ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

Cấu trúc dữ liệu và giải thuật - CO2003

Bài tập lớn 1

DANH SÁCH VÀ MẠNG NƠRON NHIỀU LỚP

TP. HỒ CHÍ MINH, THÁNG 09/2024

ĐẶC TẢ BÀI TẬP LỚN

Phiên bản 1.0

1 Giới thiệu

1.1 Mục tiêu và nhiệm vụ

Mục tiêu của các bài tập lớn (BTL) trong môn học này là nhằm giúp sinh viên phát triển được các cấu trúc dữ liệu và sử dụng chúng vào để tạo ra các ứng dụng nào đó. Trong học kỳ này, ứng dụng được chọn làm chủ đề chính cho cả ba bài tập lớn là **mạng nơron và học sâu**; đây cũng là chủ đề có nhiều ứng dụng trong phân tích dữ liệu. Cả ba bài tập lớn này làm về cùng một chủ đề và có tính thừa kế nhau; nghĩa là, Bài tập lớn 3 thừa kế mã nguồn của BTL-2 và BTL-1, và BTL-2 thừa kế từ BTL-1. Do đó, sinh cần lưu ý để đảm bảo tính liên thông này. Nhiệm vụ của các BTL được trình bày như sau đây:

1. Bài tập lớn 1
    - **Phát triển** cấu trúc dữ liệu danh sách; cụ thể là, dánh sách dựa vào array và dựa vào liên kết (hai lớp **XArrayList** và **DLinkedList** trong mã nguồn được cung cấp).
    - **Sử dụng** được thư viện **xtensor** để thực hiện nhiều tác vụ trên mảng nhiều chiều (còn được gọi là tensor). Thư viện này đã được download và được cung cấp kèm theo trong mã nguồn được cung cấp.
    - **Sử dụng** các danh sách đã phát triển ở trên để xây dựng một thư viện các lớp giúp người dùng tạo ra và sử dụng mạng nơron truyền thẳng nhiều lớp (**M**ulti**L**ayer **P**erceptron, **MLP**). Thư viện về **MLP** gồm nhiều hàm và tính năng, BTL-1 chỉ quan tâm đến các lớp và phương thức để hỗ trợ hai vấn đề sau:
        (a) Tạo tập dữ liệu và chia tách dữ liệu để sẵn sàng cho việc xử lý bởi **MLP**.
        (b) Tạo các lớp và phương thức để hỗ trợ việc suy diễn (inference) bởi mạng **MLP**.
2. Bài tập lớn 2 (*sẽ có tài liệu mô tả riêng theo sau*)
    - **Phát triển** các cấu trúc dữ liệu về bảng băm (hash-table) và cấu trúc đống (heap). cụ thể là hai lớp **XHashMap** và **Heap** trong mã nguồn sẽ được cung cấp).
    - **Sử dụng** các cấu trúc dữ liệu hash-table và heap để hiện thực các lớp và phương thức nhằm hỗ trợ cho người dùng **huấn luyện (training** mạng nơron. Cụ thể, có hai vấn đề được quan tâm phát triển cho mạng nơron, đó là:
        (a) Đưa tính năng huấn luyện mạng nơron.

(b) Ứng dụng dụng mạng nơron **MLP** để thực hiện các bài toán phân tích dữ liệu dạng phân loại và hồi quy.

3. Bài tập lớn 3 (*sẽ có tài liệu mô tả riêng theo sau*)
   (a) **Phát triển** cấu trúc dữ liệu đồ thị; cụ thể là cấu trúc dữ liệu **đồ thi có hướng** và các giải thuật trên đồ thị.
   (b) **Sử dụng** cấu trúc dữ liệu đồ thị để xây dựng thư viện về **đồ thị tính toán (computational graph)** trong lĩnh vực học sâu. Đồ thị tính toán trong lĩnh vực học sâu yêu cầu khá nhiều nút tính toán khác nhau, nhiều nút tính toán trong đó yêu cầu phải được tăng tốc bằng các card đồ hoạ (GPU) thì mới nhanh được. Do đó, BTL-3 chỉ quan tâm đến một số nút cơ bản để sinh viên hiểu về khác niệm và nguyên tắc tính toán trong học sâu.

1.2 Phương pháp tiến hành

1. Chuẩn bị: **Download** và **tìm hiểu** mã nguồn được cung cấp. **Lưu ý**, sinh viên cần biên dịch mã nguồn trong các BTL với **C++17** là bắt buộc. Bộ biên dịch, nhóm thực hiện đã kiểm tra với **g++**; khuyến nghị sinh viên dựng môi trường để sử dụng **g++**.
2. **Sử dụng** thư viện **xtensor**: Kiểm tra trang môn học để tải và chạy các demo/hướng dẫn về việc sử dụng thư viện này.
3. **Phát triển** danh sách: Hiện thực lớp **XArrayList** và **DLinkedList** trong thư mục **/include/list**
4. **Phát triển và sử dụng** thư viện về **MLP**:
   (a) Kiểm tra trang môn học để lấy tài liệu hướng dẫn về mạng nơron **MLP**
   (b) Hiện thực các lớp trong thư mục **/include/ann**.
5. **Chạy thử**: Kiểm tra chương trình phải qua được các testcases mẫu được cung cấp.
6. **Nộp bài**: Phải nộp bài trên hệ thống trước deadline.

1.3 Chuẩn đầu ra của BTL-1

Sau khi hoàn thành bài tập lớn này, sinh viên ôn lại và sử dụng thành thục:

- **Sử dụng** được ngôn ngữ lập trình C/C++ ở mức nâng cao.
- **Sử dụng** được thư viện lập trình có sẵn, thư viện **xtensor**.
- **Phát triển** được cấu trúc dữ liệu danh sách, với hai phiên bản, dựa trên mảng và dựa vào liên kết.

- **Lựa chọn và sử dụng** được danh sách vào phát triển thư viện mạng nơron truyền thẳng nhiều lớp (MultiLayer Percepton, **MLP**).

1.4 Thành phần điểm và phương pháp chấm bài

Ba nội dung được chấm điểm trong BTL-1 được trình bày sau đây. **Lưu ý**: sinh viên cần hiện thực theo thứ tự được liệt kê. Vì các bài tập lớn có liên thông nhau, nên nếu sinh viên không thực hiện kịp một số nội dung, ví dụ như Nội dung 3 (hay cả 2 và 3, **Nội dung 1 là bắt buộc hoàn thành**) ở sau đây thì sang BTL-2 sinh viên cũng phải thực hiện chúng thì mới có thể **biên dịch và chạy thành công** các BTL theo sau.

Trong trường hợp sinh viên dời nội dung 2 và 3 sang BTL-2 thì chỉ có thể nhận được 50% điểm số của các nội dung đó.

1. Hiện thực danh sách: (50% **điểm**); gồm tập tin sau đây trong thư mục **/include/list**
   - **XArrayList.h**
   - **DLinkedList.h**
2. Hiện thực các lớp về tập dữ liệu và tải dữ liệu: (30% **điểm**); gồm các tập tin sau đây trong thư mục **/include/ann**:
   - **dataset.h**
   - **dataloader.h**
3. Hiện thực các lớp về tập dữ liệu và tải dữ liệu: (30% **điểm**); gồm các tập tin khác trong thư mục **/include/ann**.

Sinh viên cần chạy đánh giá với các testcases được công bố mẫu. Bài tập lớn sau khi nộp lên hệ thống sẽ được chấm điểm theo các testcases chưa công bố.

2 Hướng dẫn

2.1 Về cấu trúc dữ liệu danh sách

2.1.1 Phương pháp thiết kế của các cấu trúc dữ liệu

Trong môn học này các cấu trúc dữ liệu được thiết kế theo mẫu nhất quán; đó là,

1. Với mỗi cấu trúc dữ liệu sẽ có một hoặc vài lớp trừu tượng định nghĩa về các tác vụ sẽ được hỗ trợ bởi cấu trúc dữ liệu. Trong trường hợp danh sách, đó là lớp **IList** trong thư mục **/include/list**.
2. Trong thực tế, các cấu trúc dữ liệu sẽ chứa được các phần tử **có kiểu bất kỳ**, ví dụ như **đối tượng sinh viên**, **con trỏ đến đối tượng sinh viên**, hay chỉ là một con số kiểu **int**. Do đó, tất cả các cấu trúc dữ liệu trong môn học điều đã sử dụng **template** (trong C++) để tham số hoá kiểu phần tử.
3. Tất cả các cấu trúc dữ liệu điều được thiết kế để che dấu dữ liệu và các chi tiết ở mức cao; cũng như tách bạch hai vai trò, đó là, **người phát triển thư viện** và **người sử dụng thư viện**. Cũng chính vì vậy, tất cả các cấu trúc dữ liệu, kể cả danh sách, đã được bổ sung tính năng duyệt qua các phần tử (tính năng **iterator**) để thuận tiện cho việc duyệt qua phần tử từ phía người dùng.

2.1.2 Tổng quan về cấu trúc dữ liệu danh sách

Cấu trúc dữ liệu danh sách trong BTL này được thiết kế gồm các lớp như sau:

- Lớp `IList`, xem Hình 1: lớp này định nghĩa một danh mục các APIs được hỗ trợ bởi danh sách; bất kể là hiện thực bởi array hay bởi liên kết cũng sẽ phải hỗ trợ các APIs trong `IList`. `IList` là lớp cha của hai lớp `XArrayList` và `DLinkedList`. Một số ý chi tiết:
  - `IList` sử dụng **template** để tham số hoá kiểu dữ liệu phần tử, và cho phép chứa trong danh sách với kiểu bất kỳ.
  - Tất cả các APIs trong `IList` để ở dạng "pure virtual method"; nghĩa là các lớp kế thừa từ `IList` cần phải override tất cả các phương thức này và phương thức dạng này sẽ hỗ trợ liên kết động (tính đa hình).
- Lớp `XArrayList` và `DLinkedList`: được thừa kế từ `IList`, và sẽ hiện thực tất cả các APIs được định nghĩa trong `IList`, bằng cách sử dụng array (như trong `XArrayList`) và sử dụng liên kết (như trong `DLinkedList`).

Bên dưới là mô tả cho từng pure virtual method của `IList`:

- `virtual ~IList() {};`
  - Destructor ảo, được dùng để đảm bảo rằng destructor của các lớp con được gọi khi xóa đối tượng thông qua con trỏ lớp cơ sở.
- `virtual void add(T e) = 0;`

```cpp
template<typename T>
class IList {
public:
virtual ~IList(){};
    virtual void    add(T e)=0;
    virtual void    add(int index, T e)=0;
    virtual T       removeAt(int index)=0;
    virtual bool    removeItem(T item, void (*removeItemData)(T)=0)=0;
    virtual bool    empty()=0;
    virtual int     size()=0;
    virtual void    clear()=0;
    virtual int     indexOf(T item)=0;
    virtual string  toString(string (*item2str)(T&)=0 )=0;
};
```

Hình 1: `IList<T>`: Lớp trừu tượng định nghĩa APIs cho danh sách.

  - Thêm phần tử `e` vào cuối danh sách.
  - **Tham số**: `T e` — phần tử cần thêm.

- `virtual void add(int index, T e) = 0;`
  - Chèn phần tử `e` vào vị trí `index` trong danh sách.
  - **Tham số**:
    - `int index` — vị trí muốn chèn phần tử.
    - `T e` — phần tử cần chèn.

- `virtual T removeAt(int index) = 0;`
  - **Mô tả**: Xóa và trả về phần tử tại vị trí `index`.
  - **Tham số**: `int index` — vị trí của phần tử cần xóa.
  - **Giá trị trả về**: Trả về phần tử tại vị trí `index`.
  - **Ngoại lệ**: Ném `std::out_of_range` nếu `index` không hợp lệ.

- `virtual bool removeItem(T item, void (*removeItemData)(T) = 0) = 0;`
  - **Mô tả**: Xóa phần tử `item` được lưu trữ trong danh sách.
  - **Tham số**:
    - `T item` — phần tử cần xóa.
    - `void (*removeItemData)(T) = 0` — con trỏ hàm (mặc định là NULL) được gọi để xóa dữ liệu của phần tử trong danh sách. Điều này cần thiết vì không xác định được kiểu `T` có phải là con trỏ hay không.
  - **Giá trị trả về**: `true` nếu phần tử `item` tồn tại và được xóa; ngược lại, `false`.

- `virtual bool empty() = 0;`
  - **Mô tả**: Kiểm tra xem danh sách có rỗng không.

  - **Giá trị trả về**: `true` nếu danh sách rỗng; ngược lại, `false`.

- `virtual int size() = 0;`
  - **Mô tả**: Trả về số lượng phần tử hiện có trong danh sách.
  - **Giá trị trả về**: Số lượng phần tử trong danh sách.

- `virtual void clear() = 0;`
  - **Mô tả**: Xóa tất cả các phần tử trong danh sách, đưa danh sách về trạng thái ban đầu.

- `virtual T& get(int index) = 0;`
  - **Mô tả**: Trả về tham chiếu đến phần tử tại vị trí `index`.
  - **Tham số**: `int index` — vị trí của phần tử cần lấy.
  - **Giá trị trả về**: Tham chiếu đến phần tử tại vị trí `index`.
  - **Ngoại lệ**: Ném `std::out_of_range` nếu `index` không hợp lệ.

- `virtual int indexOf(T item) = 0;`
  - **Mô tả**: Trả về chỉ số của phần tử `item` đầu tiên tìm thấy trong danh sách.
  - **Tham số**: `T item` — phần tử cần tìm chỉ số.
  - **Giá trị trả về**: Chỉ số của phần tử `item`; trả về `-1` nếu không tìm thấy.

- `virtual bool contains(T item) = 0;`
  - **Mô tả**: Kiểm tra xem danh sách có chứa phần tử `item` hay không.
  - **Tham số**: `T item` — phần tử cần kiểm tra.
  - **Giá trị trả về**: `true` nếu danh sách chứa phần tử `item`; ngược lại, `false`.

- `virtual string toString(string (*item2str)(T&) = 0) = 0;`
  - **Mô tả**: Trả về một chuỗi mô tả danh sách.
  - **Tham số**: `string (*item2str)(T&) = 0` — con trỏ hàm để chuyển đổi từng phần tử trong danh sách thành chuỗi.
  - **Giá trị trả về**: Chuỗi mô tả danh sách.

2.1.3 Danh sách đặc (Array List)

`XArrayList<T>` (xem Hình 2) là một phiên bản hiện thực danh sách sử dụng mảng (array) để chứa các phần tử, kiểu `T`. Về nguyên tắc, `XArrayList<T>` phải chủ động duy trì một mảng đủ lớn để chứa các phần tử trong nó. Việc biến động phần tử chỉ liên quan đến APIs như `add`, `remove` và `removeItem`; do đó, khi hiện thực các APIs cần phải kiểm tra kích thước mảng đang có để đảm bảo đủ không gian lưu trữ các phần tử.

Ngoài các phương thức như trong Hình 2 để hiện các APIs có trong `IList`, `XArrayList<T>` cũng còn cần các phương thức hỗ trợ khác, xem trong tập tin **XArrayList.h**, trong thư mục **/include/list**.

```
template<typename T>
class XArrayList: public IList<T> {
protected:
    T*  data;
    int capacity;
    int count;
public:
    XArrayList(int capacity=10);
    ~XArrayList();

public:
    //Inherit from IList: BEGIN
    void    add(T e);
    void    add(int index, T e);
    T       removeAt(int index);
    bool    removeItem(T item, void (*removeItemData)(T)=0);
    bool    empty();
    int      size();
    void    clear();
    T&      get(int index);
    int      indexOf(T item);
    bool    contains(T item);
    string  toString(string (*item2str)(T&)=0 );
    //Inherit from IList: BEGIN
};

```

Hình 2: `XArrayList<T>`: Danh sách được hiện thực bằng mảng (array)

1. **Các thuộc tính:**
    - `T* data`: Mảng động lưu trữ các phần tử của danh sách.
    - `int capacity`: Sức chứa hiện tại của mảng động `data`.
    - `int count`: Số lượng phần tử hiện có trong danh sách.
2. **Hàm khởi tạo và hàm hủy:**
    - `XArrayList(int capacity)`: Hàm khởi tạo cho phép khởi tạo danh sách với sức chứa ban đầu
    - `~XArrayList()`: Hàm hủy, giải phóng bộ nhớ đã cấp phát cho mảng động `data` và các phần tử.
3. **Các phương thức:**
    - **void add(T e)**

  - **Chức năng**: Thêm một phần tử e vào cuối danh sách.
  - **Ngoại lệ**: Không có.
- **void add(int index, T e)**
  - **Chức năng**: Thêm phần tử e vào vị trí chỉ định *index* trong danh sách.
  - **Ngoại lệ**: Nếu *index* không hợp lệ (ngoài đoạn $[0, \text{count}]$), ném ra ngoại lệ `out_of_range("Index is out of range!")`.
- **T removeAt(int index)**
  - **Chức năng**: Xóa phần tử tại vị trí *index* và trả về phần tử bị xóa.
  - **Ngoại lệ**: Nếu *index* không hợp lệ (ngoài đoạn $[0, \text{count} - 1]$), ném ra ngoại lệ `out_of_range("Index is out of range!")`.
- **bool removeItem(T item, void (*removeItemData)(T) = 0)**
  - **Chức năng**: Xóa phần tử đầu tiên trong danh sách có giá trị bằng *item*.
  - **Ngoại lệ**: Không có.
- **bool empty()**
  - **Chức năng**: Kiểm tra xem danh sách có rỗng hay không.
  - **Ngoại lệ**: Không có.
- **int size()**
  - **Chức năng**: Trả về số lượng phần tử hiện có trong danh sách.
  - **Ngoại lệ**: Không có.
- **void clear()**
  - **Chức năng**: Xóa tất cả các phần tử trong danh sách và đặt danh sách về trạng thái ban đầu.
  - **Ngoại lệ**: Không có.
- **T& get(int index)**
  - **Chức năng**: Trả về tham chiếu đến phần tử tại vị trí *index*.
  - **Ngoại lệ**: Nếu *index* không hợp lệ (ngoài đoạn $[0, \text{count} - 1]$), ném ra ngoại lệ `out_of_range("Index is out of range!")`.
- **int indexOf(T item)**
  - **Chức năng**: Trả về chỉ số của phần tử đầu tiên có giá trị bằng *item* trong danh sách, hoặc -1 nếu không tìm thấy.
  - **Ngoại lệ**: Không có.
- **bool contains(T item)**
  - **Chức năng**: Kiểm tra xem danh sách có chứa phần tử *item* hay không.
  - **Ngoại lệ**: Không có.

- **string toString(string (*item2str)(T&) = 0)**
  - **Chức năng**: Trả về chuỗi biểu diễn các phần tử trong danh sách.
  - **Ngoại lệ**: Không có.

2.1.4 Danh sách liên kết đôi (Doubly Linked List)

`DLinkedList<T>` (xem Hình 3) là một phiên bản hiện thực danh sách sử dụng hai liên kết **next** và **prev**. Về phía người sử dụng của danh sách, họ không quan tâm về hiện thực chi tiết bên trong. Do đó, họ cũng không phải nhận thức rằng có liên kết hay không. Để làm được việc này, về nguyên tắc, `DLinkedList<T>` cần có đối tượng "**node**", và đối tượng này sẽ chứa dữ liệu của người dùng, cũng như hai liên kết, một đến phần tử kế tiếp, một đến phần tử theo sau.

Ngoài các phương thức như trong Hình 3 để hiện các APIs có trong `IList`, `DLinkedList<T>` cũng còn cần các phương thức hỗ trợ khác, xem trong tập tin **DLinkedList.h**, trong thư mục **/include/list**.

1. **class Node:**
   - `T data`: Biến kiểu `T`, lưu trữ dữ liệu của node. `T` là kiểu dữ liệu tổng quát (sử dụng template).
   - `Node* next`: Con trỏ trỏ tới node tiếp theo trong danh sách liên kết đôi.
   - `Node* prev`: Con trỏ trỏ tới node trước đó trong danh sách liên kết đôi.
   - `Node(Node* next = 0, Node* prev = 0)`: Constructor mặc định cho phép khởi tạo một node với các con trỏ `next` và `prev` nhận giá trị mặc định là `nullptr`.
   - `Node(T data, Node* next = 0, Node* prev = 0)`: Constructor cho phép khởi tạo một node với dữ liệu `data`, cùng với các con trỏ `next` và `prev`.
2. **Hàm khởi tạo và hàm hủy:**
   - `DLinkedList()`: Khởi tạo một đối tượng danh sách liên kết đôi rỗng.
   - `~DLinkedList()`: Hủy đối tượng danh sách liên kết đôi, giải phóng tất cả các nút trong danh sách.
3. **Các thuộc tính:**
   - `Node* head`: Con trỏ đến đầu danh sách liên kết đôi.
   - `Node* tail`: Con trỏ đến cuối danh sách liên kết đôi.
   - `int count`: Biến số nguyên lưu trữ số lượng node chứa dữ liệu của người dùng trong danh sách.
4. **Các phương thức:**

```
template<class T>
class DLinkedList: public IList<T> {
public:
    class Node; //Forward declaration
protected:
    Node *head;
    Node *tail;
    int count;

public:
    DLinkedList();
    ~DLinkedList();

    //Inherit from IList: BEGIN
    void    add(T e);
    void    add(int index, T e);
    T       removeAt(int index);
    bool    removeItem(T item, void (*removeItemData)(T)=0);
    bool    empty();
    int      size();
    void    clear();
    T&      get(int index);
    int      indexOf(T item);
    bool    contains(T item);
    string  toString(string (*item2str)(T&)=0 );
    //Inherit from IList: END

public:
    class Node{
    public:
        T data;
        Node *next;
        Node *prev;
        friend class DLinkedList<T>;

    public:
        Node(Node* next=0, Node* prev=0);
        Node(T data, Node* next=0, Node* prev=0);
    };
};

```

Hình 3: `DLinkedList<T>`: Danh sách được hiện thực bằng liên kết kép (`next` và `prev`).

- `void add(T e)`
  - **Chức năng:** Thêm phần tử `e` vào cuối danh sách.
  - **Ngoại lệ:** Không có ngoại lệ.
- `void add(int index, T e)`
  - **Chức năng:** Thêm phần tử `e` vào vị trí chỉ định `index` trong danh sách.
  - **Ngoại lệ:** Nếu `index` không hợp lệ (ví dụ: nhỏ hơn 0 hoặc lớn hơn kích thước

danh sách), sẽ ném ra ngoại lệ `std::out_of_range`.

- `T removeAt(int index)`
  - **Chức năng:** Xóa và trả về phần tử tại vị trí chỉ định `index` trong danh sách.
  - **Ngoại lệ:** Nếu `index` không hợp lệ (ví dụ: nhỏ hơn 0 hoặc lớn hơn hoặc bằng kích thước danh sách), sẽ ném ra ngoại lệ `std::out_of_range`.
- `bool removeItem(T item, void (*removeItemData)(T)=0)`
  - **Chức năng:** Xóa phần tử đầu tiên trong danh sách mà có giá trị bằng với `item`. Nếu tìm thấy và xóa thành công, trả về `true`; ngược lại trả về `false`.
  - **Ngoại lệ:** Nếu con trỏ hàm `removeItemData` được cung cấp, hàm này sẽ được gọi để xử lý dữ liệu trước khi xóa. Không có ngoại lệ khác.
- `bool empty()`
  - **Chức năng:** Kiểm tra xem danh sách có rỗng hay không. Trả về `true` nếu rỗng, ngược lại trả về `false`.
  - **Ngoại lệ:** Không có ngoại lệ.
- `int size()`
  - **Chức năng:** Trả về số lượng phần tử hiện tại trong danh sách.
  - **Ngoại lệ:** Không có ngoại lệ.
- `void clear()`
  - **Chức năng:** Xóa tất cả các phần tử trong danh sách, làm cho danh sách trở thành rỗng.
  - **Ngoại lệ:** Không có ngoại lệ.
- `T& get(int index)`
  - **Chức năng:** Trả về tham chiếu đến phần tử tại vị trí chỉ định `index` trong danh sách.
  - **Ngoại lệ:** Nếu `index` không hợp lệ (ví dụ: nhỏ hơn 0 hoặc lớn hơn hoặc bằng kích thước danh sách), sẽ ném ra ngoại lệ `std::out_of_range`.
- `int indexOf(T item)`
  - **Chức năng:** Trả về chỉ số của phần tử đầu tiên có giá trị bằng với `item` trong danh sách. Nếu không tìm thấy, trả về -1.
  - **Ngoại lệ:** Không có ngoại lệ.
- `bool contains(T item)`
  - **Chức năng:** Kiểm tra xem danh sách có chứa phần tử có giá trị bằng với `item` hay không. Trả về `true` nếu có, ngược lại trả về `false`.
  - **Ngoại lệ:** Không có ngoại lệ.

- `string toString(string (*item2str)(T&)=0)`
  - **Chức năng:** Trả về một chuỗi ký tự đại diện cho danh sách. Nếu con trỏ hàm `item2str` được cung cấp, hàm này sẽ được gọi để chuyển đổi mỗi phần tử sang chuỗi ký tự.
  - **Ngoại lệ:** Không có ngoại lệ.

2.2 Các lớp liên quan đến tập dữ liệu và tải dữ liệu

Nội dung hiện thực này liên quan đến hai nhóm các lớp sau đây.

1. Nhóm các lớp hiện thực tập dữ liệu (dataset), nhóm này gồm các lớp sau đây:
   (a) Lớp `Dataset`: trong tập tin **/include/ann/dataset.h**
   (b) Lớp `TensorDataset`: trong tập tin **/include/ann/dataset.h**
   (c) Lớp `ImageFolderDataset`: *lớp này sinh viên sẽ tự phát thảo ý tưởng.* Gợi ý: lớp này là lớp con của `Dataset` và hàm khởi tạo của nó phải nhận vào tên của thư mục chứa dữ liệu và nhãn.
2. Nhóm các lớp hiện thực việc tải dữ liệu (dataloader) từ các tập dữ liệu, nhóm này gồm lớp sau đây:
   (a) Lớp `DataLoader`: trong tập tin **/include/ann/dataloader.h**

2.2.1 Lớp `Dataset`

Để tương thích với việc tải dữ liệu trong `DataLoader` thì bất kỳ dataset nào cũng phải dẫn ra từ `Dataset<DT, LT>`. Tất cả bốn phương thức trong `Dataset<DT, LT>` đều ở dạng "virtual pure method"; do đó, nó phải override trong các lớp dẫn xuất. `Dataset<DT, LT>` được mô tả chi tiết hơn như sau:

1. **Các tham số kiểu `DT` và `LT`:**
   - `DT`: Dữ liệu trước khi nạp vào mạng nơron được mô tả ở dạng số; cụ thể, là array hai hay nhiều chiều. `DT` sẽ là kiểu dữ liệu của phần tử trong array; thông thường là `double` hoặc `float`.
   - `LT`: Mỗi đơn vị dữ liệu quan sát được, còn được gọi là một quan sát hay một mẫu, thường sẽ được gán nhãn. Ví dụ, một ảnh trong bài toán phân loại ảnh, nhãn là tên loại; nhưng máy tính sẽ dùng con số để chỉ loại. `LT` là kiểu của con số biểu thị nhãn.
2. **Các phương thức**:

- Phương thức `len`: phương thức này trả về kích thước tập dữ liệu; nghĩa là có bao nhiêu số mẫu (số quan sát) trong tập dữ liệu. Một số ví dụ:
  - Nếu tập dữ liệu là một ma trận, thì các hàng là các mẫu dữ liệu, hay còn được gọi là vector đặc trưng. Do đó, kích thước của tập sẽ là số hàng của ma trận.
  - Nếu tập dữ liệu là một ma trận nhiều chiều, thì kích thước của chiều đầu tiên (chiều số 0) là số mẫu trong tập. Đây cũng là trường hợp của `TensorDataset` được trình bày sau đây.
  - Nếu tập dữ liệu là số tập tin trong một thư mục, thì số tập tin là kích thước của tập dữ liệu. Đây cũng là trường hợp của `ImageFolderDataset`.
- Phương thức `getitem`: phương thức này trả về một đối tượng có kiểu `DataLabel`. Lớp `DataLabel` được cho sẵn mã nguồn trong tập tin **dataset.h**.
- Phương thức `get_data_shape` và `get_label_shape`: hai phương thức này trả về hình dạng của dữ liệu và nhãn. Xem thêm về ma trận nhiều chiều trong thư viện **xtensor** để hiểu rõ hơn về thuộc tính này.

```
template<typename DType, typename LType>
class Dataset{
private:
public:
    Dataset(){};
    virtual ~Dataset(){};

    virtual int len()=0;
    virtual DataLabel<DType, LType> getitem(int index)=0;
    virtual xt::svector<unsigned long> get_data_shape()=0;
    virtual xt::svector<unsigned long> get_label_shape()=0;

};
```

Hình 4: `Dataset<T>`: Lớp cha của mọi dataset bất kỳ.

2.2.2 Lớp TensorDataset

`TensorDataset<DT, LT>` (xem Hình 5) là lớp con của lớp `Dataset<DT, LT>`; nó biểu diễn các tập dữ liệu **đã ở sẵn** định dạng tensor. Điều này có nghĩa là dữ liệu của `TensorDataset<DT, LT>` có kiểu là `xt::xarray<DT>` và nhãn của tập dữ liệu có kiểu là `xt::xarray<LT>`. Ý nghĩa của các biến thành viên và các phương thức được trình bày theo sau.

1. **Các biến thành viên**:

- `data` và `label`: hai tensor này có kiểu phần tử là `DT` và `LT` tương ứng. Chúng chứa dữ liệu và nhãn trong tập dữ liệu.
- `data_shape` và `label_shape`: hai tensor này chứa hình dạng dữ liệu của các biến `data` và `label` tương ứng. Với thư viện **xtensor** thì hình dạng dữ liệu có kiểu là `xt::svector`.

2. **Hàm khởi tạo**: Hàm khởi tạo `TensorDataset(.,.)` có hai tham số. Sinh viên phải gán lại trong các biến thành viên và lưu giữ hình dạng dữ liệu của nhãn trong các biến tương ứng.
3. **Các phương thức khác được override từ lớp** `Dataset`: Hiện thực các phương thức này theo mô tả trong lớp `Dataset`.

2.2.3 Lớp Dataloader

Mục đích của việc phát triển các lớp dataset và dataloader là để hỗ trợ chạy được đoạn chương trình như trong Hình 6. Đoạn chương trình này có thể hiểu như sau:

- **Dòng 15**: nhằm để tạo tập dữ liệu có kiểu TensorDatset. Tập dữ liệu này nhận các tensor được tạo sẵn ở các dòng trên.
- **Dòng 16**: nhằm để tạo ra một bộ loader cho tập dữ liệu với các thông tin như sau:
  1. **&ds**: loader cho tập dữ liệu Tensor nói ở trên. Lưu ý, nó nhận vào là địa chỉ của loader nhằm để sử dụng năng đa hình.
  2. **batch_size=30**: Tập dữ liệu có tổng cộng `100` mẫu; loader sẽ chia thành các batch có kích thước `30` mẫu. Do đó, được `3` batch và còn dư `10` mẫu. Vì, thông số cuối là `drop_last =false`, nghĩa là `10` còn dư này sẽ được gộp batch cuối cùng.
  3. **shuffle=true**: Nghĩa là **bắt đầu công việc chia thành batch**, loader sẽ **xáo trộn dữ liệu** (**xáo trộn thứ tự các mẫu dữ liệu**) theo thứ tự ngẫu nhiên (phân phối uniform). Nếu thông số này là `false` thì dữ liệu sẽ không được xáo trộn ngẫu nhiên.
  4. **drop_last=false**: như được trình bày theo trên.
- **Dòng 17-19**: các dòng này có ý nghĩa là "cho mỗi batch tải được bởi loader, ta xử lý batch dữ liệu này". Việc xử lý trong dòng `18-19` chỉ là in ra hình dạng của dữ liệu và nhãn trong batch.

Dựa theo ví dụ ứng dụng ở trên và dựa theo cách hiện thực để hỗ trợ cú pháp "foreach" đã được thực hiện trong các lớp `XArrayList` và `DLinkedList` sinh viên nghiên cứu cách hiện thực lớp `DataLoader`. Mã nguồn của lớp này được cho trong Hình 7.

```cpp
template<typename DType, typename LType>
class TensorDataset: public Dataset<DType, LType>{
private:
    xt::xarray<DType> data;
    xt::xarray<LType> label;
    xt::svector<unsigned long> data_shape, label_shape;

public:
    /* TensorDataset:
     * need to initialize:
     * 1. data, label;
     * 2. data_shape, label_shape
    */
    TensorDataset(xt::xarray<DType> data, xt::xarray<LType> label){
        /* TODO: your code is here for the initialization
         */
    }
    /* len():
     *  return the size of dimension 0
    */
    int len(){
        /* TODO: your code is here to return the dataset's length
         */
        return 0; //remove it when complete
    }

    /* getitem:
     * return the data item (of type: DataLabel) that is specified by index
     */
    DataLabel<DType, LType> getitem(int index){
        /* TODO: your code is here
         */
    }

    xt::svector<unsigned long> get_data_shape(){
        /* TODO: your code is here to return data_shape
         */
    }
    xt::svector<unsigned long> get_label_shape(){
        /* TODO: your code is here to return label_shape
         */
    }
};
```

Hình 5: `TensorDataset<DT, LT>`: Lớp biểu diễn dữ liệu tập dữ liệu đã ở định dạng tensor.

2.3 Các lớp về mạng nơron MLP

2.3.1 Tập tin định nghĩa và tập tin hiện thực

Tất cả các lớp được hiện thực trong phần này có mã nguồn nằm ở hai nơi sau đây trong dự án (tương đối so với thư mục dự án).

```cpp
#include <iostream>
#include <iomanip>
#include <sstream>
using namespace std;

#include "ann/funtions.h"
#include "ann/xtensor_lib.h"
#include "ann/dataset.h"
#include "ann/dataloader.h"

int main(int argc, char** argv) {
    int nsamples = 100;
    xt::xarray<double> X = xt::random::randn<double>({nsamples, 10});
    xt::xarray<double> T = xt::random::randn<double>({nsamples, 5});
    TensorDataset<double, double> ds(X, T);
    DataLoader<double, double> loader(&ds, 30, true, false);
    for(auto batch: loader){
        cout <<  shape2str(batch.getData().shape()) << endl;
        cout <<  shape2str(batch.getLabel().shape()) << endl;
    }
    return 0;
}
```

Hình 6: Cách sử dụng `DataLoader`.

- **/include/ann/**: chứa các tập tin định nghĩa (“`.h`“, header).
- **/src/ann/**: chứa các tập tin hiện thực (“.cpp“, source)

2.3.2 Danh mục các lớp cần hiện thực

Danh mục các lớp và các hàm cần hiện thực được cho trong Bảng 1. Sinh viên tham khảo mã nguồn để có chi tiết các hàm cần hiện thực. Tài liệu chi tiết hơn về cách hiện thực này sẽ được bổ sung theo sau.

Bảng 1: Bảng các lớp và các tập cần hiện thực

Lớp	Tập tin header	Tập tin mã nguồn
Layer	Layer.h	Layer.cpp
FCLayer	FCLayer.h	FCLayer.cpp
ReLU	ReLU.h	ReLU.cpp
Softmax	Softmax.h	Softmax.cpp
BaseModel	BaseModel.h	BaseModel.cpp
Hàm bổ sung cho `xtensor`	xtensor_lib.h	xtensor_lib.cpp
Các hàm khác	funtions.h	funtions.cpp

```cpp
template<typename DType, typename LType>
class DataLoader{
public:
private:
    Dataset<DType, LType>* ptr_dataset;
    int batch_size;
    bool shuffle;
    bool drop_last;
    /*TODO: add more member variables to support the iteration*/
public:
    DataLoader(Dataset<DType, LType>* ptr_dataset,
            int batch_size,
            bool shuffle=true,
            bool drop_last=false){
        /*TODO: Add your code to do the initialization */
    }
    virtual ~DataLoader(){}

    //////////////////////////////////////////////////////////////
    // The section for supporting the iteration and for-each to DataLoader
    //
    /// START: Section
    //////////////////////////////////////////////////////////////

    /*TODO: Add your code here to support iteration on batch*/

    //////////////////////////////////////////////////////////////
    // The section for supporting the iteration and for-each to DataLoader
    //
    /// END: Section
    //////////////////////////////////////////////////////////////
};
```

Hình 7: `DataLoader<DT, LT>`: Lớp chia dữ liệu thành bó (batch) và hỗ trợ cú pháp "**foreach batch in the-dataset**".

3 Viết code và biên dịch

3.1 Chiến lược phát triển

BTL-1 gồm ba nội dung có thành phần các cột điểm như đã được trình bày trong Phần 1.4. Sinh viên phải theo tuần từ các đầu việc trong đó mà thực hiện.

Để dễ dàng biên dịch từng phần, sinh viên nên tách thành các dự án nhỏ hơn. Cụ thể, dự án ban đầu chỉ là phát triển danh sách, chỉ bao gồm hai tập tin `XArrayList.h` và `DLinkedList.h`.

Sau khi biên dịch thành công dự án về danh sách ở trên, sinh viên nên tiếp bổ sung dần dần hai công việc còn lại.

3.2 Biên dịch

Sinh viên **Nên** tích hợp code vào một trong các IDE nào đó cảm thấy thuận tiện cho cá nhân, và sử dụng giao diện để biên dịch. Ví dụ, giảng viên đã sử dụng **Apache NetBeans IDE 13** để phát triển.

Nếu cần biên dịch bằng dòng, sinh viên có thể tham khảo dòng lệnh sau đây: **g++ -Iinclude -Isrc -std=c++17 your_main.cpp**

4 Nộp bài

Sinh viên sẽ điền code vào các tập tin được giao và nén lại rồi submit lên hệ thống. Chỉ dẫn nộp bài sẽ được công bố chi tiết sau.

5 Các quy định bổ sung

- Sinh viên phải hoàn thành dự án này một cách độc lập và ngăn chặn người khác sao chép kết quả của mình. Nếu không làm được điều này sẽ dẫn đến hành động kỷ luật vì gian lận học thuật.
- Tất cả các quyết định của giảng viên phụ trách dự án là quyết định cuối cùng.
- Sinh viên không được phép cung cấp testcase sau khi đã chấm điểm nhưng có thể cung cấp thông tin về chiến lược thiết kế testcase và phân phối số lượng sinh viên cho từng testcase.
- Nội dung của dự án sẽ được đồng bộ với một câu hỏi trong kỳ thi có nội dung tương tự.

6 Theo dõi các thay đổi qua các phiên bản

- Ngày 09/09/2024: Version 1.0 được công bố.

——————————HẾT——————————