ECE 449/590 – OOP and Machine Learning
Lecture 11 Tensor Class Design

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

September 28, 2022

Outline

Reading Assignment

- This lecture: Accelerated C++ 9
- Next lecture: Accelerated C++ 13

Outline

Assertion and Exception

Tensor

Tensor Class Design

Enforcing Preconditions

- Class invariants help to enforce preconditions regarding data members for member functions.
  - What about preconditions regarding other function parameters?
- Use comments to document precondition around the function declaration.
  - Comments are in natural languages, which are usually ambiguous when the preconditions are very complicated.
  - Programmers may violate it accidentally even if they follow the instruction.
  - You have to update the comments as you update the function implementation.
- A function may be called many times during execution, some with correct arguments and some without.
  - Compiler/linker are not quite helpful in such cases (as of now).
  - Runtime validations are a must.

Assertion

- A C++ feature that allows the debugger to break your program when a precondition is violated.
  - The program simply prints an error message and exits if a debugger is not presented.
- The program will break at a point depending on the library implementations of assertions.
  - There may exist many implementations of assertions that provide different diagnosis informations.
- You can always use the call stack to navigate to your code that causes the violation before resolving it.
  - The assertion itself may provide useful information on what causes the violation.

The assert Macro

```
#include <assert.h>
date::date(int y, int m, int d):
    year_(y), month_(m), day_(d) {
    assert(valid());
}
```

- You can write your own assertions by using the `assert` macro from the standard header `assert.h` .
  - A macro is a piece of code like a function.
- `assert` takes an expression as the argument and will produce an informative message if the expression evaluates to false.
  - Assume `valid()` returns `true` if the object is valid.
  - If there is a debugger, it will also be triggered.

Exceptions

```
date::date(int y, int m, int d):
    year_(y), month_(m), day_(d) {
    if (!valid()) {
        throw std::runtime_error("Invalid date");
    }
}
```

- The violation of the precondition can be notified by throwing an exception.
  - You can `throw` an object of any type.
  - Though in practice, people design different class types for different reasons of errors.
- `std::runtime_error` is a class type from the standard header `stdexcept` indicating an error at runtime.
  - Recall that `std::runtime_error("Invalid date")` constructs a `std::runtime_error` object with the argument `"Invalid date"`.
- There are other standard exception types.

Exception Handling

```
void some_function() {
    date someday(2019, 2, 29);
    ... // usual business flow
}
bool do_business() {
    try {
        // usual business flow
        some_function();
        another_function();
    }
    catch (std::exception &e) {
        std::cerr << e.what() << std::endl;
        return false;
    }
    return true;
}
```

- Though you are forced to handle exceptions in your program, you don't need to handle them immediately.
  - Improve readability by NOT flooding usual business flows with error handling codes
  - If an exception is not handled, the program will abort.

Returned Error Codes vs Assertions vs Exceptions

- Returned error codes: function may also choose to return error codes to indicate violation of preconditions.
  - Can recover from an error.
  - No enforcement of error handling.
  - Awkward to return error codes from many functions.
  - Mixing code for business logic and error handling will affect readability.
- Assertions
  - No error recovery.
  - Enforce error handling by terminating program.
  - May be turned off to improve performance.
- Exceptions
  - Can recover from an error.
  - Enforce error handling by terminating program if not handled.
  - Prefer centralized error handling to improve readability.
- Choice between exceptions and assertions is a design decision. You need to make trade-offs.

Outline

Tensor

- Multidimensional data structure widely used for machine and deep learning.
  - E.g. a video can be represented as a tensor with 4 dimensions: frame, x, y, color
- Can be treated as generalization of scalars, vectors, and matrices.
  - Which are tensors of 0, 1, and 2 dimensions respectively.
- Performance of many machine and deep algorithms is closed related to efficiency of the underlying tensor operations.
  - Factors affecting efficiency include memory layout, cache architecture, parallel implementation, etc.
  - Let's focus on memory layout for this lecture.

Memory Layout

- The way to store a multidimensional tensor in memory.
- Need to consider many trade-offs.
  - Implementation of tensor operations with some kinds of layouts are more efficient than others.
  - Different libraries and languages may support different kinds of layouts.
  - Converting from one kind of layout to another usually requires to make copy of the data, consuming substantial amount of time even when there is enough memory.

(Contiguous) Row-Major Order

- A widely used memory layout for dense matrices.
- Consider a matrix A with R rows and C columns.
  - Let $A(i,j)$ be the elements on the ith row and jth column.
  - For simplicity we assume 0 based indices, i.e. $i = 0, 1, \ldots, R-1$ and $j = 0, 1, \ldots, C-1$.
- Store A in an array *data* row-by-row:

$$\begin{aligned}
&A(0,0), A(0,1), \ldots, A(0, C-1),\\
&A(1,0), A(1,1), \ldots, A(1, C-1),\\
&\ldots,\\
&A(R-1,0), A(R-1,1), \ldots, A(R-1, C-1)
\end{aligned}$$

  - The array *data* has $R * C$ elements.
  - $A(i,j) = data[i * C + j]$

Contiguous Row-Major Order for Tensors

- Dimension of the tensor A: N
- Shape of A: $s_0, s_1, \ldots, s_{N-1}$
  - Can be stored in an array of N elements.
- One element: $A(i_0, i_1, \ldots, i_{N-1})$
  - $i_k = 0, 1, \ldots, s_k - 1$ for $k = 0, 1, \ldots, N-1$
- Store A in an array *data* as:

$$
\begin{aligned}
&A(0,\ldots,0,0), A(0,\ldots,0,1), \ldots, A(0,\ldots,0,s_{N-1}-1),\\
&A(0,\ldots,1,0), A(0,\ldots,1,1), \ldots, A(0,\ldots,1,s_{N-1}-1),\\
&\ldots,\\
&A(s_0-1,\ldots,s_{N-2}-1,0), A(s_0-1,\ldots,s_{N-2}-1,1), \ldots, A(s_0-1,\ldots,s_{N-2}-1,s_{N-1}-1)
\end{aligned}
$$

  - The array *data* has $s_0 * s_1 * \cdots * s_{N-1}$ elements.
  - $A(i_0, i_1, \ldots, i_{N-1}) = data[i_0 * s_1 * \cdots * s_{N-1} + i_1 * s_2 * \cdots * s_{N-1} + \cdots + i_{N-1}]$.

Passing Tensors from Python to C

```
extern "C" int add_op_param_ndarray(
    program *prog, const char *key,
    int dim, size_t shape[], double data[]);
```

- Use contiguous row-major order.
  - `dim` is the dimension N of the tensor.
  - The `shape` array contains $s_0, \ldots, s_{N-1}$.
  - The `data` array contains the elements of the tensor.
- Some implementation details
  - NumPy `ndarray`s are converted to such format when necessary.
  - As Python GC may release those buffers after this function return, you should make copies of `shape` and `data` arrays.

Passing Tensors from C to Python

```
extern "C" int execute(evaluation *eval,
    int *p_dim, size_t **p_shape, double **p_data);
```

- We make use of pointer-to-pointers to allow Python code to access the shape and data arrays in C code.
- Since Python code will access those two arrays after this function returns, the two arrays need to have lifetimes beyond this function.
- The same function can return a scalar when necessary.
  - Recall scalars are tensors of dimension 0.
  - e.g. for Project 2

What about multidimensional arrays in C?

- There are other methods to support multidimensional arrays in C, e.g.
  - Built-in C multidimensional arrays.
  - Multiple levels of pointer-to-pointers.
- Not easy to work with
  - Need to specify dimension and/or shape at compile time.
  - Need to work with multiple levels of pointers and pointer arithmetics.
- Let's focus on contiguous row-major order and how to manage it in C++.

Outline

Class Invariant

- To represent a tensor in contiguous row-major order, we need,
  - `dim`, `shape`, and `data`.
  - As data members of `tensor` class.
- Class invariant
  - `dim` is a positive integer.
  - The `shape` array should have `dim` elements, which are all positive integers.
  - The `data` array should have `shape[0]`$*$`shape[1]` $*\cdots*$`shape[dim-1]` elements.
- What about scalars?

Updated Class Invariant

- If `dim == 0`, then a scalar is stored in `tensor`.
  - The `shape` array should be empty.
  - The `data` array should have a single element being the scalar.
- If `dim > 0`, then a tensor is stored in `tensor`.
  - The `shape` array should have `dim` elements, which are all positive integers.
  - The `data` array should have `shape[0]`$*$`shape[1]` $*\cdots*$`shape[dim-1]` elements.

The tensor Class

```
class tensor {
public:
    tensor(); // scalar 0
    explicit tensor(double v); // scalar v
    tensor(int dim, size_t shape[], double data[]); // from C
    ...
private:
    std::vector<size_t> shape_;
    std::vector<double> data_;
}; // class tensor
```

- Use ctors to establish class invariant.
  - Use `std::vector` to store arrays.
  - There is no need to store `dim` as it can be obtained as `shape_.size()`.
- Ctors taking a single parameter should usually be declared as `explicit` to prevent implicit conversions that may cause hard-to-debug issues.

Implementing Ctors

```
tensor::tensor():
    data_(1, 0) {
}

tensor::tensor(double v):
    data_(1, v) {
}

tensor::tensor(int dim, size_t shape[], double data[]):
    shape_(shape, shape+dim) {
    // calculate N as shape[0]*shape[1]*...*shape[dim-1]
    ...
    data_.assign(data, data+N);
}
```

- Similar to how we handle `inputs` for Project 2, C arrays can be copied into C++ vectors using ctors or `assign`.

Accessors

```
class tensor {
public:
    ...
    int get_dim() const;

    // scalar only
    double item() const;
    double &item();
    ...
}; // class tensor
```

- Allow users of `tensor` to know the dimension by `get_dim`.
- Scalar can be accessed by `item` – note that two versions are provided.
  - The `const item` will be called with `const tensor` objects, only allowing to read the scalar.
  - The non-`const item` will be called with other `tensor` objects, allowing to read and write the scalar via the reference.

Implementing const and Non-const Members

```
double tensor::item() const {
    assert(shape_.empty());
    return data_[0];
}

double &tensor::item() {
    assert(shape_.empty());
    return data_[0];
}
```

- Although the two function bodies look exactly the same, they are actually different since different `[]` operators are used.
  - We will study how to implement our own `vector` later.
- Use assertions to make sure that the `tensor` object indeed holds a scalar.
  - You may choose to use exceptions here as well.
- What about accessors for tensors?

More Accessors

```
class tensor {
public:
    ...
    double at(size_t i) const;
    double at(size_t i, size_t j) const;
    ...
}; // class tensor

double tensor::at(size_t i) const {
    assert(get_dim() == 1);
    assert(i < shape_[0]);
    return data_[i];
}
double tensor::at(size_t i, size_t j) const {
    assert(get_dim() == 2);
    assert((i < shape_[0]) && (j < shape_[1]));
    return data_[i*shape_[1]+j];
}
```

- We may create accessors for specific dimensions.
  - Many tensor operations we need to implement have specific requirements on tensor dimensions.
- Use assertions or exceptions to guard against misuse.

Passing tensor Back to C Code

```
class tensor {
public:
    ...
    size_t *get_shape_array();
    double *get_data_array();
    ...
}; // class tensor

size_t *tensor::get_shape_array() {
    return shape_.empty()? nullptr: &shape_[0];
}
double *tensor::get_data_array() {
    return &data_[0];
}
```

- `std::vector` provides backward compatibility with C arrays.
    - However, this feature should be used with care.

Managing tensor Lifetime

```
class evaluation {
    ...
    tensor &get_result();
private:
    ...
    std::map<int, tensor> variables_;
}; // class evaluation
```

- Intermediate variables as `tensor`s are stored within `evaluation`.
- They will be there as long as the `evaluation` object is not destroyed and `variables_` is not cleared.
- `get_result` will return one of them corresponding to the result of the evaluation.

Updated execute() Function

```
int execute(evaluation *eval,
    int *p_dim, size_t **p_shape, double **p_data)
{
    ... // logging and error checking
    tensor &res = eval->get_result();
    *p_dim = res.get_dim();
    *p_shape = res.get_shape_array();
    *p_data = res.get_data_array();
    return 0;
}
```

- Please modify the code once you are done with Project 2.
- Our Python code will be able to construct a NumPy `ndarray` as result after `execute` returns.

Summary and Advice

- Use assertions and exceptions to enforce error handling.
- Tensors are multidimensional arrays.
- Tensors can be conveniently stored and passed around in contiguous row-major order.
- Always start your class design with a well designed class invariant.