, keepdim=True)
print("Without keepdim:", sum_no_keep, "shape =", sum_no_keep.shape)
print("With    keepdim:", sum_keep,    "shape =", sum_keep.shape)
```

Thí nghiệm này cho thấy reduction theo `dim=1` làm giảm ma trận A_1 từ shape 2×3 xuống vector độ dài 2 khi không giữ chiều, còn khi `keepdim=True` thì kết quả vẫn có hai chiều nhưng cột thu gọn về kích thước 1. Đây là cơ chế quan trọng để kiểm soát shape trong các mạng nơ-ron, đặc biệt khi cần broadcast kết quả reduction trở lại.

Trên tensor 3D T_{seq} kích thước $5 \times 3 \times 4$, ta thực hiện reduction theo từng trục bằng cách gọi `sum` trên từng `dim`:

```
print("T_seq shape:", T_seq.shape)
print("Sum over batch   (dim=0), shape", T_seq.sum(dim=0).shape)
print("Sum over time    (dim=1), shape", T_seq.sum(dim=1).shape)
print("Sum over feature (dim=2), shape", T_seq.sum(dim=2).shape)
```

Kết quả cho thấy sum theo `dim=0` cho tensor 3×4 (gộp các mẫu), theo `dim=1` cho tensor 5×4 (gộp theo time steps), và theo `dim=2` cho tensor 5×3(gộp theo feature). Điều này đúng với lý thuyết rằng reduction trên trục nào sẽ loại bỏ trục đó khỏi shape.

Một trường hợp quan trọng khác là biểu thức chuẩn hóa hàng $A/A.\text{sum}(\text{axis} = 1)$. Ta kiểm tra trên một ma trận X để xem mỗi hàng sau chuẩn hóa có tổng xấp xỉ bằng 1:

```
X = torch.tensor([
    [1., 2., 0.],
    [3., 1., 1.],
    [0., 0., 1.],
    [2., 3., 4.],
    [4., 1., -1.]
])

row_sum = X.sum(dim=1, keepdim=True)
X_norm = X / row_sum

print("X =\n", X)
```

```
print("Row sums  =", row_sum.squeeze())
print("X_norm =\n", X_norm)
print("Row sums of X_norm =", X_norm.sum(dim=1))
```

Output cho thấy mỗi hàng của `X_norm` có tổng rất gần 1, minh họa cơ chế broadcasting: `row_sum` có shape 5×1 được broadcast thành 5×3 khi chia, đồng thời khẳng định ý nghĩa của biểu thức $A/A.\text{sum}(\text{axis} = 1)$ như một phép chuẩn hóa theo hàng.

Phần non-reduction sum và broadcasting tiếp tục sử dụng các phép cộng để kiểm tra cách PyTorch xử lý các tensor có cùng shape và các trường hợp được broadcast. Ta cộng hai vector cùng shape:

```
print("v2 =", v2)
print("v3 =", v3)
print("v2 + v3 =", v2 + v3)
```

và hai ma trận cùng shape A_3, A_4:

```
print("A3 =\n", A3)
print("A4 =\n", A4)
print("A3 + A4 =\n", A3 + A4)
```

Sau đó, ta áp dụng broadcasting giữa một scalar và ma trận A_1:

```
print("alpha4 =", alpha4.item())
print("A1 + alpha4 =\n", A1 + alpha4)
```

Kết quả cho thấy scalar α_4 được nhân bản trên toàn bộ phần tử của A_1, đúng với quy tắc broadcasting scalar.

Để kiểm tra broadcasting theo cột, ta sử dụng ma trận X kích thước 5×3 và một vector `b_feat` có shape $(3,\ )$, rồi cộng chúng:

```
b_feat = torch.tensor([0.1, -0.2, 0.3])

print("X shape      =", X.shape)
print("b_feat shape =", b_feat.shape)
print("X + b_feat =\n", X + b_feat)
```

Output cho thấy `b_feat` được broadcast thành một ma trận 5×3 bằng cách lặp lại vector này cho từng hàng, minh họa quy tắc "align theo trục cuối" của broadcasting.

Cuối cùng, broadcasting trên tensor 3D T_{seq} được kiểm tra bằng cách cộng thêm một vector đặc trưng chiều dài 4:

```
feat_bias = torch.tensor([0.5, 0., -0.5, 1.0])
print("T_seq shape      =", T_seq.shape)
print("feat_bias shape  =", feat_bias.shape)
print("T_seq + feat_bias shape =", (T_seq + feat_bias).shape)
```

Kết quả cho thấy vector `feat_bias` shape $(4,\ )$ được broadcast thành shape $5 \times 3 \times 4$, tương ứng với việc cộng cùng một bias cho mọi mẫu và mọi time step trên từng feature, giống với cách cộng bias theo chiều feature trong nhiều kiến trúc mạng nơ-ron.

Trong phần dot product, ta dùng lại các vector 3D trong tập nghiệm để kiểm tra sự tương đương giữa `torch.dot` và $\sum_i a_i\, b_i$. Một hàm nhỏ được định nghĩa để in song song hai cách tính trên từng cặp vector:

```
def show_dot(a, b, name_a, name_b):
    dot1 = torch.dot(a, b)
    dot2 = (a * b).sum()
    print(f"{name_a} · {name_b} via torch.dot   = {dot1}")
    print(f"{name_a} · {name_b} via sum(a*b)    = {dot2}")
    print()

show_dot(v1, v5, "v1", "v5")
show_dot(v2, v4, "v2", "v4")
show_dot(v2, v3, "v2", "v3")
```

Kết quả cho tất cả các cặp đều trùng khớp, xác nhận công thức $\langle a, b\rangle = \sum_i a_i\, b_i$ trong code. Ta cũng thiết lập một trường hợp để xem dot product dưới dạng tổng có trọng số (weighted sum) bằng cách coi $v_4$là vector trọng số và $v_2$là vector giá trị đặc trưng:

```
w     = v4
x_vec = v2
print("w · x_vec =", torch.dot(w, x_vec))
```

Giá trị trả về có thể được diễn giải như một dự đoán tuyến tính đơn giản từ các feature trong x_{vec}.

Trong phần matrix–vector product, ta sử dụng ma trận A_1 kích thước 2×3 nhân với vector v_2 độ dài 3 bằng phép toán `@`, và đồng thời tính dot product của từng hàng với v_2:

```
y_mv = A1 @ v2
print("A1 =\n", A1)
print("v2 =", v2)
print("A1 @ v2 =", y_mv)

row0_dot = torch.dot(A1[0], v2)
row1_dot = torch.dot(A1[1], v2)
print("dot(A1[0], v2) =", row0_dot)
print("dot(A1[1], v2) =", row1_dot)
```

Output cho thấy phần tử thứ i của $A_1@v_2$ đúng bằng dot product giữa hàng thứ i của A_1và v_2, khẳng định mô tả $y_i = a_i^{\top} x$ trong lý thuyết. Thao tác tương tự được thực hiện với ma trận vuông $A_5(2 \times 2)$ và vector x_2 lấy từ hai phần tử đầu của v_2:

```
x2 = v2[:2]
print("A5 =\n", A5)
print("x2 =", x2)
print("A5 @ x2 =", A5 @ x2)
```

Trong phần matrix–matrix multiplication, ta trước hết sử dụng hai ma trận vuông nhỏ B_1, B_2 kích thước 2×2 để tính `B1 @ B2` và `B2 @ B1`, qua đó tạo hai kết quả để so sánh:

```
B1 = torch.tensor([[1., 2.],
                   [3., 4.]])
B2 = torch.tensor([[0., 1.],
                   [-1., 0.]])

print("B1 @ B2 =\n", B1 @ B2)
print("B2 @ B1 =\n", B2 @ B1)
```

Hai kết quả thu được khác nhau, minh họa rõ ràng rằng nhân ma trận nói chung không giao hoán, tức $B_1 B_2 \neq B_2 B_1$.

Sau đó ta nhân ma trận chữ nhật $A_1(2 \times 3)$ với $A_2(3 \times 2)$ để tạo ma trận kết quả $C(2 \times 2)$:

```
C = A1 @ A2
print("A1 =\n", A1)
print("A2 =\n", A2)
print("C = A1 @ A2 =\n", C, "shape =", C.shape)
```

Để kiểm tra tính chất composition của các biến đổi tuyến tính, ta chọn một vector 2D x và so sánh hai cách áp dụng lần lượt các ma trận A_2, A_1:

```
x = torch.tensor([1., -1.])

left  = A1 @ (A2 @ x)
right = (A1 @ A2) @ x

print("A2 @ x       =", A2 @ x)
print("A1 @ (A2 @ x)=", left)
print("(A1 @ A2) @ x=", right)
print("Equal? =", torch.allclose(left, right))
```

Kết quả cho thấy `left` và `right` trùng nhau, minh họa tính kết hợp của nhân ma trận $(A_1 A_2)x = A_1(A_2 x)$.

Cuối cùng, để liên hệ với các bài toán kích thước lớn trong phần lý thuyết bài tập 10 và 11, ta không tạo trực tiếp ma trận kích thước $2^{10} \times 2^{16}$ mà chỉ phân tích số phép tính dựa trên quy tắc chi phí mnp cho phép nhân $m \times n$với $n \times p$. Việc này cho phép rút ra rằng với cùng tích ABC nhưng khác thứ tự nhân, số phép tính và kích thước ma trận trung gian có thể chênh lệch rất lớn. Điều này giải thích vì sao khi triển khai trên thực tế, việc lựa chọn thứ tự nhân ma trận hợp lý là yếu tố quan trọng về mặt hiệu năng, dù về mặt toán học các biểu thức vẫn tương đương nhờ tính kết hợp của phép nhân.

4.3.2.b. Kết quả thử nghiệm

Khi kiểm tra các tính chất đại số cơ bản trên các vector v_2, v_3, v_4, mọi đẳng thức đều được thỏa mãn: với mỗi cặp vector, kết quả của $x + y$ và $y + x$ trùng nhau, tức là phép cộng tuân theo tính giao hoán $x + y = y + x$. Tương tự, hai biểu thức $(x + y) + z$ và $x + (y + z)$ luôn cho cùng kết

quả, xác nhận tính kết hợp của phép cộng. Khi nhân một vô hướng $\alpha = \alpha_5 = 2.0$ vào, hai biểu thức $\alpha(x + y)$ và $\alpha x + \alpha y$ cũng luôn bằng nhau, đúng với tính phân phối $\alpha(x + y) = \alpha x + \alpha y$. Lặp lại các phép so sánh tương tự với ma trận A_1, A_5 cho kết quả $A_1 + A_5 = A_5 + A_1$và $\beta(A_1 + A_5) = \beta A_1 + \beta A_5$ với mọi β đã chọn, cho thấy các tính chất đại số đó giữ nguyên trên tensor 2 chiều. Khi đặt cạnh kết quả của $x * y$ (Hadamard product) và $x + y$, ta thấy chúng có cùng shape nhưng giá trị hoàn toàn khác nhau, nhấn mạnh rằng các phép toán cộng, trừ, nhân theo phần tử đều được áp dụng độc lập trên từng phần tử của tensor, còn những khái niệm như tích vô hướng hay biến đổi tuyến tính chỉ xuất hiện khi ta thêm bước cộng tổng hoặc thay đổi cách gom chiều.

Trong phần reduction, kết quả trên vector $v_2 = [1.,2.,3.]$ cho thấy `v2.sum()` bằng 6., `v2.mean()` bằng 2., và `v2.max()` bằng 3.; cả ba đều là tensor 0D (scalar). Điều này phù hợp với trực giác rằng các hàm `sum`, `mean`, `max` đã "gom" toàn bộ thông tin của một vector một chiều xuống một giá trị vô hướng duy nhất. Với ma trận

$$A_1 = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 1 & 1 \end{bmatrix},$$

kết quả `A1.sum(dim=0)` là một vector chiều 3 (tổng theo cột, tức tổng từng đặc trưng), còn `A1.sum(dim=1)` là một vector chiều 2 (tổng theo hàng, tức tổng từng mẫu). Khi so sánh `A1.sum(dim=1)` và `A1.sum(dim=1, keepdim=True)`, ta được cùng giá trị số $(3, 5)$ nhưng shape lần lượt là `torch.Size([2])` và `torch.Size([2, 1])`. Như vậy, tùy chọn `keepdim=True` chỉ giữ lại chiều bị gom với kích thước 1, phục vụ cho các bước broadcasting về sau, chứ không thay đổi giá trị số. Trên tensor 3D T_{seq} với `shape = torch.Size([5, 3, 4])`, việc tính `T_seq.sum(dim=0)`, `sum(dim=1)`, `sum(dim=2)` lần lượt cho ra các tensor có shape `torch.Size([3, 4])`, `torch.Size([5, 4])` và `torch.Size([5, 3])`. Điều này cho thấy rõ: cùng là phép cộng nhưng chọn trục nào để gom sẽ quyết định ta đang cộng theo batch, theo time step hay theo feature.

Một trường hợp đặc biệt là biểu thức chuẩn hóa hàng $A/A.\text{sum}(\text{axis} = 1)$. Khi áp dụng trên ma trận X kích thước 5×3, các hàng của ma trận chuẩn hóa X_{norm} đều có tổng xấp xỉ bằng 1. Điều này xác nhận rằng vector tổng theo hàng có shape $(5, 1)$ đã được broadcast dọc theo trục feature để chia cho từng phần tử của mỗi hàng, và cho thấy rõ ý nghĩa của biểu thức này như một phép chuẩn hóa phân phối xác suất theo hàng, rất giống cơ chế normalizing trong nhiều mô hình học máy.

Ở phần non-reduction sum và broadcasting, cộng hai vector cùng shape như $v_2 + v_3$ cho một vector mới shape `torch.Size([3])`, trong đó mỗi phần tử đúng bằng tổng hai phần tử tương ứng; cộng hai ma trận A_3 và A_4 shape `torch.Size([2, 2])` cho một ma trận 2×2, mỗi ô là tổng của hai ô tương ứng. Khi cộng một scalar α_4 với ma trận A_1, mọi phần tử trong A_1 đều được tăng cùng một lượng, trong khi shape vẫn là `torch.Size([2, 3])`, cho thấy scalar đã được broadcast thành một ma trận cùng kích thước trước khi cộng. Với ma trận Xshape `torch.Size([5, 3])` và vector b_{feat} shape `torch.Size([3])`, kết quả $X + b_{\text{feat}}$ có shape `torch.Size([5, 3])`, trong đó mỗi hàng của X được cộng cùng một vector b_{feat}. Đây là minh họa cụ thể cho việc cộng một vector bias vào toàn bộ batch dữ liệu. Tương tự, với tensor 3D T_{seq} shape `torch.Size([5, 3, 4])` và vector feat_bias shape `torch.Size([4])`, phép cộng $T_{\text{seq}} +$ feat_bias cho một tensor vẫn có shape `torch.Size([5, 3, 4])`, nhưng mỗi lát cắt

(batch,time, :) đã được cộng cùng một vector bias theo trục feature, thể hiện cơ chế broadcasting đồng thời trên nhiều chiều, tương tự cách cộng bias theo feature trong các lớp tuyến tính hoặc tích chập.

Kết quả phần dot product cho thấy với mọi cặp vector thử nghiệm (chẳng hạn v_1 với v_5, v_2 với v_4, v_2 với v_3), giá trị `torch.dot(a, b)` luôn trùng với `(a * b).sum()`. Nói cách khác, thực nghiệm xác nhận chính xác định nghĩa

$$a \cdot b = \sum_i a_i \, b_i.$$

Khi chọn $w = v_4$ làm vector trọng số và $x_{\text{vec}} = v_2$ làm vector giá trị, dot product $w \cdot x_{\text{vec}}$ cho ra đúng một số vô hướng biểu diễn tổng có trọng số của các thành phần trong x_{vec}. Điều này cho phép ta nhìn dot product vừa như một phép toán hình học (liên quan đến góc giữa hai vector), vừa như một phép tính weighted sum rất quen thuộc trong các mô hình tuyến tính và mạng nơ-ron.

Ở phần matrix–vector product, với ma trận A_1 kích thước 2×3 và vector v_2 có độ dài 3, kết quả $A_1 v_2$ là một vector chiều 2. Khi tính riêng `torch.dot(A1[0], v2)` và `torch.dot(A1[1], v2)`, hai giá trị thu được trùng khớp với hai phần tử của `A1 @ v2`. Do đó, thực nghiệm cho thấy rõ rằng phần tử thứ i của Av chính là tích vô hướng $a_i^\top v$ giữa hàng thứ i của ma trận A và vector v, đúng với công thức

$$Av = \begin{bmatrix} a_1^\top v \\ a_2^\top v \\ \vdots \end{bmatrix}.$$

Thử nghiệm tương tự với ma trận vuông A_5 (2×2) và vector x_2 độ dài 2 cũng cho kết luận như vậy, củng cố cách nhìn mỗi hàng của ma trận là một "neuron" với một vector trọng số riêng.

Trong phần matrix–matrix multiplication, hai ma trận vuông 2×2 là B_1 và B_2 cho kết quả `B1 @ B2` và `B2 @ B1` hoàn toàn khác nhau về giá trị từng phần tử, nên ta quan sát được rõ ràng rằng nói chung

$$B_1 B_2 \neq B_2 B_1,$$

tức là phép nhân ma trận không giao hoán. Với cặp A_1 (2×3) và A_2 (3×2), ma trận $C = A_1 A_2$ thu được có shape `torch.Size([2, 2])`. Nếu tính thủ công từng phần tử c_{ij} bằng dot product giữa hàng thứ i của A_1 và cột thứ j của A_2, ta thu được đúng các giá trị trong C, kiểm chứng quy tắc "hàng nhân cột". Khi so sánh hai cách tính $A_1(A_2 x)$ và $(A_1 A_2)x$ với cùng một vector x, cả hai đều cho cùng một vector kết quả và `torch.allclose` trả về `True`. Điều này phù hợp với đẳng thức

$$A_1(A_2 x) = (A_1 A_2)x,$$

cho thấy nhân ma trận thực chất là cách ghép nhiều biến đổi tuyến tính thành một biến đổi duy nhất.

Đối với các tính chất liên quan đến transpose, các thí nghiệm trên những ma trận nhỏ (ví dụ A_3, A_4, A_5) cho thấy $(A^\top)^\top = A$ đúng với mọi ma trận được thử, và $(A+B)^\top = A^\top + B^\top$ đúng cho mọi cặp ma trận cùng shape. Hơn nữa, ma trận $A + A^\top$ luôn đối xứng theo đường chéo chính, xác nhận thực nghiệm rằng $A + A^\top$ là một cách đơn giản để xây dựng ma trận đối xứng từ một ma trận bất kỳ.

Cuối cùng, khi phân tích chi phí tính toán cho các tích ma trận kích thước lớn (theo khuôn bài tập 10 và 11), việc áp dụng quy tắc chi phí mnp cho phép nhân ma trận $m \times n$ với $n \times p$ cho thấy rõ sự khác biệt rất lớn về số phép nhân và số phần tử ma trận trung gian giữa hai cách nhân $(AB)C$ và $A(BC)$. Cách nhân phù hợp (chẳng hạn $(AB)C$ trong bài toán cụ thể) giúp giảm đáng kể số phép toán và tránh phải lưu trữ ma trận trung gian khổng lồ. Thí nghiệm về AB so với $AC^\top$ cũng cho kết luận rằng, nếu C chỉ là view của $B^\top$ (không clone bộ nhớ), thì $AC^\top$ về bản chất chính là AB và chi phí tính toán tương đương, trong khi việc clone không cần thiết sẽ chỉ làm tăng footprint bộ nhớ. Nhìn chung, toàn bộ kết quả thực nghiệm của cụm B cho thấy các phép arithmetic, reduction, dot product, nhân ma trận, transpose và broadcasting trên tập nghiệm đều khớp với lý thuyết đại số tuyến tính, đồng thời kết nối trực tiếp với các thao tác thường dùng trong học máy như weighted sum, lớp tuyến tính, chuẩn hóa theo hàng và tối ưu thứ tự nhân để đạt hiệu năng tính toán tốt hơn.

4.3.3. Cụm C - Norms & Distances

4.3.3.a. Phương thức thử nghiệm

Trong cụm C, ta tiếp tục sử dụng chung tập nghiệm đã xây dựng ở hai cụm trước: các vector $v_1, \dots, v_5$, các ma trận $A_1, \dots, A_5$ và các scalar $\alpha_1, \dots, \alpha_5$. Bên cạnh đó, ta bổ sung thêm một vài vector có giá trị "outlier" để dễ so sánh hành vi giữa ℓ_1-norm và ℓ_2-norm, cũng như vài vector "dự đoán" để minh họa việc dùng norm làm thước đo lỗi. Toàn bộ phép thử đều được cài đặt bằng PyTorch, với cú pháp thống nhất `torch.norm(x, p=...)` cho các ℓ_p-norm và `torch.norm(X)` cho Frobenius norm trên ma trận.

Đầu tiên, ta chọn một nhóm vector đại diện, bao gồm các vector đã dùng trước đó v_2, v_3, v_5 và một vector mới có phần tử rất lớn để đóng vai trò outlier:

```
v_outlier = torch.tensor([1., 2., 100.])
vectors_for_norm = [v2, v3, v5, v_outlier]
```

Với mỗi vector x trong nhóm này, ta lần lượt tính các loại norm khác nhau:

```
for x in vectors_for_norm:
    l2 = torch.norm(x, p=2)   # ||x||_2
    l1 = torch.norm(x, p=1)   # ||x||_1
    lp = torch.norm(x, p=3)   # ||x||_3, ví dụ p = 3
```

```
    print("x =", x)
    print("  ||x||_2 =", l2.item())
    print("  ||x||_1 =", l1.item())
    print("  ||x||_3 =", lp.item())
```

Mục đích ở đây là đặt các giá trị $\| x \|_2$, $\| x \|_1$, $\| x \|_3$ cạnh nhau cho cùng một vector để sau này phân tích sự khác biệt: ℓ_1-norm nhạy với tổng độ lệch tuyệt đối, ℓ_2-norm nhạy hơn với các phần tử lớn (outlier), còn ℓ_3-norm là một ví dụ trung gian trong họ ℓ_p.

Tiếp theo, ta kiểm tra hai tính chất cơ bản của norm: tính đồng biến theo scale $\| \alpha x \|_p = | \alpha | \ \| x \|_p$ và bất đẳng thức tam giác $\| x + y \|_p \leq \| x \|_p + \| y \|_p$.

Đối với tính scale, ta chọn một vài vector v_2, v_3 và các scalar như α_2, α_5, rồi so sánh hai vế bằng `torch.allclose`:

```
test_vectors = [v2, v3]
scalars_for_scale = [alpha2, alpha5]

for x in test_vectors:
    for a in scalars_for_scale:
        left  = torch.norm(a * x, p=2)           # ||alpha x||_2
        right = torch.abs(a) * torch.norm(x, p=2)# |alpha| ||x||_2
        print("x =", x, "alpha =", a.item())
        print("  ||alpha x||_2   =", left.item(),
              " |alpha| ||x||_2 =", right.item(),
              " equal? =", torch.allclose(left, right))
```

Bất đẳng thức tam giác được kiểm tra bằng cách chọn vài cặp vector (v_2, v_3), (v_2, v_4) và so sánh trực tiếp hai vế của bất đẳng thức:

```
pairs = [(v2, v3), (v2, v4)]
for x, y in pairs:
    lhs = torch.norm(x + y, p=2)                    # ||x + y||_2
    rhs = torch.norm(x, p=2) + torch.norm(y, p=2)   # ||x||_2 + ||y||_2
    print("x =", x, "y =", y)
    print("  ||x + y||_2        =", lhs.item(),
          "  ||x||_2 + ||y||_2 =", rhs.item(),
          "  triangle holds?   =", lhs <= rhs + 1e-6)
```

Ở đây, ℓ_2-norm được dùng như một chuẩn "chuẩn" vì nó liên hệ trực tiếp với khoảng cách Euclid.

Để thể hiện vai trò của norm như một thước đo "khoảng cách", ta xây dựng một vector tham chiếu $x_{\text{ref}} = v_2$ và ba vector khác nhau: một vector "gần", một vector "xa", và một vector chủ yếu khác biệt ở một chiều duy nhất (outlier). Sau đó, ta tính khoảng cách ℓ_2 và ℓ_1 giữa x_{ref} và từng vector:

```
x_ref     = v2
x_close   = v2 + torch.tensor([0.01, -0.02, 0.01])
x_far     = v2 + torch.tensor([5., -5., 3.])
x_outlier = v2.clone()
x_outlier[2] = x_outlier[2] + 50.
```

```
candidates = [("close",   x_close),
              ("far",     x_far),
              ("outlier", x_outlier)]

for name, x in candidates:
    d2 = torch.norm(x_ref - x, p=2)  # khoảng cách Euclid
    d1 = torch.norm(x_ref - x, p=1)  # khoảng cách Manhattan
    print(f"Distance from ref to {name}:")
    print("  d2 =", d2.item(), "  d1 =", d1.item())
```

Các giá trị này sẽ được so sánh trong phần kết quả để xem vector nào "gần" hay "xa" hơn theo từng kiểu norm, và để minh họa việc ℓ_1 và ℓ_2 phản ứng khác nhau với sai lệch tập trung ở một chiều.

Để minh họa Frobenius norm $\| X \|_F$ trên ma trận, ta sử dụng các ma trận A_1, A_3, A_5. Với mỗi ma trận X, ta tính hai giá trị: Frobenius norm bằng `torch.norm(X)` (mặc định là ℓ_2-norm trên vector hóa) và ℓ_2-norm của vector hóa `X.reshape(-1)` để kiểm tra rằng $\| X \|_F = \| \text{vec}(X) \|_2$:

```
matrices_for_norm = [A1, A3, A5]

for X in matrices_for_norm:
    frob   = torch.norm(X)                    # ||X||_F
    vec_l2 = torch.norm(X.reshape(-1), p=2)   # ||vec(X)||_2
    print("X =\n", X)
    print("  ||X||_F      =", frob.item())
    print("  ||vec(X)||_2 =", vec_l2.item())
```

Ta cũng kiểm tra tính scale trên ma trận bằng cách nhân một scalar α_2 hoặc α_5:

```
for X in matrices_for_norm:
    for a in [alpha2, alpha5]:
        left  = torch.norm(a * X)                 # ||alpha X||_F
        right = torch.abs(a) * torch.norm(X)      # |alpha| ||X||_F
        print("X shape =", X.shape, "alpha =", a.item())
        print("  ||alpha X||_F      =", left.item(),
              " |alpha| ||X||_F    =", right.item(),
              " equal? =", torch.allclose(left, right))
```

Cuối cùng, để kết nối norm với khái niệm lỗi trong học máy, ta xem v_2 là ground truth y, và tạo hai vector dự đoán $\hat{y}_1$ (tương đối chính xác) và $\hat{y}_2$ (sai nhiều hơn). Ta tính vector sai số $e_k = \hat{y}_k - y$ và lấy ℓ_2-norm (tương tự RMSE nhưng không chia cho số chiều) và ℓ_1-norm (tương tự MAE nhưng không chia):

```
y_true = v2
y_hat1 = v2 + torch.tensor([0.1, -0.1, 0.05])
y_hat2 = v2 + torch.tensor([1.5, -1.0, -2.0])

for name, y_hat in [("y_hat1", y_hat1), ("y_hat2", y_hat2)]:
    e    = y_hat - y_true
    e_l2 = torch.norm(e, p=2)
    e_l1 = torch.norm(e, p=1)
    print(name, "prediction error:")
    print("  e       =", e)
    print("  ||e||_2 =", e_l2.item(), "  ||e||_1 =", e_l1.item())
```

Các giá trị norm này sẽ được sử dụng trong phần kết quả để so sánh "mức độ sai" giữa hai mô hình dự đoán, qua đó minh họa cách ℓ_2-norm và ℓ_1-norm được sử dụng như các thước đo lỗi phổ biến (tương ứng với loss dạng MSE và MAE trong thực tế).

4.3.3.b. Kết quả thử nghiệm

Khi tính các norm trên nhóm vector thử nghiệm v_2, v_3, v_5 và $v_{\text{outlier}} = [1,2,100]$, ta quan sát được rằng với mỗi vector x, các giá trị $\| x \|_2$, $\| x \|_1$và $\| x \|_3$ đều là vô hướng dương (scalar, tensor 0D), đúng với định nghĩa norm là một hàm $\|\cdot\|: \mathbb{R}^n \to \mathbb{R}_{\geq 0}$. Trên các vector "vừa phải" như v_2, v_3, ta thấy $\| x \|_2$và $\| x \|_1$ có độ lớn cùng bậc, còn $\| x \|_3$ thường nằm giữa hai giá trị này. Tuy nhiên với vector outlier có một phần tử rất lớn, cả $\| x \|_2$lẫn $\| x \|_3$ đều bị chi phối mạnh bởi thành phần lớn đó (do xuất hiện lũy thừa bậc cao), trong khi $\| x \|_1 = \sum_i | x_i |$ phản ánh rõ hơn tổng độ lớn trên toàn bộ các thành phần. Điều này phù hợp với trực giác: norm $\ell_2$nhấn mạnh các phần tử lớn, còn norm ℓ_1"cộng đều" độ lớn ở mọi chiều, ít bị đè bởi một outlier đơn lẻ.

Khi kiểm tra tính đồng biến theo scale, với mỗi vector x(ví dụ v_2, v_3) và mỗi vô hướng α(chẳng hạn α_2, α_5), hai vế $\| \alpha x \|_2$ và $| \alpha | \| x \|_2$ luôn cho kết quả trùng khớp (so sánh bằng `torch.allclose` đều trả về `True`). Thực nghiệm này xác nhận đúng tính chất cơ bản của norm

$$\| \alpha x \|_p = | \alpha | \| x \|_p.$$

Đối với bất đẳng thức tam giác, với các cặp vector $(x, y) = (v_2, v_3)$ và (v_2, v_4), ta luôn thu được giá trị $\| x + y \|_2$ nhỏ hơn hoặc bằng $\| x \|_2 + \| y \|_2$ (sai khác chỉ ở mức sai số số thực rất nhỏ). Điều này xác nhận norm ℓ_2 thỏa mãn bất đẳng thức tam giác

$$\| x + y \|_2 \leq \| x \|_2 + \| y \|_2,$$

và do đó thực sự là một norm đúng nghĩa theo lý thuyết.

Khi xem norm như một thước đo "khoảng cách", ta chọn $x_{\text{ref}} = v_2$ làm vector tham chiếu và xây dựng ba vector khác $x_{\text{close}}, x_{\text{far}}, x_{\text{outlier}}$. Các giá trị $d_2(x_{\text{ref}}, x) = \| x_{\text{ref}} - x \|_2$ và $d_1(x_{\text{ref}}, x) = \| x_{\text{ref}} - x \|_1$ đều là số dương và tăng dần khi chuyển từ "close" → "far" → "outlier". Cụ thể, khoảng cách tới x_{close} nhỏ nhất, phản ánh việc ta chỉ thêm nhiễu rất nhỏ; khoảng cách tới x_{far} lớn hơn rõ rệt do sai lệch trên nhiều chiều; còn khoảng cách tới x_{outlier} là lớn nhất vì sai lệch cực lớn ở một chiều duy nhất. Thực nghiệm này cho thấy rõ ràng norm có thể được dùng để định nghĩa metric

$$d_p(x, y) = \| x - y \|_p$$

và đo được "mức độ gần/xa" giữa các vector trên cùng một không gian đặc trưng, tùy theo lựa chọn p.

Đối với ma trận, khi tính `torch.norm(X)` trên các ma trận A_1, A_3, A_5 và so sánh với ℓ_2-norm của vector hóa `X.reshape(-1)`, ta luôn thu được

$$\| X \|_F = \| \text{vec}(X) \|_2,$$

trong đó $\| X \|_F$ là Frobenius norm. Các giá trị in ra cho thấy hai vế trùng nhau tới sai số số học, xác nhận rằng Frobenius norm trên ma trận đúng là norm ℓ_2 của vector chứa toàn bộ phần tử của ma trận. Khi kiểm tra tính scale với các scalar α, hai vế $\| \alpha X \|_F$ và $| \alpha | \ \| X \|_F$ cũng luôn bằng nhau trong thực nghiệm, cho thấy Frobenius norm cũng thỏa mãn đầy đủ các tính chất của một norm: không âm, chỉ bằng 0 khi ma trận bằng 0, đồng biến theo scale và (theo lý thuyết) thỏa bất đẳng thức tam giác, dù ở đây ta chủ yếu kiểm tra trực tiếp tính scale và mối quan hệ với vector hóa.

Cuối cùng, khi áp dụng norm để đo lỗi dự đoán, ta xem v_2 là ground truth y và xây dựng hai vector dự đoán $\hat{y}_1$ (sai số nhỏ) và $\hat{y}_2$ (sai số lớn hơn). Với mỗi dự đoán, ta tính vector lỗi $e_k = \hat{y}_k - y$ rồi lấy $\| e_k \|_2$ và $\| e_k \|_1$. Kết quả cho thấy $\| e_1 \|_2 < \| e_2 \|_2$ và $\| e_1 \|_1 < \| e_2 \|_1$, tức là mô hình $\hat{y}_1$ thực sự "gần" ground truth hơn $\hat{y}_2$ theo cả hai tiêu chí. Về mặt trực giác, $\| e \|_2$ giống như một dạng RMSE (bỏ bước bình phương và trung bình), nhấn mạnh mạnh hơn các sai số lớn, trong khi $\| e \|_1$ giống như MAE, cộng tuyến tính độ lớn sai số trên từng chiều. Nhờ đó, cụm C cho thấy rõ vai trò của norm không chỉ trong việc đo "độ dài" của vector hoặc ma trận, mà còn là công cụ trung tâm để định nghĩa khoảng cách, đo lỗi và thiết kế hàm mất mát trong các bài toán tối ưu hóa và học sâu.

Chương V. ĐÁNH GIÁ VÀ HƯỚNG PHÁT TRIỂN

5.1. Kết Luận và Đánh giá:

Các khái niệm từ 2.1 đến 2.11 tạo thành một mạch lý thuyết tương đối khép kín và nhất quán: từ những viên gạch nhỏ nhất là scalar, ta xây dựng lên vector để mô tả trạng thái nhiều chiều, rồi tới ma trận và tensor để tổ chức cả một tập lớn vector theo cấu trúc. Nhờ hệ thống này, rất nhiều dạng dữ liệu trong khoa học máy tính có thể được mô tả thống nhất: tọa độ hình học, tập đặc trưng của một mẫu dữ liệu, ảnh nhiều kênh, chuỗi thời gian nhiều biến, cho tới cả batch dữ liệu trong deep learning đều được gom vào khung "vector–ma trận–tensor". Điều này không chỉ giúp ta có ngôn ngữ chung để trình bày bài toán, mà còn cho phép tái sử dụng cùng một tập công cụ toán học cho các bối cảnh rất khác nhau.

Trên nền các đối tượng đó, các phép toán như tensor arithmetic, reduction và non-reduction sum đóng vai trò là "ngữ pháp thao tác" cơ bản. Reduction cho phép ta rút gọn bớt chiều, tính các thống kê như tổng, trung bình, cực đại trên một trục, từ đó chắt lọc thông tin quan trọng hơn trong dữ liệu. Non-reduction sum và các phép cộng element-wise lại cho phép gộp nhiều nguồn thông tin mà không làm mất cấu trúc, điều này xuất hiện rất nhiều trong các kiến trúc mạng nơ-ron hiện đại (residual connection, skip connection…). Khi đi vào tầng tính toán cốt lõi, dot product, matrix–vector product và matrix–matrix product là ba phép nhân xuất hiện ở hầu như mọi nơi: từ việc đo độ tương đồng giữa hai vectơ đặc trưng, chiếu dữ liệu sang không gian mới, cho đến việc tính toán đầu ra của một lớp fully connected hay áp dụng cùng một biến đổi tuyến tính cho cả batch dữ liệu. Các bài tập về Manhattan distance hay về thứ tự nhân ma trận (AB)C

so với A(BC) minh họa rất rõ ràng rằng đại số tuyến tính không chỉ là ngôn ngữ mô tả, mà còn ảnh hưởng trực tiếp đến chi phí bộ nhớ và thời gian chạy của chương trình trên máy thật.

Cuối cùng, phần norms như $\|x\|_1$, $\|x\|_2$ và $\|A\|_F$ giúp ta gắn khung hình học cho toàn bộ không gian này: ta có thể đo độ lớn của một vectơ, đo khoảng cách giữa hai điểm, và đo "kích thước" của cả một ma trận trọng số. Norm ℓ_2 cho ta độ dài Euclid quen thuộc, đóng vai trò trung tâm trong rất nhiều thuật toán tối ưu và regularization kiểu L2. Norm ℓ_1, với hình học "góc cạnh" hơn, khuyến khích nghiệm thưa, giúp mô hình tự động loại bỏ bớt thành phần không cần thiết. Frobenius norm là phiên bản ma trận của ℓ_2, dùng để so sánh hai ma trận, đánh giá độ lớn tổng thể của một lớp trọng số, hoặc làm penalty cấp-lớp. Việc hiểu được ý nghĩa và tác dụng của từng loại norm giúp ta nhìn rõ hơn cách các kỹ thuật regularization thực sự tác động lên không gian tham số. Nhìn chung, cụm lý thuyết 2.1–2.11 cung cấp bộ khung toán học để ta hiểu dữ liệu, hiểu phép biến đổi và hiểu cách "đo lường" mọi thứ trong không gian đó, từ đó làm nền vững chắc cho các chương sau về thuật toán và mô hình.

5.2. Hướng phát triển:

Tuy các nội dung từ 2.1 đến 2.11 tập trung vào thế giới tuyến tính, chúng không phải là điểm kết thúc mà là xuất phát điểm cho nhiều hướng phát triển sâu hơn. Một hướng tự nhiên là mở rộng từ ma trận sang tensor bậc cao trong các bài toán thực tế phức tạp, chẳng hạn như xử lý video (thời gian × chiều cao × chiều rộng × kênh), mô hình hóa dữ liệu không gian–thời gian hoặc kết hợp nhiều modality (hình ảnh, âm thanh, văn bản) trong cùng một mô hình. Khi đó, khái niệm về rank của tensor, phân rã tensor, cũng như các dạng norm và phép nhân chuyên biệt cho tensor trở nên quan trọng hơn rất nhiều so với trường hợp ma trận đơn thuần.

Một hướng phát triển khác là kết nối sâu hơn giữa đại số tuyến tính và tối ưu hóa. Từ các phép nhân ma trận và norms, ta tiến tới các phân rã quan trọng như eigen decomposition, singular value decomposition (SVD), PCA, hay spectral norm của ma trận. Những công cụ này giúp phân tích cấu trúc nội tại của dữ liệu và của chính mô hình, ví dụ đánh giá độ "nhạy" của mạng nơ-ron, khả năng khái quát, hay thiết kế regularization tinh vi hơn dựa trên giá trị riêng hoặc singular value. Đồng thời, các thuật toán tối ưu dựa trên gradient (như SGD, Adam) cũng được xây dựng trực tiếp trên không gian vector, ma trận và tensor, nên việc hiểu kỹ những khái niệm nền tảng sẽ giúp ta tiếp cận được các chủ đề như tối ưu lồi, điều kiện hội tụ hay phân tích độ phức tạp một cách hệ thống hơn.

Cuối cùng, khi bước sang các kiến trúc phi tuyến mạnh như mạng sâu nhiều lớp, attention mechanism, graph neural networks hay các mô hình sequence-to-sequence, phần tuyến tính $Ax + b$ vẫn xuất hiện trong gần như mọi lớp cơ bản (projection, linear layer, query–key–value, message passing trên đồ thị…). Điều này có nghĩa là việc nắm vững tuyến tính không chỉ giúp hiểu những mô hình "cổ điển" như hồi quy tuyến tính, mà còn là chìa khóa để giải thích và debug các kiến trúc phức tạp hiện đại. Trong tương lai, ta có thể tiếp tục mở rộng theo cả hai hướng: một mặt đào sâu vào các công cụ đại số tuyến tính nâng cao (phân rã, norm phức tạp, toán tử tuyến tính), mặt khác kết nối với các lĩnh vực khác như giải tích số, tính toán song song và tối ưu trên phần cứng để triển khai những mô hình lớn hơn, hiệu quả hơn mà vẫn dựa trên cùng bộ khái niệm nền tảng đã được xây dựng trong chương này.

TÀI LIỆU THAM KHẢO

[1] I. Goodfellow, Y. Bengio, và A. Courville, *Deep Learning*, MIT Press, 2016.

[2] G. H. Golub và C. F. Van Loan, *Matrix Computations (4th Edition)*, Johns Hopkins University Press, 2012.

[3] S. Boyd và L. Vandenberghe, *Introduction to Applied Linear Algebra*, Cambridge University Press, 2018.

[4] G. Strang, *Linear Algebra and Learning from Data*, Wellesley-Cambridge Press, 2019.

[5] D. P. Kingma và J. Ba, "Adam: A method for stochastic optimization," *ICLR*, 2015.

[6] A. Zhang, Z. C. Lipton, M. Li, và A. J. Smola, *Dive into Deep Learning*, bản trực tuyến tại địa chỉ: https://d2l.ai/chapter_preliminaries/linear-algebra.html

LINK GITHUB

https://github.com/Quang1402/linear_algebra_demo.git