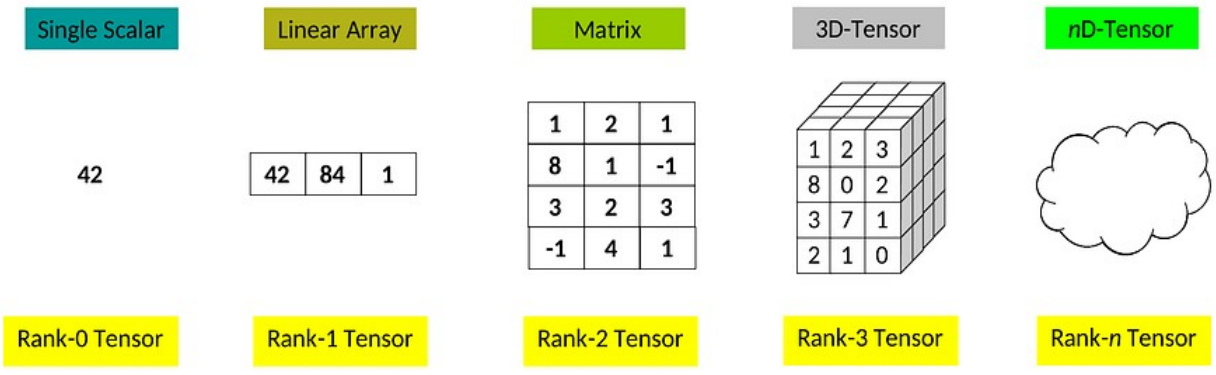
C++: Tensor Programming

Tensors are the primary way to represent data in Deep Learning algorithms. They are widely used to implement inputs, outputs, parameters, and internal states during the algorithm execution.

- What are tensors
- How to define tensors in C++
- How to compute tensor operations
- Tensor Reductions and Convolutions

What is a Tensor?

Tensors are grid-like data structures that generalize the concepts of vectors and matrices for an arbitrary number of axis. In machine learning, usually instead of “axis” the word “**dimension**” is used . The number of different dimensions of a tensor is also called tensor **rank**:



The simplest operations we can perform with tensors are the so-called element-wise operations: given two operand tensors with the same dimensions, the operation results in a new tensor with the same dimensions where the value of each coefficient is obtained from the binary evaluation on the respective elements in the operands:

1	2	1
8	1	-1
3	2	3
-1	4	1

X

2	1	2
3	1	1
-2	1	4
2	2	3

=

1 x 1	2 x 1	1 x 2
8 x 3	1 x 1	-1 x 1
3 x -2	2 x 1	3 x 4
-1 x 2	4 x 2	1 x 3

Like matrices, we can perform other more sophisticated operations with tensors, such as the matrix-like product, convolutions, reductions, and a myriad of geometric operations.

How to declare and use tensors in C++

Eigen is a linear algebra library broadly used for matrix computations. In addition to the well know support for matrices, Eigen also has an (unsupported) module for Tensors.

> *Although the Eigen Tensor API is denoted as unsupported, it is actually well supported by the developers of the Google TensorFlow framework.*

We can easily define a tensor using Eigen:

```cpp
#include <iostream>
#include <unsupported/Eigen/CXX11/Tensor>
int main() {
    Eigen::Tensor<int, 3> my_tensor(2, 3, 4);
    my_tensor.setConstant(42);
    std::cout << "my_tensor:\n\n" << my_tensor << "\n\n";
    std::cout << "tensor size is " << my_tensor.size() << "\n\n";
    return 0;
}
```

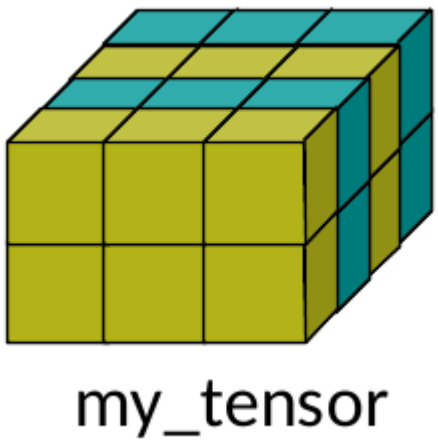
```
my_tensor:

42 42 42 42
42 42 42 42
42 42 42 42

42 42 42 42
42 42 42 42
42 42 42 42

tensor size is 24
```

We can represent `my_tensor` as follows:



We can set the tensor data if we want it:

```
my_tensor.setValues({{{1, 2, 3, 4}, {5, 6, 7, 8}}});
std::cout << "my_tensor:\n\n" << my_tensor << "\n\n";
```

```
my_tensor:

 1  2  3  4
 5  6  7  8
42 42 42 42

42 42 42 42
42 42 42 42
42 42 42 42
```

Creating tensor with Eigen::TensorMap

Sometimes, we have some data allocated and only want to manipulate it using a tensor. `Eigen::TensorMap` is similar to `Eigen::Tensor` but, instead of allocating new data, it is only a view of the data passed as the parameter.

```
// a vector with size 12
std::vector<float> storage(4*3);
// filling vector from 1 to 12
std::iota(storage.begin(), storage.end(), 1.);
for (float v: storage) std::cout << v << ',';
std::cout << "\n\n";
// setting a tensor view with 4 rows and 3 columns
Eigen::TensorMap<Eigen::Tensor<float, 2>> my_tensor_view(storage.data(), 4, 3);
std::cout << "my_tensor_view before update:\n\n" << my_tensor_view << "\n\n";
// updating the vector
storage[4] = -1.;
std::cout << "my_tensor_view after update:\n\n" << my_tensor_view << "\n\n";
// updating the tensor
```

```
my_tensor_view(2, 1) = -8;
std::cout << "vector after two updates:\n\n";
for (float v: storage) std::cout << v << ',';
std::cout << "\n\n";
```

```
1,2,3,4,5,6,7,8,9,10,11,12,

my_tensor_view before update:

 1  5  9
 2  6 10
 3  7 11
 4  8 12

my_tensor_view after update:

 1 -1  9
 2  6 10
 3  7 11
 4  8 12

vector after two updates:

1,2,3,4,-1,6,-8,8,9,10,11,12,
```

Performing unary and binary operations

The Eigen Tensor API defines common arithmetic overload operators, making programming tensors intuitive and straightforward. For example, we can add and subtract tensors:

```
Eigen::Tensor<float, 2> A(2, 3), B(2, 3);
A.setRandom();
B.setRandom();
Eigen::Tensor<float, 2> C = 2.f*A + B.exp();
std::cout << "A is\n\n"<< A << "\n\n";
std::cout << "B is\n\n"<< B << "\n\n";
std::cout << "C is\n\n"<< C << "\n\n";
```

```
A is

  0.171153   0.598665   0.894185
  0.700085  0.0262164 0.00292921

B is

 0.138388 0.0556408  0.793353
 0.551635  0.942522   0.32321

C is

1.49073 2.25455 3.99917
3.13626 2.61888 1.38741
```

The Eigen Tensor API has several other element-wise functions like `.exp()` such as `sqrt()`, `log()`, and `abs()`. In addition, we can use `unaryExpr()` as follows:

```
auto cosine = [](float v) {return cos(v);};
Eigen::Tensor<float, 2> D = A.unaryExpr(cosine);
std::cout << "D is\n\n"<< D << "\n\n";
```

Similarly, we can use `binaryExpr` :

```
auto fun = [](float a, float b) {return 2.*a + b;};
Eigen::Tensor<float, 2> E = A.binaryExpr(B, fun);
std::cout << "E is\n\n"<< E << "\n\n";
```

```
D is

0.985389 0.826089 0.626154
0.764787 0.999656 0.999996

E is

0.480694  1.25297  2.58172
  1.9518 0.994955 0.329068
```

Lazy evaluation and the auto keyword

The Google engineers who worked on the Eigen Tensor API followed the same strategies found at the top of the Eigen Library. One of these strategies, and probably the most important one, is the way how expressions are lazily evaluated.

The lazy evaluation strategy : instead of evaluating multiple individual expressions step-by-step, the optimized code evaluates only one expression, aiming to leverage the final overall performance.

For example, if `A` and `B` are tensors, the expression `A + B` does not actually evaluates the sum of A and B. In fact, the expression`A + B` results in a special object that **knows** how to compute `A + B` . The operation will only be performed when this special object is assigned to an actual tensor.

Geometric Operations

Geometric operations result in tensors with different dimensions and, sometimes, sizes. Examples of these operations are: `reshape` , `pad` , `shuffle` , `stride` , and `broadcast` .

Reductions

Reductions are a special case of operations that results in a tensor with fewer dimensions than the original. Intuitive cases of reductions are `sum()` and `maximum()` :

```
Eigen::Tensor<float, 3> X(5, 2, 3);
X.setRandom();
std::cout << "X is\n\n"<< X << "\n\n";
```

```
std::cout << "X.sum(): " << X.sum() << "\n\n";
std::cout << "X.maximum(): " << X.maximum() << "\n\n";
```

In the example above, we reduced all the dimensions once. We can also perform reductions along specific axes. For example:

```
Eigen::array<int, 2> dims({1, 2});
std::cout << "X.sum(dims): " << X.sum(dims) << "\n\n";
std::cout << "X.maximum(dims): " << X.maximum(dims) << "\n\n";
```

```
X is

  0.652547  0.0620345   0.974936
  0.142522   0.256882  0.0953116

  0.291391    0.52403   0.630934
  0.938475    0.94273   0.344463

 0.0616169   0.648677 0.00778913
  0.604869   0.876383   0.922909

  0.877274  0.0568734   0.604315
  0.181578   0.908388   0.677456

  0.355512   0.757973    0.13508
  0.925048   0.863703   0.210591

X.sum(): 15.5323

X.maximum(): 0.974936

X.sum(dims): 2.18423 3.67202 3.12224 3.30588 3.24791

X.maximum(dims): 0.974936  0.94273 0.922909 0.908388 0.925048
```

Tensor Convolutions

The EigenTensor API has a handy function to perform convolutions on Eigen Tensor objects:

```
Eigen::Tensor<float, 4> input(1, 6, 6, 3);
input.setRandom();
Eigen::Tensor<float, 2> kernel(3, 3);
kernel.setRandom();
Eigen::Tensor<float, 4> output(1, 4, 4, 3);
Eigen::array<int, 2> dims({1, 2});
output = input.convolve(kernel, dims);
std::cout << "input:\n\n" << input << "\n\n";
std::cout << "kernel:\n\n" << kernel << "\n\n";
std::cout << "output:\n\n" << output << "\n\n";
```

```
input:

 0.962597  0.743963  0.900069
 0.479409   0.62146  0.885605
 0.823462  0.146216    0.5326
 0.444228  0.624226  0.821309
 0.399761   0.66927  0.796553
 0.105454  0.660206  0.428697

 0.258843  0.527224 0.0800565
0.0581139   0.46684  0.465314
 0.234161  0.269673   0.97252
 0.403703  0.364218  0.449049
 0.225875  0.898043  0.990027
 0.675387  0.792336  0.760356

 0.809585  0.801422  0.822022
 0.507742  0.893558  0.440074
 0.255932  0.765428  0.320114
 0.963614  0.831387  0.405793
 0.684739  0.114154  0.252944
 0.268413  0.122313  0.542188

 0.447567  0.919464  0.563842
 0.386719  0.151272  0.577875
 0.500138  0.368536  0.459911
 0.278458 0.0804685  0.626319
 0.241699  0.337662 0.0512156
 0.261568  0.786341  0.785995

 0.531592  0.771128  0.676824
 0.418316   0.60076  0.970422
 0.302456  0.307922  0.781982
 0.471977  0.128051  0.802593
 0.638098  0.868197 0.0396085
 0.277032   0.25938  0.778151

 0.937312  0.258756  0.968418
 0.382177  0.488793  0.715074
 0.284284  0.477707  0.447849
 0.725899  0.433522  0.540112
 0.703743  0.411378  0.329663
 0.970909  0.952258  0.693393
```

```
kernel:

0.898731 0.450139 0.978761
0.894915 0.305695 0.206081
0.948376 0.255745 0.227028

output:

 3.1403 2.92393 3.10797
2.20886 3.04252 3.23713
2.35543 2.56965 3.30442
2.40184 3.11765 2.95011

2.05735 3.09037 2.95634
1.84185  2.3584 2.67864
1.87837  2.6947 3.11579
2.63781  2.6524 2.93116

2.50827 3.77156  3.0938
2.65226 2.90353 3.02612
 2.4689 2.11976 2.28739
2.48765 1.65794 2.69532

2.78298 2.67121 3.48349
2.00926 1.72291  3.5276
1.97288 1.84887 2.33642
2.37843 2.15994 2.99993
```

Limitations

The Eigen Tensor API docs cite some limitations of which we can be aware:

- The GPU support was tested and optimized to the float type. Even if we can declare `Eigen::Tensor<int,...> tensor;`, the usage of non-float tensors is discouraged when using the GPU.
- The default layout (col-major) is the only one actually supported. We shouldn't use row-major, at least for now.
- The max number of dimensions is 250. This size is only achieved when using a C++11-compatible compiler.

Conclusion

Tensors are essential data structures for machine learning programming, allowing us to represent and handle multi-dimensional data as straightforwardly as regular two-dimensional matrices.

The Eigen Tensor API was introduced and it has been learned how to use tensors relatively easily. It was also considered that the Eigen Tensor API has a lazy evaluation mechanism, resulting in an optimized execution in terms of memory and processing time.

Source:

https://ai.plainenglish.io/deep-learning-from-scratch-in-c-tensor-programming-83bca6930e96