C++ neural networks from scratch – Pt 1. building a matrix library (https://lyndonduong.com/linalg-cpp/)

12 minute read

Creating a bare-bones linear algebra library to train a neural net.

Open on GitHub (https://github.com/lyndond/lyndond.github.io/blob/master/code/2021-12-22_neural_net_cpp/)

- Part 1 – building a matrix library
- Part 2 – building an MLP
- Part 3 – model training

Without having access to any linear algebra or autodifferentiation libraries, what's the absolute **bare-minimum** required to train a neural network? Specifically, what do we need to build and train a multilayer perceptron (a cascade of linear-nonlinear layers) to regress onto some data?

In this first part of a 3-part series of posts, I'm going to build a bare-bones linear algebra library from the ground up using modern C++. In the following parts, I'll create a trainable simple neural network, and validate it on some toy data.

Building our own matrix library

Consider the forward pass of a single layer neural network, $y \leftarrow f(Wx+b)$. From this we see that we'll need to build tools that can:

- store data in a matrix (or vector)
- perform matrix products
- perform matrix addition
- apply a function element-wise to a matrix

Other operations that will be helpful for training a simple neural net are:

- element-wise multiplication
  - scalar multiplication as a special case
- matrix subtraction
- matrix transposition

Constructor and printing methods

We'll create a templated class to allow us to create matrices with arbitrary type (float, double, int, etc.). To instantiate a 2x3 float `Matrix`, we just need to call `Matrix<float> M(2, 3);`. The constructor will also instatiate an empty 1D `(2 * 3)` `std::vector` with which to store the data. The shape and number of elements of the matrix are stored in a public tuple. For simplicity, the minimum size of each dimension should be 1, like MATLAB, and unlike numpy.

```
// matrix.h
#pragma once
#include <vector>
#include <cmath>
#include <cassert>
#include <iostream>
#include <tuple>
#include <functional>
#include <random>

template<typename Type>
class Matrix {

  size_t cols;
  size_t rows;

 public:
  std::vector<Type> data;
  std::tuple<size_t, size_t> shape;
  int numel = rows * cols;

  /* constructors */
  Matrix(size_t rows, size_t cols)
      : cols(cols), rows(rows), data({}) {

    data.resize(cols * rows, Type());  // init empty vector for data
    shape = {rows, cols};

  }
  Matrix() : cols(0), rows(0), data({}) { shape = {rows, cols}; };

  /* [print methods go here] */

  /* [linear algebra methods go here] */

};
```

Helper print methods

We'll also add some `print()` methods to the class that will help for debugging and displaying the data of the matrix. The `print()` method simply loops through each element of a row and prints each row to the screen using `std::cout`.

```
// shape printer for debugging, PyTorch style formatting
void print_shape() {
  std::cout << "Matrix Size([" << rows << ", " << cols << "])" << std::endl;
}

void print() {
  for (size_t r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c) {
      std::cout << (*this)(r, c) << " ";
    }
    std::cout << std::endl;
  }
  std::cout << std::endl;
}
```

Accessing matrix elements

Storing the underlying data in a fixed-size vector is more efficient than storing it explicitly as a 2D array because the data is guaranteed to be contiguous in memory. We can just do some simple indexing to translate from (row, col) to linear vector index.

To access elements of the matrix MATLAB-style (except zero-indexed), we can define an overload for the `operator()` that returns a reference to the element to allow us to edit the matrix. Despite the data being stored in a 1D `std::vector` for efficiency, we can easily find the `(row,col)` element using some simple indexing.

```
Type& operator()(size_t row, size_t col) {
  return data[row * cols + col];
}
```

Now we can access a matrix like `M(row, col)` and get the corresponding element of the array.

Basic linear algebra methods

Matrix multiplication

The most important method we'll need is a matrix multiply. If we're given a target matrix with which to multiply our current instantiated matrix, we need to first assert (from `#include <cassert>`) that the `cols` dimension of our current matrix matches the `rows` of our target

matrix.

We're going to implement the most naive/simple matrix multiplication algorithm without worrying about efficiency. If the matrices were both dense and `(n x n)` square, then this would be O(n^3) time complexity. Optimal implentations lie somewhere between O(n^2.3) and O(n^3) (the best we could possibly hope for is O(n^2) since we'd have to loop through all the elements at least once).

```
//O(rows^2 * cols) time | O(rows*cols) space
Matrix matmul(Matrix &target) {
  assert(cols == target.rows);
  Matrix output(rows, target.cols);

  for (size_t r = 0; r < output.rows; ++r) {
    for (size_t c = 0; c < output.cols; ++c) {
      for (size_t k = 0; k < target.rows; ++k)
        output(r, c) += (*this)(r, k) * target(k, c);
    }
  }
  return output;
}
```

A much more straightforward multiplication operation is element-wise multiplication. Given some `target`, we'll assert that its size is the same as our current matrix Then we'll loop through and multiply each element together, and assign to the appropriate element of an `output` matrix.

Using this same logic, we can easily create a `square()` method that does an element-wise squaring of a matrix (useful for computing squared error), and a matrix-times-scalar operation.

```
// O(rows*cols) time | O(rows*cols) space
Matrix multiply_elementwise(Matrix &target){
  assert(shape == target.shape);
  Matrix output((*this));
  for (size_t r = 0; r < output.rows; ++r) {
    for (size_t c = 0; c < output.cols; ++c) {
      output(r, c) = target(r,c) * (*this)(r, c);
    }
  }
  return output;
}

// O(rows*cols) time | O(rows*cols) space
Matrix square() {
  Matrix output((*this));
  output = multiply_elementwise(output);
  return output;
}

// O(rows*cols) time | O(rows*cols) space
Matrix multiply_scalar(Type scalar) {
  Matrix output((*this));
  for (size_t r = 0; r < output.rows; ++r) {
    for (size_t c = 0; c < output.cols; ++c) {
      output(r, c) = scalar * (*this)(r, c);
    }
  }
  return output;
}
```

Matrix addition

Matrix addition is probably the simplest matrix operation apart from scalar multiplication. Given a reference to a `target Matrix`, we'll first assert that the shapes are the same as our current matrix. Then, we'll just loop through each component of both arrays, sum them together, and assign them to the corresponding position of our `output` matrix.

```
// binary arg matrix addition
// O(rows*cols) time | O(rows*cols) space
Matrix add(Matrix &target) {
  assert(shape == target.shape);
  Matrix output(rows, get<1>(target.shape));

  for (size_t r = 0; r < output.rows; ++r) {
    for (size_t c = 0; c < output.cols; ++c) {
      output(r, c) = (*this)(r, c) + target(r, c);
    }
  }
  return output;
}
Matrix operator+(Matrix &target) {
  return add(target);
}
```

Matrix subtraction

To do a matrix operation like $A = B - C$, we can break it down and view it like $A \leftarrow (B + (-C))$. So we can first negate `C`, then use our existing method to add `-C` to `B`.

To negate a matrix, we can write a unary `operator-()` that loops through the current matrix's data, negates every element and assigns them each to the corresponding elements of an `output` matrix.

```
// unary-arg matrix negation
// O(rows*cols) time | O(rows*cols) space
Matrix operator-() {
  Matrix output(rows, cols);
  for (size_t r = 0; r < rows; ++r) {
    for (size_t c = 0; c < cols; ++c) {
      output(r, c) = -(*this)(r, c);
    }
  }
  return output;
}
```

Now that we have a unary negation operator, to subtract a `target` matrix from our current matrix, we simply negate `target`, then call our `add()` method on the negated `target`.

```
// binary-arg matrix subtraction
// O(rows*cols) time | O(rows*cols) space
Matrix sub(Matrix &target) {
  Matrix neg_target = -target;
  return add(neg_target);
}
Matrix operator-(Matrix &target) {  // for cleaner usage
  return sub(target);
}
```

This can be optimized by rewriting the `add()` method entirely while replacing the `+` with a `-`, but for our purposes, doing it this way is a nice exercise in composing different methods together.

Matrix transpose

To transpose a matrix, we simply swap the elements of its rows and columns. We can do this easily by instantiating a new `Matrix` with the transposed size of our current matrix, assign the appropriate elements to it by looping through our current matrix, then returning the new matrix.

```
// swap rows and cols
// O(rows*cols) time | O(rows*cols) space
Matrix transpose() {
  size_t new_rows{cols}, new_cols{rows};
  Matrix transposed(new_rows, new_cols);

  for (size_t r = 0; r < new_rows; ++r) {
    for (size_t c = 0; c < new_cols; ++c) {
      transposed(r, c) = (*this)(c, r);  // swap row and col
    }
  }
  return transposed;
}
Matrix T(){ // Similar to numpy, etc.
  return transpose();
}
```

Element-wise function application

Neural networks would be pretty useless without nonlinearities, because a cascade of linear operations can be collapsed into a single linear operation. So we'll need to create a way to apply a nonlinear function (i.e. activation function) to each element of a matrix.

To apply a function (e.g. sigmoid, Tanh, ReLU, etc.) to each element of the matrix, we'll need a method that can take as input a reference to a function (from `#include <functional>`). We'll then loop through every component of the array and assign the output of the function of that component to a new `output` matrix.

```
// O(rows*cols* O(function)) time | space depends on function
Matrix apply_function(const std::function<Type(const Type &)> &function) {
  Matrix output((*this));
  for (size_t r = 0; r < rows; ++r) {
    for (size_t c = 0; c < cols; ++c) {
      output(r, c) = function((*this)(r, c));
    }
  }
  return output;
}
```

Matrix initialization

Lastly, we'll need to create matrices filled with structured data in order to build and train a neural net. Different options include something analogous to `zeros()`, or `ones()`, or various random matrices like `random.randn()` in numpy. Here's an implementation of a `randn()` matrix function that we'll use to initialize our weight and bias matrices to Normal-distributed values.

```
template <typename T>
struct mtx {
  static Matrix<T> randn(size_t rows, size_t cols) {
    Matrix<T> M(rows, cols);

    std::random_device rd{};
    std::mt19937 gen{rd()};

    // init Gaussian distr. w/ N(mean=0, stdev=1/sqrt(numel))
    T n(M.numel);
    T stdev{1 / sqrt(n)};
    std::normal_distribution<T> d{0, stdev};

    // fill each element w/ draw from distribution
    for (size_t r = 0; r < rows; ++r) {
      for (int c = 0; c < cols; ++c) {
        M(r, c) = d(gen);
      }
    }
    return M;
  }
```

The fruits of our labour

Now let's test all the functions we made:

```
#include "matrix.h"
int main(){
  auto M = mtx<float>::randn(2, 2); // init randn matrix

  M.print_shape();
  M.print(); // print the OG matrix

  (M-M).print();  // print M minus itself

  (M+M).print();  // print its sum
  (M.multiply_scalar(2.f)).print();  // print 2x itself

  (M.multiply_elementwise(M)).print(); // mult M w itself

  auto MT = M.T(); // transpose the matrix
  MT.print();
  (MT.matmul(M)).print();  // form symm. pos. def. matrix

  (M.apply_function([](auto x){return x-x;} )).print(); // apply fun
}
```

Outputs:

```
// M.shape
Matrix Size([2, 2])

// M
0.0383611 -0.310235
-0.615681 -0.356848

// M - M
0 0
0 0

// M + M
0.0767223 -0.62047
-1.23136 -0.713695

// 2 * M
0.0767223 -0.62047
-1.23136 -0.713695

// M elementwise multiply M
0.00147158 0.0962457
0.379063 0.12734

// M transpose
0.0383611 -0.615681
-0.310235 -0.356848

// M.T() matrix-multiply M
0.380534 0.207803
0.207803 0.223586

// Apply f(x) = x-x to each element of M.
0 0
0 0
```

With this from-scratch linear algebra library, we're ready to now build a neural network.

tags: c++ linear-algebra ml

updated: December 22, 2021