

---

# **primitiv Documentation**

***Release 0.4.0***

**primitiv Developer Group**

**Feb 24, 2018**



---

## Contents:

---

<b>1</b>	<b>Installing primitiv</b>	<b>3</b>
1.1	Prerequisites . . . . .	3
1.2	Installing primitiv from source (Debian/Ubuntu) . . . . .	4
1.3	Installing primitiv with CUDA . . . . .	5
<b>2</b>	<b>primitiv C++ Tutorials</b>	<b>7</b>
2.1	Step-by-step Example: Solving the XOR Problem . . . . .	7
<b>3</b>	<b>Library Designs</b>	<b>15</b>
3.1	Headers, Library Files and Compilation . . . . .	15
3.2	Shapes and Operation Rules . . . . .	16
3.3	Nodes and Tensors . . . . .	18
3.4	Devices . . . . .	21
3.5	Default Device and Graph . . . . .	21
3.6	Training . . . . .	21
3.7	Models . . . . .	21
3.8	Saving and Loading . . . . .	21
<b>4</b>	<b>primitiv Reference</b>	<b>23</b>
4.1	primitiv API Reference . . . . .	23
4.2	Build Options . . . . .	76
4.3	primitiv File Format v0.1 . . . . .	77



primitiv is a neural network library developed by [National Institute of Information and Communications Technology \(NICT\)](#) and [Nara Institute of Science and Technology \(NAIST\)](#). primitiv is written in C++11 and supports some other languages such as Python and Rust through bindings. primitiv allows users to write their own networks using a *define-by-run* style construction methods, and most features in the library is designed device-independent. Users can perform the own network using various computing backends such as Eigen, CUDA and OpenCL with no (or few) modifications of the original code.



This section describes how to install primitiv to your computer.

### 1.1 Prerequisites

primitiv is designed based on a device-independent policy, and you can choose dependencies between primitiv and other hardwares using *build options*.

For the minimal configuration (no other hardwares), primitiv requires below softwares/libraries:

- C++11 compiler (GCC, Clang, others)
- CMake 3.1.0 or later

For building unit tests, it requires below libraries:

- Google Test

For using specific hardwares, it requires some hardware-dependent libraries:

- `primitiv::devices::Eigen`
  - Eigen 3.3.0 or later
- `primitiv::devices::CUDA`
  - CUDA Toolkit 8.0 or later
  - cuDNN 5.1.0 or later
- `primitiv::devices::OpenCL`
  - OpenCL 1.2
  - OpenCL C++ Bindings 2 (cl2.hpp)
  - clBLAS

## 1.2 Installing primitiv from source (Debian/Ubuntu)

### 1.2.1 Installing common prerequisites

```
$ apt install build-essential cmake
```

### 1.2.2 Installing Eigen

Although you can install `primitiv` without any specific hardware, we recommend to bind at least the **Eigen** backend to compute your neural networks much faster on CPUs.

```
$ apt install wget
$ cd /path/to/your/src
$ mkdir -p eigen
$ wget http://bitbucket.org/eigen/eigen/get/3.3.4.tar.bz2
$ tar -xf 3.3.4.tar.bz2 -C eigen --strip-components=1
$ rm 3.3.4.tar.bz2
```

### 1.2.3 Installing primitiv

To select `primitiv` versions to be installed, you can retrieve some archives from [official releases](#).

```
$ cd /path/to/your/src
$ mkdir -p primitiv
$ wget https://github.com/primitiv/primitiv/archive/v0.3.1.tar.gz
$ tar -xf v0.3.1.tar.gz -C primitiv --strip-components=1
$ rm v0.3.1.tar.gz
```

Also, you can download a development (or other specific) branch using Git:

```
$ cd /path/to/your/src
$ apt install git
$ git clone https://github.com/primitiv/primitiv -b develop
```

Then we build `primitiv` using a standard process of CMake:

```
$ cd /path/to/your/src/primitiv
$ mkdir build
$ cd build
$ cmake ..
$ make
$ make install
```

`make install` will create `libprimitiv.so` in the system library directory and `primitiv` directory in the system include directory.

In some cases, you also need to add the path to the library directory to the `${LD_LIBRARY_PATH}` environment variable:

```
$ export LD_LIBRARY_PATH=/path/to/your/lib:${LD_LIBRARY_PATH}
```

If we use the Eigen backend, specify both `EIGEN3_INCLUDE_DIR` and `PRIMITIV_USE_EIGEN` options to `cmake`:



```
$ cmake .. \  
-DEIGEN3_INCLUDE_DIR=/path/to/your/src/eigen \  
-DPRIMITIV_USE_EIGEN=ON
```

## 1.3 Installing primitiv with CUDA

```
$ cmake .. -DPRIMITIV_USE_CUDA=ON
```

The build process tries to find the CUDA Toolkit and the cuDNN library by default. You can also specify the explicit locations of their libraries if searching failed or you want to switch them:

```
$ cmake .. \  
-DCUDA_TOOLKIT_ROOT_DIR=/path/to/cuda \  
-DCUDNN_ROOT_DIR=/path/to/cuda \  
-DPRIMITIV_USE_CUDA=ON
```



This section describes tutorials to learn how to use primitiv in your C++ code..

## 2.1 Step-by-step Example: Solving the XOR Problem

This tutorial introduces a basic and common usage of the primitiv by making and training a simple network for a small classification problem.

### 2.1.1 Introduction: Problem Formulation

Following lines are the formulation of the problem used in this tutorial:

$$\begin{aligned} f : \quad \mathbb{R}^2 &\rightarrow [-1, 1]; \\ f : (x_1, x_2) &\mapsto \begin{cases} 1, & \text{if } x_1 x_2 \geq 0, \\ -1, & \text{otherwise,} \end{cases} \end{aligned}$$

where  $x_1, x_2 \in \mathbb{R}$ . This is known as the *XOR problem*;  $f$  detects whether the signs of two arguments are same or not. We know that this problem is *linearly non-separable*, i.e., the decision boundary of  $f$  can NOT be represented as a straight line on  $\mathbb{R}$ :  $\alpha x_1 + \beta x_2 + \gamma = 0$ , where  $\alpha, \beta, \gamma \in \mathbb{R}$ .

For example, following code generates random data points  $(x_1 + \epsilon_1, x_2 + \epsilon_2, f(x_1, x_2))$  according to this formulation with  $x_1, x_2 \sim \mathcal{N}(x; 0, \sigma_{\text{data}})$  and  $\epsilon_1, \epsilon_2 \sim \mathcal{N}(\epsilon; 0, \sigma_{\text{noise}})$ :

```
#include <random>
#include <tuple>

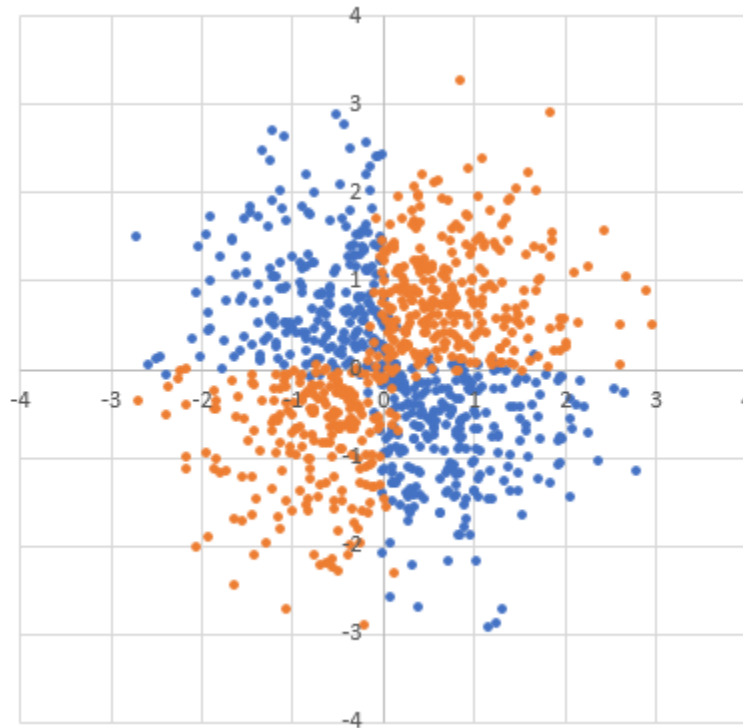
class DataSource {
    std::mt19937 rng;
    std::normal_distribution<float> data_dist, noise_dist;

public:
    // Initializes the data provider with two SDs.
```

```
DataSource(float data_sd, float noise_sd)
: rng(std::random_device()())
, data_dist(0, data_sd)
, noise_dist(0, noise_sd) {}

// Generates a data point
std::tuple<float, float, float> operator()() {
    const float x1 = data_dist(rng);
    const float x2 = data_dist(rng);
    return std::make_tuple(
        x1 + noise_dist(rng),    // x1 + err
        x2 + noise_dist(rng),    // x2 + err
        x1 * x2 >= 0 ? 1 : -1); // label
}
};
```

Following graph is an actual sample generated by above class with data\_sd is 1 and noise\_sd is 0.1:



In this tutorial, we construct a 2-layers (input-hidden-output) perceptron to solve this problem. The whole model formulation is:

$$y := \tanh(W_{hy}\mathbf{h} + b_y),$$

$$\mathbf{h} := \tanh(W_{xh}\mathbf{x} + \mathbf{b}_h),$$

where  $y \in \mathbb{R}$  is an output value to be fit to  $f(x_1, x_2)$ ,  $\mathbf{x} := (x_1 \ x_2)^\top \in \mathbb{R}^2$  is an input vector,  $\mathbf{h} \in \mathbb{R}^N$  represents the  $N$ -dimensional *hidden state* of the network. There are also 4 free parameters: 2 matrices  $W_{hy} \in \mathbb{R}^{1 \times N}$  and  $W_{xh} \in \mathbb{R}^{N \times 2}$ , and 2 bias (column) vectors  $b_y \in \mathbb{R}$  and  $\mathbf{b}_h \in \mathbb{R}^N$ .

## 2.1.2 Include and Initialization

primitiv requires you to include `primitiv/primitiv.h` before using any features in the source code. All features in primitiv is enabled by including this header (available features are depending on specified *options while building*).

`primitiv/primitiv.h` basically may not affect the global namespace, and all features in the library is declared in the `primitiv` namespace. But for brevity, we will omit the `primitiv` namespace in this tutorial using the `using namespace` directives. Please pay attention to this point when you reuse these snippets.

```
#include <iostream>
#include <vector>
#include <primitiv/primitiv.h>

using namespace std;
using namespace primitiv;

int main() {

    // All code will be described here.

    return 0;
}
```

Before making our network, we need to create at least two objects: Device and Graph. Device objects specifies an actual computing backends (e.g., usual CPUs, CUDA, etc.) and memory usages for these backends. If you installed primitiv with no build options, you can initialize only `primitiv::devices::Naive` device object. Graph objects describe a temporary computation graph constructed by your code and provides methods to manage their graphs.

```
devices::Naive dev;
Graph g;

// "Eigen" device can be enabled when -DPRIMITIV_USE_EIGEN=ON
//devices::Eigen dev;

// "CUDA" device can be enabled when -DPRIMITIV_USE_CUDA=ON
//devices::CUDA dev(gpu_id);
```

Note that Device and Graph is not a singleton; you can also create any number of Device/Graph objects if necessary (even multiple devices share the same backend).

After initializing a Device and a Graph, we set them as the **default device/graph** used in the library.

```
Device::set_default(dev);
Graph::set_default(g);
```

For now, it is enough to know that these are just techniques to reduce coding efforts, and we don't touch the details of this function. For more details, please read the *document about default objects*.

## 2.1.3 Specifying Parameters and an Optimizer

Our network has 4 parameters described above:  $W_{xh}$ ,  $b_h$ ,  $W_{hy}$  and  $b_y$ . We first specify these parameters as Parameter objects:

```
constexpr unsigned N = 8;
Parameter pw_xh({N, 2}, initializers::XavierUniform());
Parameter pb_h({N}, initializers::Constant(0));
```

```
Parameter pw_hy({1, N}, initializers::XavierUniform());
Parameter pb_y({}, initializers::Constant(0));
```

Parameter objects basically take two arguments: *shape* and *initializer*. Shapes specify actual volume (and number of free variables) in the parameter, and initializer gives initial values of their variables. Above code uses the [Xavier \(Glorot\) Initializer](#) for matrices, and the constant 0 for biases.

Next we initialize an `Optimizer` object and register all parameters to train their values. We use simple SGD optimizer for now:

```
constexpr float learning_rate = 0.1;
optimizers::SGD opt(learning_rate);
opt.add(pw_xh, pb_h, pw_hy, pb_y);
```

## 2.1.4 Writing the Network

primitiv adopts the **define-by-run** style for writing neural networks. Users can write their own networks as usual C++ functions. Following code specifies the network described the above formulation using a lambda functor which takes and returns `Node` objects:

```
// 2-layers feedforward neural network
// `x` should be with `Shape({2}, B)`
auto feedforward = [&](const Node &x) {
    namespace F = primitiv::functions;
    const Node w_xh = F::parameter<Node>(pw_xh); // Shape({N, 2})
    const Node b_h = F::parameter<Node>(pb_h); // Shape({N})
    const Node w_hy = F::parameter<Node>(pw_hy); // Shape({1, N})
    const Node b_y = F::parameter<Node>(pb_y); // Shape({})
    const Node h = F::tanh(F::matmul(w_xh, x) + b_h); // Shape({N}, B)
    return F::tanh(F::matmul(w_hy, h) + b_y); // Shape({}, B)
};
```

`Node` objects represent an virtual results of network calculations which are returned by functions declared in the `primitiv::functions` namespace and can be used as an argument of their functions. Each `Node` has a *shape*, which represents the volume and the size of the minibatch of the `Node`. `primitiv` encapsulates the treatment of minibatches according to the [minibatch broadcasting rule](#), and users can concentrate on writing the network structure without considering actual minibatch sizes.

We also describe a loss function about our network:

```
// Network for the squared loss function.
// `y` is that of returned from `feedforward()`
// `t` should be with `Shape({}, B)`
auto squared_loss = [&](const Node &y, const Node &t) {
    namespace F = primitiv::functions;
    const Node diff = y - t; // Shape({}, B)
    return F::batch::mean(diff * diff); // Shape({})
};
```

Also, we write the network to generate input data from above `DataSource` class:

```
constexpr float data_sd = 1.0;
constexpr float noise_sd = 0.1;
DataSource data_source(data_sd, noise_sd);

auto next_data = [&](unsigned minibatch_size) {
```

```

std::vector<float> data;
std::vector<float> labels;
for (unsigned i = 0; i < minibatch_size; ++i) {
    float x1, x2, t;
    std::tie(x1, x2, t) = data_source();
    data.emplace_back(x1);
    data.emplace_back(x2);
    labels.emplace_back(t);
}

namespace F = primitiv::functions;
return std::make_tuple(
    F::input<Node>(Shape({2}, minibatch_size), data),    // input data `x`
    F::input<Node>(Shape({}, minibatch_size), labels)); // label data `t`
};

```

`primitiv::functions::input` takes *shape* and actual data (as a `vector<float>`) to make a new `Node` object. The order of data should be the **column-major order**, and the minibatch is treated as the *last dimension* w.r.t. the actual data. For example, the `Node` with `Shape({2, 2}, 3)` has 12 values:

$$\begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix}, \begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix}, \begin{pmatrix} a_3 & b_3 \\ c_3 & d_3 \end{pmatrix}$$

and the actual data should be ordered as:

$$a_1, c_1, b_1, d_1, a_2, c_2, b_2, d_2, a_3, c_3, b_3, d_3.$$

## 2.1.5 Writing the Training Loop

Now we can perform actual training loop of our network:

```

for (unsigned epoch = 0; epoch < 100; ++epoch) {
    // Initializes the computation graph
    g.clear();

    // Obtains the next data
    Node x, t;
    std::tie(x, t) = next_data(1000);

    // Calculates the network
    const Node y = feedforward(x);

    // Calculates the loss
    const Node loss = squared_loss(y, t);
    std::cout << epoch << ": train loss=" << loss.to_float() << std::endl;

    // Performs backpropagation and updates parameters
    opt.reset_gradients();
    loss.backward();
    opt.update();
}

```

Above code uses `Node.to_float()`, which returns an actual single value stored in the `Node` (this function can be used only when the `Node` stores just one value).

You may get following results by running whole code described above (results may change randomly every time you launch the program):

```
0: loss=1.17221
1: loss=1.07423
2: loss=1.06282
3: loss=1.04641
4: loss=1.00851
5: loss=1.01904
6: loss=0.991312
7: loss=0.983432
8: loss=0.9697
9: loss=0.97692
...
```

## 2.1.6 Testing

Additionally, we launch a *test* process using a fixed data points in every 10 epochs:

- $(1, 1) \mapsto 1$
- $(-1, 1) \mapsto -1$
- $(-1, -1) \mapsto 1$
- $(1, -1) \mapsto -1$

```
for (unsigned epoch = 0; epoch < 100; ++epoch) {
    //
    // Training process written in the previous code block
    //

    if (epoch % 10 == 9) {
        namespace F = primitiv::functions;
        const Node test_x = F::input<Node>(Shape({2}, 4), {1, 1, -1, 1, -1, -1, 1, -1});
        const Node test_t = F::input<Node>(Shape({}, 4), {1, -1, 1, -1});
        const Node test_y = feedforward(test_x);
        const Node test_loss = squared_loss(test_y, test_t);
        std::cout << "test results:";
        for (float val : test_y.to_vector()) {
            std::cout << ' ' << val;
        }
        std::cout << "\ntest loss: " << test_loss.to_float() << std::endl;
    }
}
```

where `Node.to_vector()` returns all values stored in the `Node`.

Finally, you may get like below:

```
...
8: loss=0.933427
9: loss=0.927205
test results: 0.04619 -0.119208 0.0893511 -0.149148
test loss: 0.809695
10: loss=0.916669
11: loss=0.91744
...
18: loss=0.849496
19: loss=0.845048
test results: 0.156536 -0.229959 0.171106 -0.221599
```



```
test loss: 0.649342
20: loss=0.839679
21: loss=0.831217
...
```

We can see that the test results approaches correct values and the test loss becomes small by proceeding the training process.



This section describes concepts and designs of `primitiv`.

## 3.1 Headers, Library Files and Compilation

### 3.1.1 Compile Options

`primitiv` is written in **C++11**. Users must specify appropriate compiler options to enable the C++11 specification. In most GCC-like compilers, `-std=c++11` option can be used for this purpose.

### 3.1.2 Install Paths

`primitiv` is installed according to the usual process of CMake. In most UNIX-like systems, all files required to use `primitiv` is installed into `/usr/local` by default. Users can change this location using `CMAKE_INSTALL_PREFIX` standard option of CMake.

After installation, the install location should have at least following files:

```
PREFIX/include/primitiv/primitiv.h
      / ... (other files)
      /c/api.h
      / ... (other files)
/lib/libprimitiv.so
```

### 3.1.3 Header Files

All C++ header files of `primitiv` is placed in the `PREFIX/include/primitiv` directory. `primitiv.h` is a useful header to include all features of `primitiv` installed onto your machine. Whether some features (e.g. CUDA device class) can be used or not is represented as the macros defined in `config.h`

PREFIX/include/primitiv/c directory stores C-language API headers used by some bindings between other languages. c/api.h can be used similarly with primitiv.h to include all the features available through C APIs.

If the PREFIX directory is specified as an root of the include paths, you can include these header files like following:

```
#include <primitiv/primitiv.h>
#include <primitiv/c/api.h>
```

### 3.1.4 Library Files

PREFIX/lib directory has libprimitiv.so shared object file. Users should link this file when compiling your own code using primitiv.

If the PREFIX directory is specified as an root of the library paths, you can link libprimitiv.so like following:

```
cc -std=c++11 your_source.cc -lprimitiv
```

## 3.2 Shapes and Operation Rules

### 3.2.1 Shapes

Node, Tensor and Parameter objects have a **Shape** which describes actual appearances of inner data of those objects.

Shape consists of two elements: *dimension* and *minibatch size*. Dimensions are the list of integers which describes volumes of each axis. For example, following code creates new Shape descriptors of a *scalar*, a *column vector*, a *matrix* and a *image* used in CNN functions:

```
using primitiv::Shape;

// Creating Shape of scalars.
const Shape scalar1({});
const Shape scalar2 {};
const Shape scalar3;

// Creating Shape of 3-dimensional column vectors.
const Shape vector1({3});
const Shape vector2 {3};

// Creating Shape of 3x2 matrices.
const Shape matrix1({3, 2}); // {rows, columns}
const Shape matrix2 {3, 2};

// Creating Shape of the image.
const Shape image1({256, 256, 3}); // {width1, width2, channel}

// Shapes with the minibatch size 64.
const Shape scalar_minibatched({}, 64);
const Shape vector_minibatched({3}, 64);
const Shape matrix_minibatched({3, 2}, 64);
const Shape image_minibatched({256, 256, 3}, 64);
```

Two Shapes can be compared using == and != operators:

```
using primitiv::Shape;
using namespace std;

const Shape shape1 {3, 2};
const Shape shape2 {3, 2};
const Shape shape3 {3, 3};
const Shape shape4 {3, 2, 64};

cout << boolalpha;
cout << (shape1 == shape2) << endl; // true
cout << (shape1 == shape3) << endl; // false
cout << (shape1 == shape4) << endl; // false
```

primitiv does not distinguish shapes by the number of dimensions. All Shapes with smaller number of dimensions are completely *compatible* with Shapes with arbitrary bigger number of dimensions with the size of excessive dimensions 1:

```
using primitiv::Shape;
using namespace std;

const Shape scalar1 {};
const Shape scalar2 {1, 1, 1, 1};

const Shape vector1 {3};
const Shape vector2 {3, 1}; // This looks also a 3x1 matrix.

const Shape matrix1 {3, 2};
const Shape matrix2 {3, 2, 1}; // This looks also a 3x2 image with 1 channel.

cout << boolalpha;
cout << (scalar1 == scalar2) << endl; // true
cout << (vector1 == vector2) << endl; // true
cout << (matrix1 == matrix2) << endl; // true
```

### 3.2.2 Minibatch Broadcasting

All functions that take 2 or more Nodes or Tensors applies following rules:

1. If the shapes of two variables **have the same minibatch size**, the function performs independently for each data in the minibatch.
2. If at least one shape of a variable **has no minibatch (= minibatch size 1)**, the function broadcasts values to the minibatch size of the opposite side.
3. Otherwise, the function generates an error.
4. Functions that take more than 2 Nodes or Tensors perform above rules recursively.

Following examples shows how these rules work.

```
using primitiv::Node;
namespace F = primitiv::functions;

const Node a = F::input<Node>(Shape({}, 3), {1, 2, 3});

Node b = F::input<Node>(Shape({}, 3), {4, 5, 6});
Node y = a + b; // values: 5, 7, 9
```

```
b = F::input<Node>({}, {4});
y = a + b; // values: 5, 6, 7
y = b + a; // values: 5, 6, 7

b = F::input<Node>(Shape({}, 2), {4, 5});
y = a + b; // Error: different minibatch sizes: 3 and 2.
y = b + a; // Error: different minibatch sizes: 2 and 3.

b = F::input<Node>({}, {4});
const Node c = F::input(Shape({}, 3), {5, 6, 7});
y = F::concat({a, b, c}, 0); // values: [1, 4, 5], [2, 4, 6], [3, 4, 7]
```

### 3.2.3 Scalar Operations

Elementwise binary operations such as **arithmetic operations** (`operator+`, `operator-`, `operator*` and `operator/`) and **exponentiation** (`primitiv::functions::pow`) supports the calculation between an arbitrary and scalar shapes. If a shape of one operand is a scalar, these functions broadcast the scalar value to all elements in the opposite side:

```
using primitiv::Node;
namespace F = primitiv::functions;

const Node a = F::input<Node>({3}, {1, 2, 3});
const Node b = F::input<Node>({}, {4});

Node y = a + b; // values: [5, 6, 7]
y = a - b; // values: [-3, -2, -1]
y = b - a; // values: [3, 2, 1]
```

## 3.3 Nodes and Tensors

### 3.3.1 Nodes

`primitiv` has two different classes to calculate neural networks: `Node` and `Tensor`. Nevertheless the basic usage of these classes are identical, the inner behavior of them is essentially different.

A `Node` object behaves a reference to an intermediate result of the network. Each `Node` object corresponds a `Device` object that represents the physical location of the calculated data, and a `Graph` object that the intermediate result belongs to. `Node` objects contain only a few information to identify the corresponding intermediate result in the `Graph` object, and have interfaces to communicate the `Graph` object to obtain actual data. Copying `Node` objects is typically a light operation.

#### Lazy Evaluation

Alithmetic operators between `Node` objects and functions defined in the `primitiv::functions` namespace register a new operation to the `Graph` object, and return a new `Node` object representing the result of the new operation. Actual calculation of each operation is postponed until the values are actually required. Once the operation is performed, the resulting values will be cached in the `Graph` object to prevent duplicated calculation.

Following examples show how `Node` objects work:

```

using namespace primitiv;
namespace F = primitiv::functions;

// Creating a `Node` object with no information: it does not point to any
// existing data.
const Node n0;

// Creating a `Device` and a `Graph` and setting them as the defaults.
devices::Naive dev;
Device::set_default(dev);
Graph g;
Graph::set_default(g);

// Creating two `Node` objects as the data sources of the computation graph.
const Node n1 = F::input<Node>({3}, {1, 2, 3});
const Node n2 = F::input<Node>({3}, {1, 1, 1});

// Creating a new `Node` object representing the result of some operations.
const Node n3 = n1 + n2;
const Node n33 = F::tanh(n1);

// Copying a `Node` object.
// This operation does not yield copying physical results.
const Node n4 = n3;

// Obtaining the actual results corresponding to a `Node` object.
// The `n1 + n2` operation will be actually performed here.
// And n33 is not calculated because it is not necessary to calculate `n4`.
const std::vector<float> values4 = n4.to_vector(); // {2, 3, 4}

// Defining an additional operation.
const Node n5 = n4 + F::input<Node>({3}, {3, 2, 1});

// Obtaining the result.
// The value represents `(n1 + n2) + {3, 2, 1}`, but the actual calculation
// will prevent the `n1 + n2` operation, and use the cached values of `n4`.
const std::vector<float> values5 = n5.to_vector(); // {5, 5, 5}

```

## Executing Backpropagation

Node object can perform the backpropagation. Unlike the *forward* operations described above, results of the backpropagation (*gradients* corresponding to Node objects) will be discarded whenever it is no longer used. To execute the backpropagation from a specified Node object (typically the Node representing the *sum of loss values*), users should call the `Node::backward()` function:

```

using namespace primitiv;
namespace F = primitiv::functions;

devices::Naive dev;
Device::set_default(dev);
Graph g;
Graph::set_default(g);

// Creating the graph with a `Parameter`.
Parameter p({3}, {0, 0, 0});
const Node w = F::parameter(p);

```

```
const Node x = F::input({3}, {1, 2, 3});
const Node y = w * x; // Elementwise multiplication

// Initializes the gradients of parameters.
p.reset_gradient();
const std::vector grad1 = y.gradient().to_vector(); // {0, 0, 0}

// Executing the backpropagation.
y.backward();

// All gradient values are disposed before arriving here.
const std::vector grad2 = y.gradient().to_vector(); // {1, 2, 3}
```

### 3.3.2 Tensor

Tensor class is another interface to calculate networks using similar interface with Node. Unlike the Node objects, Tensor objects hold actual resulting values of corresponding operations, and the calculation will be performed at the same time as creating new Tensor objects. Additionally, Tensor objects can not perform the backpropagation because they do not record the history of calculation.

Instead of these disadvantages, Tensor objects do not consume more memory than actual existence of all Tensor objects at the time, and do not yield any overhead of constructing computation graphs. Users can use Tensor instead of Node when users do not need the gradient information (e.g., testing trained models).

Following examples show how the Tensor objects work:

```
using namespace primitiv;
namespace F = primitiv::functions;

// Creating a `Tensor` object with no information: it does not point to any
// existing data.
const Tensor t0;

// Creating a `Device` and setting it as the default.
// `Tensor` objects do not require the `Graph` object.
devices::Naive dev;
Device::set_default(dev);

// Creating two `Tensor` objects with their own data.
const Tensor t1 = F::input<Tensor>({3}, {1, 2, 3});
const Tensor t2 = F::input<Tensor>({3}, {1, 1, 1});

// Creating a new `Tensor` object representing the result of some operations.
// The operations will be performed as soon as these statements are evaluated.
// And `t3` and `t33` hold their own values internally.
const Tensor t3 = t1 + t2;
const Tensor t33 = F::tanh(t1);

// Copying a `Tensor` object.
// This operation basically does not yield a large overhead.
// `n3` and `n4` shares the inner memory while they refers the same values.
const Tensor t4 = t3;

// Obtaining the inner values from a `Tensor` object.
const std::vector<float> values4 = n4.to_vector(); // {2, 3, 4}
```



## 3.4 Devices

TBD.

## 3.5 Default Device and Graph

TBD.

## 3.6 Training

TBD.

## 3.7 Models

TBD.

## 3.8 Saving and Loading

TBD.



This section contains low-level information about `primitiv`.

## 4.1 `primitiv` API Reference

### 4.1.1 Devices

#### Base Class

**class** `primitiv::Device`

Interface of the *Tensor* provider.

Inherits from `primitiv::mixins::DefaultSettable< Device >`, `primitiv::mixins::Nonmovable< Device >`

Subclassed by *primitiv::devices::CUDA*, *primitiv::devices::Eigen*, *primitiv::devices::Naive*, *primitiv::devices::OpenCL*

#### Public Types

**enum** `DeviceType`

*Device* type.

*Values:*

**GROUP\_FILTER** = 0xffff0000

**GROUP\_CPU** = 0x00000000

**CPU** = 0x00000000

**NAIVE** = 0x00000000

**EIGEN** = 0x00000001

**GROUP\_CUDA** = 0x00010000

CUDA = 0x00010000

GROUP\_OPENCL = 0x00020000

OPENCL = 0x00020000

## Public Functions

**virtual** void **dump\_description** () **const** = 0

Prints device description to stderr.

**virtual** *DeviceType* **type** () **const** = 0

Retrieves the type of the device.

**Return** A *DeviceType* value.

*Tensor* **new\_tensor\_by\_constant** (**const** *Shape* &*shape*, float *k*)

Provides a new *Tensor* object with same-value elements.

**Return** A new *Tensor* object.

### Parameters

- *shape*: *Shape* of the tensor.
- *k*: Constant to initialize elements.

*Tensor* **new\_tensor\_by\_array** (**const** *Shape* &*shape*, **const** float *values*[])

Provides a new *Tensor* object with specific values.

**Return** A new *Tensor* object.

### Parameters

- *shape*: *Shape* of the tensor.
- *values*: Pointer to array of internal values.

*Tensor* **new\_tensor\_by\_vector** (**const** *Shape* &*shape*, **const** std::vector<float> &*values*)

Provides a new *Tensor* object with specific values.

**Return** A new *Tensor* object.

### Parameters

- *shape*: *Shape* of the tensor.
- *values*: List of internal values.

*Tensor* **copy\_tensor** (**const** *Tensor* &*x*)

Copies the tensor to this device with allocating a new memory.

**Return** Copied tensor.

**Remark** The value of *x* is always duplicated, and the internal memory of the resulting tensor becomes always different from *x* even if *x.device* () is same as *this*.

### Parameters

- *x*: A tensor to be copied.

void **inplace\_multiply\_const** (float *k*, *Tensor* &*x*)

Directly multiplies all elements by a constant.

**Parameters**

- k: A constant to multiply.
- x: A tensor to be updated.

void **inplace\_add** (const *Tensor* &x, *Tensor* &y)

Directly adds the first tensor to the second tensor.

**Remark** This method keeps the shape of *y*, and the behavior is conditioned according to the batch size of *y* and *x*: *y*.shape == *x*.shape: *y* += *x* *y*.shape == 1: *y* += batch\_sum(*x*) *x*.shape == 1: *y* += batch\_broadcast(*x*) otherwise: error.

**Parameters**

- x: A tensor to add.
- y: A tensor to be updated.

void **inplace\_subtract** (const *Tensor* &x, *Tensor* &y)

Directly subtracts the first tensor from the second tensor.

**Remark** The batch broadcasting behavior of this method is same as that of `inplace_add`.

**Parameters**

- x: A tensor to subtract.
- y: A tensor to be updated.

**Inherited Classes**

**class** `primitiv::devices::Naive`

*Device* class for the naive function implementations on CPU.

Inherits from `primitiv::Device`

**Public Functions**

**Naive** ()

Creates a *Naive* object.

**Naive** (std::uint32\_t *seed*)

Creates a *Naive* object.

**Parameters**

- seed: The seed value of internal random number generator.

void **dump\_description** () **const**

Prints device description to stderr.

*Device::DeviceType* **type** () **const**

Retrieves the type of the device.

**Return** A *DeviceType* value.

**class** `primitiv::devices::Eigen`

*Device* class for the Eigen3 backend.

Inherits from `primitiv::Device`

## Public Functions

**Eigen** ()

Creates a *Eigen* object.

**Eigen** (std::uint32\_t *seed*)

Creates a *Eigen* object.

### Parameters

- *seed*: The seed value of internal random number generator.

void **dump\_description** () const

Prints device description to stderr.

*Device::DeviceType* **type** () const

Retrieves the type of the device.

**Return** A DeviceType value.

**class** pseudoprimitiv::devices::CUDA

*Device* class for *CUDA*.

Inherits from *pseudoprimitiv::Device*

## Public Functions

**CUDA** (std::uint32\_t *device\_id*)

Creates a new *CUDA* device.

**Remark** The random number generator is initialized using `std::random_device`.

### Parameters

- *device\_id*: ID of the physical GPU.

**CUDA** (std::uint32\_t *device\_id*, std::uint32\_t *rng\_seed*)

Creates a new *CUDA* device.

### Parameters

- *device\_id*: ID of the physical GPU.
- *rng\_seed*: The seed value of the random number generator.

void **dump\_description** () const

Prints device description to stderr.

*Device::DeviceType* **type** () const

Retrieves the type of the device.

**Return** A DeviceType value.

## Public Static Functions

**static** std::uint32\_t **num\_devices** ()

Retrieves the number of active hardwares.

**Return** Number of active hardwares.

**static void assert\_support** (std::uint32\_t *device\_id*)

Checks whether the device corresponding to the specified ID is supported.

#### Parameters

- *device\_id*: *Device* ID to check.

#### Exceptions

- `primitiv::Error`: This class does not support the specified device.

**static bool check\_support** (std::uint32\_t *device\_id*)

Checks whether the device corresponding to the specified ID is supported.

**Return** true if this class supports the specified device, false otherwise.

#### Parameters

- *device\_id*: *Device* ID to check.

**class** `primitiv::devices::OpenCL`

*Device* class for *OpenCL*.

Inherits from *primitiv::Device*

### Public Functions

**OpenCL** (std::uint32\_t *platform\_id*, std::uint32\_t *device\_id*)

Creates a new *OpenCL* device.

#### Parameters

- *platform\_id*: Platform ID.
- *device\_id*: *Device* ID on the selected platform.

**OpenCL** (std::uint32\_t *platform\_id*, std::uint32\_t *device\_id*, std::uint32\_t *rng\_seed*)

Creates a new *OpenCL* device.

#### Parameters

- *platform\_id*: Platform ID.
- *device\_id*: *Device* ID on the selected platform.
- *rng\_seed*: Seed value of the random number generator.

void **dump\_description** () **const**

Prints device description to stderr.

*Device::DeviceType* **type** () **const**

Retrieves the type of the device.

**Return** A *DeviceType* value.

### Public Static Functions

**static** std::uint32\_t **num\_platforms** ()

Retrieves the number of active platforms.

**Return** Number of active platforms.

**static** std::uint32\_t **num\_devices** (std::uint32\_t *platform\_id*)  
Retrieves the number of active devices on the specified platform.

**Return** Number of active devices.

**Parameters**

- *platform\_id*: Platform ID. This value should be between 0 to *num\_platforms()* - 1.

**static** void **assert\_support** (std::uint32\_t *platform\_id*, std::uint32\_t *device\_id*)  
Checks whether the device corresponding to the specified IDs is supported.

**Parameters**

- *platform\_id*: Platform ID to check.
- *device\_id*: *Device* ID to check.

**Exceptions**

- `primitiv::Error`: This class does not support the specified device.

**static** bool **check\_support** (std::uint32\_t *platform\_id*, std::uint32\_t *device\_id*)  
Checks whether the device corresponding to the specified ID is supported.

**Return** true if this class supports the specified device, false otherwise.

**Parameters**

- *platform\_id*: Platform ID to check.
- *device\_id*: *Device* ID to check.

## 4.1.2 Functions

This page describes the basic/composite functions implemented in `primitiv`. They return a template type `Var`, and take 0 or more number of references of `Var` as their arguments. `Var` becomes either `Node` or `Tensor` according to the usage:

```
primitiv::Node x = ...;
primitiv::Tensor w = ...;
auto y = primitiv::functions::tanh(x); // `y` becomes a `Node`.
auto u = primitiv::functions::exp(w); // `u` becomes a `Tensor`.
```

If the function has no argument with type `Var`, you must specify the template argument appropriately:

```
auto x = primitiv::functions::input<Node>(...); // `x` becomes a `Node`.
auto w = primitiv::functions::parameter<Tensor>(...); // `w` becomes a `Tensor`.
```

**namespace** `primitiv::functions`

### Functions

**template** <typename `Var`>  
`type_traits::Identity<Var>` **positive** (const `Var` &*x*)

Applies a unary  $+$  operation. This function does not change any values of the argument, and returns a copy of it.

**Return** A variable representing  $+x$ .

**Parameters**



- $x$ : A variable representing an argument  $x$ .

**template** <typename Var>

type\_traits::Identity<Var> **negative** (const Var & $x$ )

Applies a unary  $-$  operation.

**Return** A variable representing  $-x$ .

#### Parameters

- $x$ : A variable representing an argument  $x$ .

**template** <typename Var>

type\_traits::Identity<Var> **add** (const Var & $x$ , float  $k$ )

Applies an elementwise addition between a variable and a constant.

**Return** A variable representing  $x + k$ .

#### Parameters

- $x$ : A variable representing an argument  $x$ .
- $k$ : A constant  $k$ .

**template** <typename Var>

type\_traits::Identity<Var> **add** (float  $k$ , const Var & $x$ )

Applies an elementwise addition between a constant and a variable.

**Return** A variable representing  $k + x$ .

#### Parameters

- $k$ : A constant  $k$ .
- $x$ : A variable representing an argument  $x$ .

**template** <typename Var>

type\_traits::Identity<Var> **add** (const Var & $a$ , const Var & $b$ )

Applies an elementwise addition between two variables.

**Return** A variable representing  $a + b$ .

#### Parameters

- $a$ : A variable representing an argument  $a$ .
- $b$ : A variable representing an argument  $b$ .

**template** <typename Var>

type\_traits::Identity<Var> **subtract** (const Var & $x$ , float  $k$ )

Applies an elementwise subtraction between a variable and a constant.

**Return** A variable representing  $x - k$ .

#### Parameters

- $x$ : A variable representing an argument  $x$ .
- $k$ : A constant  $k$ .

**template** <typename Var>

type\_traits::Identity<Var> **subtract** (float  $k$ , const Var & $x$ )

Applies an elementwise subtraction between a constant and a variable.

**Return** A variable representing  $k - x$ .

#### Parameters

- $k$ : A constant  $k$ .

- $x$ : A variable representing an argument  $x$ .

**template** <typename Var>

type\_traits::Identity<Var> **subtract** (const Var & $a$ , const Var & $b$ )

Applies an elementwise subtraction between two variables.

**Return** A variable representing  $a - b$ .

**Parameters**

- $a$ : A variable representing an argument  $a$ .
- $b$ : A variable representing an argument  $b$ .

**template** <typename Var>

type\_traits::Identity<Var> **multiply** (const Var & $x$ , float  $k$ )

Applies an elementwise multiplication between a variable and a constant.

**Return** A variable representing  $x \times k$ .

**Parameters**

- $x$ : A variable representing an argument  $x$ .
- $k$ : A constant  $k$ .

**template** <typename Var>

type\_traits::Identity<Var> **multiply** (float  $k$ , const Var & $x$ )

Applies an elementwise multiplication between a constant and a variable.

**Return** A variable representing  $k \times x$ .

**Parameters**

- $k$ : A constant  $k$ .
- $x$ : A variable representing an argument  $x$ .

**template** <typename Var>

type\_traits::Identity<Var> **multiply** (const Var & $a$ , const Var & $b$ )

Applies an elementwise multiplication between two variables.

**Return** A variable representing  $a \times b$ .

**Parameters**

- $a$ : A variable representing an argument  $a$ .
- $b$ : A variable representing an argument  $b$ .

**template** <typename Var>

type\_traits::Identity<Var> **divide** (const Var & $x$ , float  $k$ )

Applies an elementwise division between a variable and a constant.

**Return** A variable representing  $x/k$ .

**Parameters**

- $x$ : A variable representing an argument  $x$ .
- $k$ : A constant  $k$ .

**template** <typename Var>

type\_traits::Identity<Var> **divide** (float  $k$ , const Var & $x$ )

Applies an elementwise division between a constant and a variable.

**Return** A variable representing  $k/x$ .

**Parameters**

- $k$ : A constant  $k$ .
- $x$ : A variable representing an argument  $x$ .

**template** <typename Var>

type\_traits::Identity<Var> **divide** (const Var & $a$ , const Var & $b$ )

Applies an elementwise division between two variables.

**Return** A variable representing  $a/b$ .

#### Parameters

- $a$ : A variable representing an argument  $a$ .
- $b$ : A variable representing an argument  $b$ .

**template** <typename Var>

type\_traits::Identity<Var> **pow** (const Var & $x$ , float  $k$ )

Applies an elementwise exponentiation between a variable and a constant.

**Return** A variable representing  $x^k$ .

#### Parameters

- $x$ : A variable representing an argument  $x$ .
- $k$ : A constant  $k$ .

**template** <typename Var>

type\_traits::Identity<Var> **pow** (float  $k$ , const Var & $x$ )

Applies an elementwise exponentiation between a constant and a variable.

**Return** A variable representing  $k^x$ .

#### Parameters

- $k$ : A constant  $k$ .
- $x$ : A variable representing an argument  $x$ .

**template** <typename Var>

type\_traits::Identity<Var> **pow** (const Var & $a$ , const Var & $b$ )

Applies an elementwise exponentiation between two variables.

**Return** A variable representing  $a^b$ .

#### Parameters

- $a$ : A variable representing an argument  $a$ .
- $b$ : A variable representing an argument  $b$ .

**template** <typename Var>

type\_traits::Identity<Var> **pown** (const Var & $x$ , std::int32\_t  $k$ )

Applies an elementwise exponentiation between a variable and an integer constant. This function can be applied correctly when  $x$  has some negative values.

**Return** A variable representing  $x^k$ .

#### Parameters

- $x$ : A variable representing an argument  $x$ .
- $k$ : An integer constant  $k$ .

*Tensor* **input\_tensor** (const *Shape* & $shape$ , const std::vector<float> & $data$ , *Device* \* $dev$ )

Creates a new *Tensor* from specific shape and data.

**Return** A new *Tensor*.

**Parameters**

- shape: *Shape* of the new *Tensor*.
- data: Inner data of the new *Tensor*. `data.size()` should be equal to `shape.size()` and each data is ordered by the column-major order.
- dev: *Device* to manage inner data of the *Tensor*, or `nullptr` to use the default device.

*Node* **input\_node** (**const** *Shape* &shape, **const** std::vector<float> &data, *Device* \*dev, *Graph* \*g)  
Creates a new *Node* from specific shape and data.

**Return** A new *Node*.

**Parameters**

- shape: *Shape* of the new *Node*.
- data: Inner data of the new *Node*. `data.size()` should be equal to `shape.size()` and each data is ordered by the column-major order.
- dev: *Device* to manage inner data of the *Node*, or `nullptr` to use the default device.
- g: *Graph* to manage the instance of the *Node*, or `nullptr` to use the default graph.

**template** <typename Var>  
type\_traits::Identity<Var> **input** (**const** *Shape* &shape, **const** std::vector<float> &data, *Device* \*dev)  
Creates a new variable from specific shape and data.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Parameters**

- shape: *Shape* of the new variable.
- data: Inner data of the new variable. `data.size()` should be equal to `shape.size()` and each data is ordered by the column-major order.
- dev: *Device* to manage inner data of the variable, or `nullptr` to use the default device.

**template** <typename Var>  
type\_traits::Identity<Var> **input** (**const** *Shape* &shape, **const** std::vector<float> &data, *Device* &dev)  
Creates a new variable from specific shape and data.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Parameters**

- shape: *Shape* of the new variable.
- data: Inner data of the new variable. `data.size()` should be equal to `shape.size()` and each data is ordered by the column-major order.
- dev: *Device* to manage inner data of the variable.

**template** <typename Var>  
type\_traits::Identity<Var> **input** (**const** *Shape* &shape, **const** std::vector<float> &data)  
Creates a new variable from specific shape and data.

**Return** A new variable.

**Remark** This function always uses the default device, and also uses the default graph when specifying *Node* as the template variable.

**Parameters**

- `shape`: *Shape* of the new variable.
- `data`: Inner data of the new variable. `data.size()` should be equal to `shape.size()` and each data is ordered by the column-major order.

**Tensor** `parameter_tensor` (*Parameter* &param)

Creates a new *Tensor* from a specific *Parameter*.

**Return** A new *Tensor*.

**Parameters**

- `param`: *Parameter* to be associated with the *Tensor*.

**Node** `parameter_node` (*Parameter* &param, *Graph* \*g)

Creates a new *Node* from a specific *Parameter*.

**Return** A new *Node*.

**Parameters**

- `param`: *Parameter* to be associated with the *Node*.
- `g`: *Graph* to manage the instance of the *Node*, or `nullptr` to use the default graph.

**template** <typename Var>

`type_traits::Identity<Var> parameter` (*Parameter* &param)

Creates a new variable from a specific *Parameter*.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Parameters**

- `param`: *Parameter* to be associated with the variable.

**template** <typename Var>

`type_traits::Identity<Var> copy` (`const` Var &x, *Device* \*dev)

Copies a variable onto a specific device.

**Return** A new variable managed on dev.

**Parameters**

- `x`: A variable to be copied.
- `dev`: *Device* to manage the new variable, or `nullptr` to use the default device.

**template** <typename Var>

`type_traits::Identity<Var> copy` (`const` Var &x, *Device* &dev)

Copies a variable onto a specific device.

**Return** A new variable managed on dev.

**Parameters**

- `x`: A variable to be copied.
- `dev`: *Device* to manage the new variable.

**template** <typename Var>

`type_traits::Identity<Var> copy` (`const` Var &x)

Copies a variable onto the default device.

**Return** A new variable managed on the default device.

### Parameters

- `x`: A variable to be copied.

**template** <typename Var>

type\_traits::Identity<Var> **pick**(const Var &`x`, const std::vector<std::uint32\_t> &`ids`, std::uint32\_t *dim*)

Lookups subplanes according to the specific axis and addresses. This function can be used to an embedding lookup associated with a fixed vocabulary. Following examples show how this function work:

$$\begin{aligned}
 x &:= \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}, \\
 \text{pick}(x, [0, 0, 1], 0) &= \begin{pmatrix} 1 & 4 & 7 \end{pmatrix}, \begin{pmatrix} 1 & 4 & 7 \end{pmatrix}, \begin{pmatrix} 2 & 5 & 8 \end{pmatrix}, \\
 \text{pick}(x, [1, 2], 1) &= \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}, \begin{pmatrix} 7 \\ 8 \\ 9 \end{pmatrix}, \\
 \text{pick}(x, [0], 2) &= \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}.
 \end{aligned}$$

The minibatch broadcasting rule is applied between the *Shape* of `x` and the number of values in `ids`:

$$\begin{aligned}
 x &:= \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}, \begin{pmatrix} 11 & 14 & 17 \\ 12 & 15 & 18 \\ 13 & 16 & 19 \end{pmatrix}, \begin{pmatrix} 21 & 24 & 27 \\ 22 & 25 & 28 \\ 23 & 26 & 29 \end{pmatrix}, \\
 \text{pick}(x, [0], 1) &= \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}, \begin{pmatrix} 14 \\ 15 \\ 16 \end{pmatrix}, \begin{pmatrix} 24 \\ 25 \\ 26 \end{pmatrix}, \\
 \text{pick}(x, [0, 1, 2], 1) &= \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \begin{pmatrix} 14 \\ 15 \\ 16 \end{pmatrix}, \begin{pmatrix} 27 \\ 28 \\ 29 \end{pmatrix}.
 \end{aligned}$$

**Return** A new variable.

### Parameters

- `x`: A variable representing an original data.
- `ids`: List of subplane IDs according to the axis `dim`. Each value must be lower than `x.shape()[dim]`.
- `dim`: Axis to be processed.

**template** <typename Var>

type\_traits::Identity<Var> **slice**(const Var &`x`, std::uint32\_t *dim*, std::uint32\_t *lower*, std::uint32\_t *upper*)

Extracts a specific range  $[L, U)$  of subplanes along a specific axis. Following examples show how this function work:

$$\begin{aligned}
 x &:= \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}, \\
 \text{slice}(x, 0, 0, 1) &= \begin{pmatrix} 1 & 4 & 7 \end{pmatrix}, \\
 \text{slice}(x, 1, 1, 3) &= \begin{pmatrix} 4 & 7 \\ 5 & 8 \\ 6 & 9 \end{pmatrix}, \\
 \text{slice}(x, 2, 0, 1) &= \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}.
 \end{aligned}$$

**Return** A new variable.

**Parameters**

- `x`: A variable representing an original data.
- `dim`: Axis to be processed.
- `lower`: Lower bound  $L$  of `dim`.
- `upper`: Upper bound  $U$  of `dim`.

**template** <typename Container>

`type_traits::Reduce<Container> concat (const Container &xs, std::uint32_t dim)`

Concatenates multiple variables along specific axis. Following examples show how this function work:

$$\begin{aligned}
 x_1 &:= \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}, \\
 x_2 &:= \begin{pmatrix} 11 & 14 & 17 \\ 12 & 15 & 18 \\ 13 & 16 & 19 \end{pmatrix}, \\
 \text{concat}([x_1, x_2], 0) &= \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \\ 11 & 14 & 17 \\ 12 & 15 & 18 \\ 13 & 16 & 19 \end{pmatrix}, \\
 \text{concat}([x_1, x_2], 1) &= \begin{pmatrix} 1 & 4 & 7 & 11 & 14 & 17 \\ 2 & 5 & 8 & 12 & 15 & 18 \\ 3 & 6 & 9 & 13 & 16 & 19 \end{pmatrix}, \\
 \text{concat}([x_1, x_2], 2) &= \left( \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}, \begin{pmatrix} 11 & 14 & 17 \\ 12 & 15 & 18 \\ 13 & 16 & 19 \end{pmatrix} \right).
 \end{aligned}$$

**Return** A new variable.

**Parameters**

- `xs`: Iterable container of variables. `xs` must have both `begin()` and `end()` functions that return the begin/end iterators.
- `dim`: Axis to be processed.

**template** <typename Container>

`type_traits::ReducePtr<Container> concat (const Container &xs, std::uint32_t dim)`

Same as above, but `xs` has pointers of variables.

**Return** A new variable.

**Parameters**

- `xs`: Iterable container of pointers of variables. `xs` must have both `begin()` and `end()` functions that return the begin/end iterators.
- `dim`: Axis to be processed.

**template** <typename Var>

`type_traits::Identity<Var> reshape (const Var &x, const Shape &new_shape)`

Changes the *Shape* of the variable.

**Return** A new variable.

**Parameters**

- `x`: A variable with an old *Shape*.
- `new_shape`: A new *Shape* to be applied to the new variable. The volume and the minibatch size must be same as that of `x.shape()`.

**template** <typename Var>

`type_traits::Identity<Var> flatten (const Var &x)`

Changes the *Shape* of the variable to the column vector.

**Return** A new variable.

**Parameters**

- `x`: A variable with an old *Shape*.

**template** <typename Var>

`type_traits::Identity<Var> transpose (const Var &x)`

Applies a matrix transposition.

**Return** A new variable representing  $X^T$ .

**Parameters**

- `x`: A variable representing an argument  $X$ . The shape of `x` must be either a scalar, a column vector or a matrix.

**template** <typename Var>

`type_traits::Identity<Var> matmul (const Var &a, const Var &b)`

Applies a matrix multiplication between two matrices.

**Return** A new variable representing  $AB$ .

**Parameters**

- `a`: A variable representing an argument  $A$ . The shape of `a` must be either a scalar, a column vector or a matrix.
- `b`: A variable representing an argument  $B$ . The shape of `b` must be either a scalar, a column vector or a matrix, and `b.shape()[0]` must be equal to `a.shape()[1]`.

**template** <typename Var>

`type_traits::Identity<Var> sqrt (const Var &x)`

Applies an elementwise square root function.

**Return** A variable representing  $\sqrt{x}$ .

**Parameters**

- `x`: A variable representing an argument  $x$ .

**template** <typename Var>

`type_traits::Identity<Var> exp (const Var &x)`

Applies an elementwise exponential function.

**Return** A variable representing  $e^x$ .

**Parameters**

- `x`: A variable representing an argument  $x$ .

**template** <typename Var>

`type_traits::Identity<Var> log (const Var &x)`

Applies an elementwise natural logarithm function.

**Return** A variable representing  $\ln(x)$ .

**Parameters**

- `x`: A variable representing an argument  $x$ .



**template** <typename Var>  
 type\_traits::Identity<Var> **tanh** (const Var &x)  
 Applies an elementwise hyperbolic tangent function.

**Return** A variable representing  $\tanh(x)$ .

**Parameters**

- x: A variable representing an argument  $x$ .

**template** <typename Var>  
 type\_traits::Identity<Var> **sigmoid** (const Var &x)  
 Applies an elementwise logistic sigmoid function:

$$\text{sigmoid}(x) := \frac{1}{1 + e^{-x}}.$$

**Return** A variable representing  $\text{sigmoid}(x)$ .

**Parameters**

- x: A variable representing an argument  $x$ .

**template** <typename Var>  
 type\_traits::Identity<Var> **softplus** (const Var &x)  
 Applies an elementwise softplus function:

$$\text{softplus}(x) := \ln(1 + e^x).$$

**Return** A variable representing  $\text{softplus}(x)$ .

**Parameters**

- x: A variable representing an argument  $x$ .

**template** <typename Var>  
 type\_traits::Identity<Var> **sin** (const Var &x)  
 Applies an elementwise sin function.

**Return** A variable representing  $\sin(x)$ .

**Parameters**

- x: A variable representing an argument  $x$ .

**template** <typename Var>  
 type\_traits::Identity<Var> **cos** (const Var &x)  
 Applies an elementwise cos function.

**Return** A variable representing  $\cos(x)$ .

**Parameters**

- x: A variable representing an argument  $x$ .

**template** <typename Var>  
 type\_traits::Identity<Var> **tan** (const Var &x)  
 Applies an elementwise tangent function.

**Return** A variable representing  $\tan(x)$ .

**Parameters**

- x: A variable representing an argument  $x$ .

**template** <typename Var>

```
type_traits::Identity<Var> relu (const Var &x)
```

Applies an elementwise rectified linear unit (ReLU) function:

$$\text{ReLU}(x) := \max(x, 0).$$

**Return** A variable representing  $\text{ReLU}(x)$ .

#### Parameters

- $x$ : A variable representing an argument  $x$ .

```
template <typename Var>
```

```
type_traits::Identity<Var> lrelu (const Var &x)
```

Applies an elementwise leaky ReLU function:

$$\text{LReLU}(x) := \max(x, 0.01x).$$

**Return** A variable representing  $\text{LReLU}(x)$ .

#### Parameters

- $x$ : A variable representing an argument  $x$ .

```
template <typename Var>
```

```
type_traits::Identity<Var> prelu (const Var &x, float a)
```

Applies an elementwise parameterized ReLU function:

$$\text{PReLU}(x) := \begin{cases} x, & \text{if } x \geq 0, \\ \alpha x, & \text{otherwise.} \end{cases}$$

**Return** A variable representing  $\text{PReLU}(x)$ .

#### Parameters

- $x$ : A variable representing an argument  $x$ .
- $a$ : A scaling factor  $\alpha$ .

```
template <typename Var>
```

```
type_traits::Identity<Var> elu (const Var &x, float a)
```

Applies an elementwise exponential linear unit (ELU) function:

$$\text{ELU}(x) := \begin{cases} x, & \text{if } x \geq 0, \\ \alpha(e^x - 1), & \text{otherwise.} \end{cases}$$

**Return** A variable representing  $\text{ELU}(x)$ .

#### Parameters

- $x$ : A variable representing an argument  $x$ .
- $a$ : A scaling factor  $\alpha$ .

```
template <typename Var>
```

```
type_traits::Identity<Var> sum (const Var &x, std::uint32_t dim)
```

Applies summation along an axis. Following examples show how this function work:

$$\begin{aligned} x &:= \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}, \\ \text{sum}(x, 0) &= \begin{pmatrix} 6 & 15 & 24 \end{pmatrix}, \\ \text{sum}(x, 1) &= \begin{pmatrix} 12 \\ 15 \\ 18 \end{pmatrix}, \\ \text{sum}(x, 2) &= \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}. \end{aligned}$$

**Return** A new variable.

**Parameters**

- `x`: A variable representing values before reduction.
- `dim`: Axis to be processed.

**template** <typename Var>

type\_traits::Identity<Var> **broadcast** (const Var &*x*, std::uint32\_t *dim*, std::uint32\_t *size*)

Applies broadcasting along an axis. Following examples show how this function work:

$$\begin{aligned}
 x_1 &:= \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}, \\
 \text{broadcast}(x_1, 0, 3) &= \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}, \\
 x_2 &:= \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \\
 \text{broadcast}(x_2, 1, 3) &= \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix}, \\
 \text{broadcast}(x_2, 2, 3) &= \left( \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \right).
 \end{aligned}$$

**Return** A new variable.

**Parameters**

- `x`: A variable representing values before reduction.
- `dim`: Axis to be processed.
- `size`: New size of the axis `dim`.

**template** <typename Var>

type\_traits::Identity<Var> **logsumexp** (const Var &*x*, std::uint32\_t *dim*)

Applies a logsumexp reduction along an axis. This function performs similarly to `primitiv::functions::sum` w.r.t. the axis.

**Return** A new variable.

**Parameters**

- `x`: A variable representing values before expansion.
- `dim`: Axis to be processed.

**template** <typename Var>

type\_traits::Identity<Var> **log\_softmax** (const Var &*x*, std::uint32\_t *dim*)

Applies a softmax operation along an axis, and returns the natural logarithm of resulting values.

**Return** A new variable.

**Parameters**

- `x`: A variable representing original values.
- `dim`: Axis to be processed.

**template** <typename Var>

type\_traits::Identity<Var> **softmax** (const Var &*x*, std::uint32\_t *dim*)

Applies a softmax operation along an axis.

**Return** A new variable.

### Parameters

- x: A variable representing original values.
- dim: Axis to be processed.

```
template <typename Var>
type_traits::Identity<Var> softmax_cross_entropy (const Var &x, const Var &t, std::uint32_t
                                                    dim)
```

Applies a softmax cross entropy function between two variables along an axis.

**Return** A new variable.

### Parameters

- x: A variable representing logit values.
- t: A variable representing ground-truth distribution along the axis dim.
- dim: Axis to be processed.

```
template <typename Var>
type_traits::Identity<Var> softmax_cross_entropy (const Var &x, const
                                                    std::vector<std::uint32_t> &ids, std::uint32_t
                                                    dim)
```

Applies a softmax cross entropy function between logits and one-hot distributions along an axis.

**Return** A new variable.

### Parameters

- x: A variable representing logit values.
- ids: List of one-hot IDs along the axis dim. Each value must be lower than x.shape()[dim].
- dim: Axis to be processed.

```
template <typename Var>
type_traits::Identity<Var> stop_gradient (const Var &x)
```

Blocks the gradient propagation beyond this function. This function does not modify any values in the input variable, and force to make all gradients 0.

**Return** A new variable.

### Parameters

- x: A variable representing original values.

```
template <typename Var>
type_traits::Identity<Var> conv2d (const Var &x, const Var &w, std::uint32_t padding0,
                                   std::uint32_t padding1, std::uint32_t stride0, std::uint32_t stride1,
                                   std::uint32_t dilation0, std::uint32_t dilation1)
```

Applies a 2D convolution between two variables.

**Return** A new variable with *Shape*  $[d'_0, d'_1, c_2]$ . The first and second dimension are calculated as following:

$$d'_i := \frac{d_i + 2 \times \text{padding}_i - (u_i - 1) \times \text{dilation}_i + 1}{\text{stride}_i} + 1.$$

### Parameters

- x: A variable with *Shape*  $[d_0, d_1, c_1]$ .
- w: A variable with *Shape*  $[u_0, u_1, c_1, c_2]$ .
- padding0: Width of zero-padding along the first axis.

- `padding1`: Width of zero-padding along the second axis.
- `stride0`: Stride along the first axis.
- `stride1`: Stride along the second axis.
- `dilation0`: Dilation factor along the first axis.
- `dilation1`: Dilation factor along the second axis.

**template** <typename Var>

type\_traits::Identity<Var> **max\_pool2d** (const Var &x, std::uint32\_t window0, std::uint32\_t window1, std::uint32\_t padding0, std::uint32\_t padding1, std::uint32\_t stride0, std::uint32\_t stride1)

Applies a 2D max-pooling operation.

**Return** A new variable with *Shape*  $[d'_0, d'_1, c]$ . The first and second dimension are calculated as following:

$$d'_i := \frac{d_i + 2 \times \text{padding}_i - \text{window}_i}{\text{stride}_i} + 1.$$

#### Parameters

- `x`: A variable with *Shape*  $[d_0, d_1, c]$ .
- `window0`: Window size along the first axis.
- `window1`: Window size along the second axis.
- `padding0`: Width of  $-\infty$  padding along the first axis.
- `padding1`: Width of  $-\infty$  padding along the second axis.
- `stride0`: Stride along the first axis.
- `stride1`: Stride along the second axis.

*Tensor* **constant\_tensor** (const *Shape* &shape, float k, *Device* \*dev)

Creates a new *Tensor* with all values the constant  $k$ .

**Return** A new *Tensor*.

#### Parameters

- `shape`: *Shape* of the new *Tensor*.
- `k`: The constant  $k$  of values in the *Tensor*.
- `dev`: *Device* to manage the new *Tensor*, or `nullptr` to use the default device.

*Node* **constant\_node** (const *Shape* &shape, float k, *Device* \*dev, *Graph* \*g)

Creates a new *Node* with all values the constant  $k$ .

**Return** A new *Node*.

#### Parameters

- `shape`: *Shape* of the new *Node*.
- `k`: The constant  $k$  of values in the *Node*.
- `dev`: *Device* to manage the new *Node*, or `nullptr` to use the default device.
- `g`: *Graph* to manage the instance of the *Node*, or `nullptr` to use the default graph.

**template** <typename Var>

type\_traits::Identity<Var> **constant** (const *Shape* &shape, float k, *Device* \*dev)

Creates a new variable with all values the constant  $k$ .

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Parameters**

- shape: *Shape* of the new variable.
- k: The constant  $k$  of values in the variable.
- dev: *Device* to manage the new variable, or `nullptr` to use the default device.

**template** <typename Var>

`type_traits::Identity<Var> constant (const Shape &shape, float  $k$ , Device &dev)`

Creates a new variable with all values the constant  $k$ .

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Parameters**

- shape: *Shape* of the new variable.
- k: The constant  $k$  of values in the variable.
- dev: *Device* to manage the new variable.

**template** <typename Var>

`type_traits::Identity<Var> constant (const Shape &shape, float  $k$ )`

Creates a new variable with all values the constant  $k$ .

**Remark** This function always uses the default device, and also uses the default graph when specifying *Node* as the template variable.

**Parameters**

- shape: *Shape* of the new variable.
- k: The constant  $k$  of values in the variable.

*Tensor identity\_tensor* (std::uint32\_t size, *Device* \*dev)

Creates a new *Tensor* with an  $N$ -dimensional identity matrix.

**Return** A new *Tensor*.

**Parameters**

- size: Size  $N$  of the matrix in the new *Tensor*.
- dev: *Device* to manage the new *Tensor*, or `nullptr` to use the default device.

*Node identity\_node* (std::uint32\_t size, *Device* \*dev, *Graph* \*g)

Creates a new *Node* with an  $N$ -dimensional identity matrix.

**Return** A new *Node*.

**Parameters**

- size: Size  $N$  of the matrix in the new *Node*.
- dev: *Device* to manage the new *Node*, or `nullptr` to use the default device.
- g: *Graph* to manage the instance of the *Node*, or `nullptr` to use the default graph.

**template** <typename Var>

`type_traits::Identity<Var> identity (std::uint32_t size, Device *dev)`

Creates a new variable with an  $N$ -dimensional identity matrix.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Parameters**

- `size`: Size  $N$  of the matrix in the new *Node*.
- `dev`: *Device* to manage the new variable, or `nullptr` to use the default device.

**template** <typename Var>

`type_traits::Identity<Var> identity (std::uint32_t size, Device &dev)`

Creates a new variable with an  $N$ -dimensional identity matrix.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Parameters**

- `size`: Size  $N$  of the matrix in the new *Node*.
- `dev`: *Device* to manage the new variable.

**template** <typename Var>

`type_traits::Identity<Var> identity (std::uint32_t size)`

Creates a new variable with an  $N$ -dimensional identity matrix.

**Return** A new variable.

**Remark** This function always uses the default device, and also uses the default graph when specifying *Node* as the template variable.

**Parameters**

- `size`: Size  $N$  of the matrix in the new *Node*.

**template** <typename Var>

`type_traits::Identity<Var> selu (const Var &x, float a = 1.6732632423543772848170429916717, float  
s = 1.0507009873554804934193349852946)`

Applies an elementwise scaled ELU function:

$$\text{SELU}(x) := s \times \begin{cases} x, & \text{if } x \geq 0, \\ \alpha(e^x - 1), & \text{otherwise.} \end{cases}$$

**Return** A variable representing  $\text{SELU}(x)$ .

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- `x`: A variable representing an argument  $x$ .
- `a`: A scaling factor  $\alpha$ .
- `s`: Another scaling factor  $s$ .

**template** <typename Container>

`type_traits::Reduce<Container> sum (const Container &xs)`

Applies summation along variables in the container.

**Return** A new variable.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- `xs`: Iterable container of variables. `xs` must have both `begin()` and `end()` functions that return the begin/end iterators.

**template** <typename Container>

`type_traits::ReducePtr<Container> sum (const Container &xs)`

Same as above, but `xs` has pointers of variables.

**Return** A new variable.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- `xs`: Iterable container of pointers of variables. `xs` must have both `begin()` and `end()` functions that return the begin/end iterators.

`template <typename Var>`

`type_traits::Identity<Var> mean (const Var &x, std::uint32_t dim)`

Calculates means along an axis.

**Return** A new variable.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- `x`: A variable representing values before reduction.
- `dim`: Axis to be processed.

`template <typename Container>`

`type_traits::Reduce<Container> mean (const Container &xs)`

Calculates means along variables in the container.

**Return** A new variable.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- `xs`: Iterable container of variables. `xs` must have both `begin()` and `end()` functions that return the begin/end iterators.

`template <typename Container>`

`type_traits::ReducePtr<Container> mean (const Container &xs)`

Same as above, but `xs` has pointers of variables.

**Return** A new variable.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- `xs`: Iterable container of pointers of variables. `xs` must have both `begin()` and `end()` functions that return the begin/end iterators.

*Tensor* `zeros_tensor (const Shape &shape, Device *dev)`

Creates a new *Tensor* with all values 0.

**Return** A new *Tensor*.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- `shape`: *Shape* of the new *Tensor*.
- `dev`: *Device* to manage the new *Tensor*, or `nullptr` to use the default device.

*Node* `zeros_node (const Shape &shape, Device *dev, Graph *g)`

Creates a new *Node* with all values 0.



**Return** A new *Node*.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- shape: *Shape* of the new *Node*.
- dev: *Device* to manage the new *Node*, or `nullptr` to use the default device.
- g: *Graph* to manage the instance of the *Node*, or `nullptr` to use the default graph.

```
template <typename Var>
```

```
type_traits::Identity<Var> zeros (const Shape &shape, Device *dev)
```

Creates a new variable with all values 0.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- shape: *Shape* of the new variable.
- dev: *Device* to manage the new variable, or `nullptr` to use the default device.

```
template <typename Var>
```

```
type_traits::Identity<Var> zeros (const Shape &shape, Device &dev)
```

Creates a new variable with all values 0.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- shape: *Shape* of the new variable.
- dev: *Device* to manage the new variable.

```
template <typename Var>
```

```
type_traits::Identity<Var> zeros (const Shape &shape)
```

Creates a new variable with all values 0.

**Return** A new variable.

**Remark** This function always uses the default device, and also uses the default graph when specifying *Node* as the template variable.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- shape: *Shape* of the new variable.

```
Tensor ones_tensor (const Shape &shape, Device *dev)
```

Creates a new *Tensor* with all values 1.

**Return** A new *Tensor*.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- shape: *Shape* of the new *Tensor*.

- dev: *Device* to manage the new *Tensor*, or `nullptr` to use the default device.

*Node* **ones\_node** (`const Shape &shape, Device *dev, Graph *g`)

Creates a new *Node* with all values 1.

**Return** A new *Node*.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- shape: *Shape* of the new *Node*.
- dev: *Device* to manage the new *Node*, or `nullptr` to use the default device.
- g: *Graph* to manage the instance of the *Node*, or `nullptr` to use the default graph.

**template** <typename Var>

type\_traits::Identity<Var> **ones** (`const Shape &shape, Device *dev`)

Creates a new variable with all values 1.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- shape: *Shape* of the new variable.
- dev: *Device* to manage the new variable, or `nullptr` to use the default device.

**template** <typename Var>

type\_traits::Identity<Var> **ones** (`const Shape &shape, Device &dev`)

Creates a new variable with all values 1.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- shape: *Shape* of the new variable.
- dev: *Device* to manage the new variable.

**template** <typename Var>

type\_traits::Identity<Var> **ones** (`const Shape &shape`)

Creates a new variable with all values 1.

**Return** A new variable.

**Remark** This function always uses the default device, and also uses the default graph when specifying *Node* as the template variable.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- shape: *Shape* of the new variable.

**template** <typename Var>

type\_traits::Identity<Var> **dropout** (`const Var &x, float rate, bool enabled`)

Applies the dropout:

$$\begin{aligned} w &\sim \text{Bernoulli}(w; 1 - r), \\ \text{dropout}(x) &:= \frac{1}{1-r} \times w \times x. \end{aligned}$$

**Return** A new variable.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- `x`: A variable representing original values.
- `rate`: The dropout probability  $r$ . 0 maintains all values and 1 discards all values.
- `enabled`: If `true`, this function applies the operation. Otherwise, this function performs nothing.

**namespace batch**

## Functions

**template** <typename Var>

type\_traits::Identity<Var> **sum** (const Var &`x`)

Applies summation along the minibatch. Following example shows how this function work:

$$x := \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}, \begin{pmatrix} 11 & 14 & 17 \\ 12 & 15 & 18 \\ 13 & 16 & 19 \end{pmatrix},$$

$$\text{batch} :: \text{sum}(x) = \begin{pmatrix} 12 & 18 & 24 \\ 14 & 20 & 26 \\ 16 & 22 & 28 \end{pmatrix}.$$

**Return** A new variable.

**Parameters**

- `x`: A variable representing values before reduction.

**template** <typename Var>

type\_traits::Identity<Var> **mean** (const Var &`x`)

Calculates means along the minibatch.

**Return** A new variable.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- `x`: A variable representing values before reduction.

**template** <typename Var>

type\_traits::Identity<Var> **normalize** (const Var &`x`)

Applies the batch normalization:

$$m_x := \frac{1}{B} \sum_{i=1}^B x_i,$$

$$v_x := \frac{B}{B-1} \left( \frac{1}{B} \sum_{i=0}^B x_i^2 - m_x^2 \right),$$

$$\text{batch} :: \text{normalize}(x) := \frac{x - m_x}{\sqrt{v_x + \epsilon}},$$

where  $B$  is the minibatch size of  $x$ .

**Return** A new variable.

**Remark** This function is implemented as a composite of some other functions.

**Parameters**

- `x`: A variable representing values before normalization.

**namespace random**

## Functions

*Tensor* **bernoulli\_tensor** (**const** *Shape* &*shape*, float *p*, *Device* \**dev*)

Creates a new *Tensor* with values sampled from the Bernoulli distribution.

**Return** A new *Tensor*.

**Parameters**

- *shape*: *Shape* of the new *Tensor*.
- *p*: The parameter *p* of the Bernoulli distribution.
- *dev*: *Device* to manage the new *Tensor*, or `nullptr` to use the default device.

*Node* **bernoulli\_node** (**const** *Shape* &*shape*, float *p*, *Device* \**dev*, *Graph* \**g*)

Creates a new *Node* with values sampled from the Bernoulli distribution.

**Return** A new *Node*.

**Parameters**

- *shape*: *Shape* of the new *Node*.
- *p*: The parameter *p* of the Bernoulli distribution.
- *dev*: *Device* to manage the new *Node*, or `nullptr` to use the default device.
- *g*: *Graph* to manage the instance of the *Node*, or `nullptr` to use the default graph.

**template** <typename *Var*>

`type_traits::Identity<Var>` **bernoulli** (**const** *Shape* &*shape*, float *p*, *Device* \**dev*)

Creates a new variable with values sampled from the Bernoulli distribution.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Parameters**

- *shape*: *Shape* of the new variable.
- *p*: The parameter *p* of the Bernoulli distribution.
- *dev*: *Device* to manage the new variable, or `nullptr` to use the default device.

**template** <typename *Var*>

`type_traits::Identity<Var>` **bernoulli** (**const** *Shape* &*shape*, float *p*, *Device* &*dev*)

Creates a new variable with values sampled from the Bernoulli distribution.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Parameters**

- *shape*: *Shape* of the new variable.
- *p*: The parameter *p* of the Bernoulli distribution.
- *dev*: *Device* to manage the new variable.

**template** <typename *Var*>

`type_traits::Identity<Var>` **bernoulli** (**const** *Shape* &*shape*, float *p*)

Creates a new variable with values sampled from the Bernoulli distribution.

**Return** A new variable.

**Remark** This function always uses the default device, and also uses the default graph when specifying *Node* as the template variable.

**Parameters**

- *shape*: *Shape* of the new variable.
- *p*: The parameter *p* of the Bernoulli distribution.

*Tensor* **uniform\_tensor** (**const** *Shape* &*shape*, float *lower*, float *upper*, *Device* \**dev*)

Creates a new *Tensor* with values sampled from the uniform distribution.

**Return** A new *Tensor*.

**Parameters**

- *shape*: *Shape* of the new *Tensor*.
- *lower*: The lower bound *L* of the uniform distribution.
- *upper*: The upper bound *U* of the uniform distribution.

- dev: *Device* to manage the new *Tensor*, or `nullptr` to use the default device.

*Node* **uniform\_node**(const *Shape* &shape, float lower, float upper, *Device* \*dev, *Graph* \*g)

Creates a new *Node* with values sampled from the uniform distribution.

**Return** A new *Node*.

**Parameters**

- shape: *Shape* of the new *Node*.
- lower: The lower bound  $L$  of the uniform distribution.
- upper: The upper bound  $U$  of the uniform distribution.
- dev: *Device* to manage the new *Node*, or `nullptr` to use the default device.
- g: *Graph* to manage the instance of the *Node*, or `nullptr` to use the default graph.

**template** <typename Var>

type\_traits::Identity<Var> **uniform**(const *Shape* &shape, float lower, float upper, *Device* \*dev)

Creates a new variable with values sampled from the uniform distribution.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Parameters**

- shape: *Shape* of the new variable.
- lower: The lower bound  $L$  of the uniform distribution.
- upper: The upper bound  $U$  of the uniform distribution.
- dev: *Device* to manage the new variable, or `nullptr` to use the default device.

**template** <typename Var>

type\_traits::Identity<Var> **uniform**(const *Shape* &shape, float lower, float upper, *Device* &dev)

Creates a new variable with values sampled from the uniform distribution.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Parameters**

- shape: *Shape* of the new variable.
- lower: The lower bound  $L$  of the uniform distribution.
- upper: The upper bound  $U$  of the uniform distribution.
- dev: *Device* to manage the new variable.

**template** <typename Var>

type\_traits::Identity<Var> **uniform**(const *Shape* &shape, float lower, float upper)

Creates a new variable with values sampled from the uniform distribution.

**Return** A new variable.

**Remark** This function always uses the default device, and also uses the default graph when specifying *Node* as the template variable.

**Parameters**

- shape: *Shape* of the new variable.
- lower: The lower bound  $L$  of the uniform distribution.
- upper: The upper bound  $U$  of the uniform distribution.

*Tensor* **normal\_tensor**(const *Shape* &shape, float mean, float sd, *Device* \*dev)

Creates a new *Tensor* with values sampled from the normal distribution.

**Return** A new *Tensor*.

**Parameters**

- shape: *Shape* of the new *Tensor*.
- mean: The mean  $\mu$  of the normal distribution.
- sd: The standard deviation  $\sigma$  of the normal distribution.
- dev: *Device* to manage the new *Tensor*, or `nullptr` to use the default device.

*Node* **normal\_node**(const *Shape* &shape, float mean, float sd, *Device* \*dev, *Graph* \*g)

Creates a new *Node* with values sampled from the normal distribution.

**Return** A new *Node*.

**Parameters**

- shape: *Shape* of the new *Node*.
- mean: The mean  $\mu$  of the normal distribution.
- sd: The standard deviation  $\sigma$  of the normal distribution.
- dev: *Device* to manage the new *Node*, or `nullptr` to use the default device.
- g: *Graph* to manage the instance of the *Node*, or `nullptr` to use the default graph.

**template** <typename Var>type\_traits::Identity<Var> **normal** (const *Shape* &shape, float mean, float sd, *Device* \*dev)

Creates a new variable with values sampled from the normal distribution.

**Return** A new variable.**Remark** This function uses the default graph when specifying *Node* as the template variable.**Parameters**

- shape: *Shape* of the new variable.
- mean: The mean  $\mu$  of the normal distribution.
- sd: The standard deviation  $\sigma$  of the normal distribution.
- dev: *Device* to manage the new variable, or `nullptr` to use the default device.

**template** <typename Var>type\_traits::Identity<Var> **normal** (const *Shape* &shape, float mean, float sd, *Device* &dev)

Creates a new variable with values sampled from the normal distribution.

**Return** A new variable.**Remark** This function uses the default graph when specifying *Node* as the template variable.**Parameters**

- shape: *Shape* of the new variable.
- mean: The mean  $\mu$  of the normal distribution.
- sd: The standard deviation  $\sigma$  of the normal distribution.
- dev: *Device* to manage the new variable.

**template** <typename Var>type\_traits::Identity<Var> **normal** (const *Shape* &shape, float mean, float sd)

Creates a new variable with values sampled from the normal distribution.

**Return** A new variable.**Remark** This function always uses the default device, and also uses the default graph when specifying *Node* as the template variable.**Parameters**

- shape: *Shape* of the new variable.
- mean: The mean  $\mu$  of the normal distribution.
- sd: The standard deviation  $\sigma$  of the normal distribution.

*Tensor* **log\_normal\_tensor** (const *Shape* &shape, float mean, float sd, *Device* \*dev)Creates a new *Tensor* with values sampled from the log-normal distribution.**Return** A new *Tensor*.**Parameters**

- shape: *Shape* of the new *Tensor*.
- mean: The parameter  $\mu$  of the log-normal distribution.
- sd: The parameter  $\sigma$  of the log-normal distribution.
- dev: *Device* to manage the new *Tensor*, or `nullptr` to use the default device.

*Node* **log\_normal\_node** (const *Shape* &shape, float mean, float sd, *Device* \*dev, *Graph* \*g)Creates a new *Node* with values sampled from the log-normal distribution.**Return** A new *Node*.**Parameters**

- shape: *Shape* of the new *Node*.
- mean: The parameter  $\mu$  of the log-normal distribution.
- sd: The parameter  $\sigma$  of the log-normal distribution.
- dev: *Device* to manage the new *Node*, or `nullptr` to use the default device.

- `g`: *Graph* to manage the instance of the *Node*, or `nullptr` to use the default graph.

**template** <typename Var>

`type_traits::Identity<Var> log_normal (const Shape &shape, float mean, float sd, Device &dev)`

Creates a new variable with values sampled from the log-normal distribution. Creates a new variable with values sampled from the log-normal distribution.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Parameters**

- `shape`: *Shape* of the new variable.
- `mean`: The parameter  $\mu$  of the log-normal distribution.
- `sd`: The parameter  $\sigma$  of the log-normal distribution.
- `dev`: *Device* to manage the new variable, or `nullptr` to use the default device.

**Parameters**

- `shape`: *Shape* of the new variable.
- `mean`: The parameter  $\mu$  of the log-normal distribution.
- `sd`: The parameter  $\sigma$  of the log-normal distribution.
- `dev`: *Device* to manage the new variable.

**template** <typename Var>

`type_traits::Identity<Var> log_normal (const Shape &shape, float mean, float sd)`

Creates a new variable with values sampled from the log-normal distribution.

**Return** A new variable.

**Remark** This function always uses the default device, and also uses the default graph when specifying *Node* as the template variable.

**Parameters**

- `shape`: *Shape* of the new variable.
- `mean`: The parameter  $\mu$  of the log-normal distribution.
- `sd`: The parameter  $\sigma$  of the log-normal distribution.

*Tensor* **gumbel\_tensor** (const Shape &shape, float mu, float beta, Device \*dev)

Creates a new *Tensor* with values sampled from the Gumbel distribution.

**Return** A new *Tensor*.

**Parameters**

- `shape`: *Shape* of the new *Tensor*.
- `mu`: The location parameter  $\mu$  of the Gumbel distribution.
- `beta`: The scale parameter  $\beta$  of the Gumbel distribution.
- `dev`: *Device* to manage the new *Tensor*, or `nullptr` to use the default device.

*Node* **gumbel\_node** (const Shape &shape, float mu, float beta, Device \*dev, Graph \*g)

Creates a new *Node* with values sampled from the Gumbel distribution.

**Return** A new *Node*.

**Parameters**

- `shape`: *Shape* of the new *Node*.
- `mu`: The location parameter  $\mu$  of the Gumbel distribution.
- `beta`: The scale parameter  $\beta$  of the Gumbel distribution.
- `dev`: *Device* to manage the new *Node*, or `nullptr` to use the default device.
- `g`: *Graph* to manage the instance of the *Node*, or `nullptr` to use the default graph.

**template** <typename Var>

`type_traits::Identity<Var> gumbel (const Shape &shape, float mu, float beta, Device *dev)`

Creates a new variable with values sampled from the Gumbel distribution.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Parameters**

- shape: *Shape* of the new variable.
- mu: The location parameter  $\mu$  of the Gumbel distribution.
- beta: The scale parameter  $\beta$  of the Gumbel distribution.
- dev: *Device* to manage the new variable, or `nullptr` to use the default device.

**template** <typename Var>

type\_traits::Identity<Var> **gumbel** (const *Shape* &shape, float mu, float beta, *Device* &dev)

Creates a new variable with values sampled from the Gumbel distribution.

**Return** A new variable.

**Remark** This function uses the default graph when specifying *Node* as the template variable.

**Parameters**

- shape: *Shape* of the new variable.
- mu: The location parameter  $\mu$  of the Gumbel distribution.
- beta: The scale parameter  $\beta$  of the Gumbel distribution.
- dev: *Device* to manage the new variable.

**template** <typename Var>

type\_traits::Identity<Var> **gumbel** (const *Shape* &shape, float mu, float beta)

Creates a new variable with values sampled from the Gumbel distribution.

**Return** A new variable.

**Remark** This function always uses the default device, and also uses the default graph when specifying *Node* as the template variable.

**Parameters**

- shape: *Shape* of the new variable.
- mu: The location parameter  $\mu$  of the Gumbel distribution.
- beta: The scale parameter  $\beta$  of the Gumbel distribution.

### 4.1.3 Graph

**class** primitiv::Graph

Computation graph.

Inherits from primitiv::mixins::DefaultSettable< Graph >, primitiv::mixins::Nonmovable< Graph >

#### Public Functions

void **clear** ()

Clear all operators in the graph.

**Remark** After calling this method, all *Node* objects supplied by the graph itself is invalidated.

*Node* **add\_operator** (std::unique\_ptr<Operator> &&op, const std::vector<*Node*> &args)

Adds a operator subgraph.

**Return** A new *Node* object of the resulting value.

**Parameters**

- op: Interface of the new operator.
- args: List of arguments. Each node should point a node in the same computation graph.

const *Tensor* &**forward** (const *Node* &node)

Calculates the value of given node.

**Return** Calculated value.



**Remark** This function calculates only the subgraph which is required to calculate the target node. Each intermediate result is stored to the corresponding node in the subgraph and they are re-used for future calculation. I.e., each node is calculated only once while the lifetime of the *Graph* object.

#### Parameters

- node: *Node* object specifying the target node.

void **backward**(const *Node* &node)  
Calculates the backpropagation.

**Remark** If node is not yet forwarded, this function implicitly calls `forward(node)`.

#### Parameters

- node: *Node* object specifying the output node.

*Shape* **get\_shape**(const *Node* &node) const  
Retrieves the shape of the node.

**Return** The shape of the node.

#### Parameters

- node: *Node* object specifying the target node.

*Device* &**get\_device**(const *Node* &node) const  
Retrieves the device of the node.

**Return** *Device* of the node.

#### Parameters

- node: *Node* object specifying the target node.

std::string **dump**(const std::string &format) const  
Dump internal graph structure.

**Return** A string that represents the internal graph using given format.

#### Parameters

- format: Name of the format. Available options: “dot” ... Graphviz’s dot format.

std::uint32\_t **num\_operators**() const  
Returns the number of operators in the computation graph.

**Return** Number of nodes.

## 4.1.4 Initializers

### Base Class

**class** `primitiv::Initializer`

Abstract class to provide parameter initialization algorithms.

Inherits from `primitiv::mixins::Nonmovable<Initializer>`

Subclassed by `primitiv::initializers::Constant`, `primitiv::initializers::Identity`, `primitiv::initializers::Normal`, `primitiv::initializers::Uniform`, `primitiv::initializers::XavierNormal`, `primitiv::initializers::XavierNormalConv2D`, `primitiv::initializers::XavierUniform`, `primitiv::initializers::XavierUniformConv2D`

## Public Functions

**virtual** void **apply** (*Tensor* &x) **const** = 0

Provides an initialized tensor.

### Parameters

- x: *Tensor* object to be initialized.

## Inherited Classes

**class** primitiv::initializers::**Constant**

*Initializer* to generate a same-value tensor.

Inherits from *primitiv::Initializer*

## Public Functions

**Constant** (float *k*)

Crates a new *Constant* initializer.

### Parameters

- k: Initial value of all variables in the parameter.

void **apply** (*Tensor* &x) **const**

Provides an initialized tensor.

### Parameters

- x: *Tensor* object to be initialized.

**class** primitiv::initializers::**Uniform**

*Initializer* using a parameterized uniform distribution with the range  $(L, U]$ .

Inherits from *primitiv::Initializer*

## Public Functions

**Uniform** (float *lower*, float *upper*)

Creates a new *Uniform* initializer.

### Parameters

- lower: Lower bound  $L$  of the uniform distribution.
- upper: Upper bound  $U$  of the uniform distribution.

void **apply** (*Tensor* &x) **const**

Provides an initialized tensor.

### Parameters

- x: *Tensor* object to be initialized.

**class** primitiv::initializers::**Normal**

*Initializer* using a parameterized normal distribution  $\mathcal{N}(\mu, \sigma)$ .

Inherits from *primitiv::Initializer*

## Public Functions

**Normal** (float *mean*, float *sd*)

Creates a new *Normal* initializer.

### Parameters

- *mean*: Mean  $\mu$  of the normal distribution.
- *sd*: Standard deviation  $\sigma$  of the normal distribution.

void **apply** (*Tensor* &*x*) **const**

Provides an initialized tensor.

### Parameters

- *x*: *Tensor* object to be initialized.

**class** `primitiv::initializers::Identity`

*Identity* matrix initializer.

Inherits from *primitiv::Initializer*

## Public Functions

**Identity** ()

Creates a new *Identity* initializer.

void **apply** (*Tensor* &*x*) **const**

Provides an initialized tensor.

### Parameters

- *x*: *Tensor* object to be initialized.

**class** `primitiv::initializers::XavierUniform`

The Xavier matrix initialization with the uniform distribution.

Inherits from *primitiv::Initializer*

## Public Functions

**XavierUniform** (float *scale* = 1.0f)

Creates a new *XavierUniform* initializer.

### Parameters

- *scale*: Additional scaling factor of the uniform distribution.

void **apply** (*Tensor* &*x*) **const**

Provides an initialized tensor.

### Parameters

- *x*: *Tensor* object to be initialized.

**class** `primitiv::initializers::XavierNormal`

The Xavier matrix initialization with the normal distribution.

Inherits from *primitiv::Initializer*

## Public Functions

**XavierNormal** (float *scale* = 1.0f)

Creates a new *XavierNormal* initializer.

### Parameters

- *scale*: Additional scaling factor of the normal distribution.

void **apply** (*Tensor* &*x*) **const**

Provides an initialized tensor.

### Parameters

- *x*: *Tensor* object to be initialized.

**class** `primitiv::initializers::XavierUniformConv2D`

The Xavier initialization with the uniform distribution for conv2d filters.

Inherits from *primitiv::Initializer*

## Public Functions

**XavierUniformConv2D** (float *scale* = 1.0f)

Creates a new *XavierUniformConv2D* initializer.

### Parameters

- *scale*: Additional scaling factor of the uniform distribution.

void **apply** (*Tensor* &*x*) **const**

Provides an initialized tensor.

### Parameters

- *x*: *Tensor* object to be initialized.

**class** `primitiv::initializers::XavierNormalConv2D`

The Xavier initialization with the normal distribution for conv2d filters.

Inherits from *primitiv::Initializer*

## Public Functions

**XavierNormalConv2D** (float *scale* = 1.0f)

Creates a new *XavierNormalConv2D* initializer.

### Parameters

- *scale*: Additional scaling factor of the normal distribution.

void **apply** (*Tensor* &*x*) **const**

Provides an initialized tensor.

### Parameters

- *x*: *Tensor* object to be initialized.

### 4.1.5 Model

**class** `primitiv::Model`

Set of parameters and specific algorithms.

Inherits from `primitiv::mixins::Nonmovable< Model >`

#### Public Functions

`void load (const std::string &path, bool with_stats, Device *device)`

Loads all parameters from a file.

##### Parameters

- `path`: Path of the file.
- `with_stats`: Whether or not to load all additional statistics.
- `device`: *Device* object to manage parameters.

`void load (const std::string &path, bool with_stats, Device &device)`

Loads all parameters from a file.

##### Parameters

- `path`: Path of the file.
- `with_stats`: Whether or not to load all additional statistics.
- `device`: *Device* object to manage parameters.

`void load (const std::string &path, bool with_stats)`

Loads all parameters from a file.

##### Parameters

- `path`: Path of the file.
- `with_stats`: Whether or not to load all additional statistics.

`void load (const std::string &path)`

Loads all parameters from a file.

##### Parameters

- `path`: Path of the file.

`void save (const std::string &path, bool with_stats) const`

Saves all parameters to a file.

##### Parameters

- `path`: Path of the file.
- `with_stats`: Whether or not to save all additional statistics.

`void save (const std::string &path) const`

Saves all parameters to a file.

##### Parameters

- `path`: Path of the file.

void **add** (**const** std::string &name, *Parameter* &param)

Registers a new parameter.

**Remark** name should not be overlapped with all registered parameters and submodels.

#### Parameters

- name: Name of the parameter.
- param: Reference to the parameter.

void **add** (**const** std::string &name, *Model* &model)

Registers a new submodel.

**Remark** name should not be overlapped with all registered parameters and submodels.

#### Parameters

- name: Name of the submodel.
- model: Reference to the submodel.

**const** *Parameter* &get\_parameter (**const** std::string &name) **const**

Retrieves a parameter with specified name.

**Return** Const-reference of the corresponding *Parameter* object.

#### Parameters

- name: Name of the parameter.

#### Exceptions

- primitiv::Error: *Parameter* with name not found.

*Parameter* &get\_parameter (**const** std::string &name)

Retrieves a parameter with specified name.

**Return** Reference of the corresponding *Parameter* object.

#### Parameters

- name: Name of the parameter.

#### Exceptions

- primitiv::Error: *Parameter* with name not found.

**const** *Parameter* &get\_parameter (**const** std::vector<std::string> &names) **const**

Recursively searches a parameter with specified name hierarchy.

**Return** Const-reference of the corresponding *Parameter* object.

#### Parameters

- names: Name hierarchy of the parameter.

#### Exceptions

- primitiv::Error: *Parameter* with names not found.

*Parameter* &get\_parameter (**const** std::vector<std::string> &names)

Recursively searches a parameter with specified name hierarchy.

**Return** Const-reference of the corresponding *Parameter* object.

#### Parameters

- names: Name hierarchy of the parameter.

#### Exceptions

- `primitiv::Error: Parameter` with names not found.

**const *Parameter* &get\_parameter (const std::initializer\_list<std::string> names) const**  
 Recursively searches a parameter with specified name hierarchy.

**Return** Const-reference of the corresponding *Parameter* object.

#### Parameters

- names: Name hierarchy of the parameter.

#### Exceptions

- `primitiv::Error: Parameter` with names not found.

***Parameter* &get\_parameter (const std::initializer\_list<std::string> names)**  
 Recursively searches a parameter with specified name hierarchy.

**Return** Const-reference of the corresponding *Parameter* object.

#### Parameters

- names: Name hierarchy of the parameter.

#### Exceptions

- `primitiv::Error: Parameter` with names not found.

**const *Model* &get\_submodel (const std::string &name) const**  
 Retrieves a submodel with specified name.

**Return** Const-reference of the corresponding *Model* object.

#### Parameters

- name: Name of the submodel.

#### Exceptions

- `primitiv::Error: Submodel` with name not found.

***Model* &get\_submodel (const std::string &name)**  
 Retrieves a submodel with specified name.

**Return** Reference of the corresponding *Model* object.

#### Parameters

- name: Name of the submodel.

#### Exceptions

- `primitiv::Error: Submodel` with name not found.

**const *Model* &get\_submodel (const std::vector<std::string> &names) const**  
 Recursively searches a submodel with specified name hierarchy.

**Return** Const-reference of the corresponding *Model* object.

#### Parameters

- names: Name hierarchy of the submodel.

#### Exceptions

- `primitiv::Error`: Submodel with names not found.

*Model* &`get_submodel (const std::vector<std::string> &names)`

Recursively searches a submodel with specified name hierarchy.

**Return** Const-reference of the corresponding *Model* object.

#### Parameters

- `names`: Name hierarchy of the submodel.

#### Exceptions

- `primitiv::Error`: Submodel with names not found.

`const Model &get_submodel (const std::initializer_list<std::string> names) const`

Recursively searches a submodel with specified name hierarchy.

**Return** Const-reference of the corresponding *Model* object.

#### Parameters

- `names`: Name hierarchy of the submodel.

#### Exceptions

- `primitiv::Error`: Submodel with names not found.

*Model* &`get_submodel (const std::initializer_list<std::string> names)`

Recursively searches a submodel with specified name hierarchy.

**Return** Const-reference of the corresponding *Model* object.

#### Parameters

- `names`: Name hierarchy of the submodel.

#### Exceptions

- `primitiv::Error`: Submodel with names not found.

`std::map<std::vector<std::string>, Parameter*> get_all_parameters () const`

Retrieves all parameters in the model.

**Return** Dictionary of parameters.

`std::map<std::vector<std::string>, Parameter*> get_trainable_parameters () const`

Retrieves trainable parameters in the model.

**Return** Dictionary of parameters.

## 4.1.6 Node

`class primitiv::Node`

Pointer of a node in the computation graph.

### Public Functions

`bool valid () const`

Returns whether the node is valid or not.

**Return** true or false w.r.t. the node is valid or not.



*Graph* &graph() const

Returns corresponding *Graph* object.

**Return** *Graph* object.

std::uint32\_t operator\_id() const

Returns the operator ID.

**Return** Operator ID.

std::uint32\_t value\_id() const

Returns the value ID of the operator.

**Return** Value ID.

*Shape* shape() const

Returns shape of the node.

**Return** A *Shape* object.

*Device* &device() const

Returns device of the node.

**Return** *Device* object.

float to\_float() const

Calculates the value of this node and returns a float.

**Return** A calculated float value.

**Remark** This function calls *Graph::forward()* internally. This function can be used only when the *Node* has a scalar and non-minibatched shape (i.e., *shape()* == *Shape()*)

std::vector<float> to\_vector() const

Calculates the value of this node and returns a list of float.

**Return** A list of calculated values.

**Remark** This function calls *Graph::forward()* internally.

std::vector<std::uint32\_t> argmax(std::uint32\_t dim) const

Returns argmax indices along an axis of this node.

**Return** A list of integers that indicates positions of the maximum values.

**Parameters**

- dim: A specified axis.

std::vector<std::uint32\_t> argmin(std::uint32\_t dim) const

Returns argmin indices along an axis of this node.

**Return** A list of integers that indicates positions of the minimum values.

**Parameters**

- dim: A specified axis.

void backward() const

Executes the backward operation from this node.

## 4.1.7 Optimizers

### Base Class

**class** `primitiv::Optimizer`

Abstract class for parameter optimizers.

Inherits from `primitiv::mixins::Nonmovable< Optimizer >`

Subclassed by *`primitiv::optimizers::AdaDelta`*, *`primitiv::optimizers::AdaGrad`*, *`primitiv::optimizers::Adam`*, *`primitiv::optimizers::MomentumSGD`*, *`primitiv::optimizers::RMSPProp`*, *`primitiv::optimizers::SGD`*

### Public Functions

void **load** (**const** `std::string &path`)

Loads configurations from a file.

#### Parameters

- `path`: Path of the optimizer parameter file.

void **save** (**const** `std::string &path`) **const**

Saves current configurations to a file.

#### Parameters

- `path`: Path of the file that will store optimizer parameters.

`std::uint32_t` **get\_epoch** () **const**

Retrieves current epoch.

**Return** Current epoch.

void **set\_epoch** (`std::uint32_t epoch`)

Sets current epoch.

#### Parameters

- `epoch`: New epoch.

float **get\_learning\_rate\_scaling** () **const**

Retrieves current learning rate scaling factor.

**Return** The scaling factor.

void **set\_learning\_rate\_scaling** (float *scale*)

Sets learning rate scaling factor.

**Remark** Could not set negative values.

#### Parameters

- `scale`: New scaling factor.

float **get\_weight\_decay** () **const**

Retrieves current L2 decay strength.

**Return** Current L2 decay strength.

void **set\_weight\_decay** (float *strength*)

Sets L2 decay strength.

**Remark** Could not set negative values.

**Parameters**

- strength: New L2 decay strength, or 0 to disable L2 decay.

float **get\_gradient\_clipping**() **const**  
Retrieves current gradient clipping threshold.

**Return** Current gradient clipping threshold.

void **set\_gradient\_clipping**(float *threshold*)  
Sets gradient clipping threshold.

**Remark** Could not set negative values.

**Parameters**

- threshold: New clipping threshold, or 0 to disable gradient clipping.

void **add**()  
Do nothing. This function is used as the sentinel of other specialized functions.

**template** <typename T, typename... Args>

void **add**(T &*model\_or\_param*, Args&... *args*)  
Registers multiple parameters and models. This function behaves similar to multiple *add()* calls with the same order of arguments. E.g., below lines should behave similarly (except the case of exceptions):

```
add(a, b, c, d);
add(a, b); add(c, d);
add(a); add(b); add(c); add(d);
```

**Parameters**

- model\_or\_param: *Parameter* or *Model* to be optimized.
- args: List of remaining *Parameter* or *Model* to be optimized.

void **reset\_gradients**()  
Resets all gradients of registered parameters.

void **update**()  
Updates parameter values.

**virtual** void **get\_configs**(std::unordered\_map<std::string, std::uint32\_t> &*uint\_configs*,  
std::unordered\_map<std::string, float> &*float\_configs*) **const**  
Gathers configuration values.

**Parameters**

- uint\_configs: Configurations with std::uint32\_t type.
- float\_configs: Configurations with float type.

**virtual** void **set\_configs**(**const** std::unordered\_map<std::string, std::uint32\_t> &*uint\_configs*,  
**const** std::unordered\_map<std::string, float> &*float\_configs*)  
Sets configuration values.

**Parameters**

- uint\_configs: Configurations with std::uint32\_t type.
- float\_configs: Configurations with float type.

## Inherited Classes

**class** `primitiv::optimizers::SGD`  
Simple stochastic gradient descent.  
Inherits from *primitiv::Optimizer*

### Public Functions

**SGD** (float *eta* = 0.1)  
Creates a new *SGD* object.

#### Parameters

- *eta*: Learning rate.

float **eta** () **const**  
Returns the learning rate.

**Return** Learning rate.

**class** `primitiv::optimizers::MomentumSGD`  
Stochastic gradient descent with momentum.  
Inherits from *primitiv::Optimizer*

### Public Functions

**MomentumSGD** (float *eta* = 0.01, float *momentum* = 0.9)  
Creates a new *MomentumSGD* object.

#### Parameters

- *eta*: Learning rate.
- *momentum*: Decay factor of the momentum.

float **eta** () **const**  
Returns the hyperparameter *eta*.

**Return** The value of *eta*.

float **momentum** () **const**  
Returns the hyperparameter *momentum*.

**Return** The value of *momentum*.

**class** `primitiv::optimizers::AdaGrad`  
*AdaGrad* optimizer.  
Inherits from *primitiv::Optimizer*

### Public Functions

**AdaGrad** (float *eta* = 0.001, float *eps* = 1e-8)  
Creates a new *AdaGrad* object.

#### Parameters

- eta: Learning rate.
- eps: Bias of power.

float **eta** () **const**  
Returns the hyperparameter eta.

**Return** The value of eta.

float **eps** () **const**  
Returns the hyperparameter eps.

**Return** The value of eps.

**class** `primitiv::optimizers::RMSProp`  
*RMSProp Optimizer.*

Inherits from *primitiv::Optimizer*

## Public Functions

**RMSProp** (float *eta* = 0.01, float *alpha* = 0.9, float *eps* = 1e-8)  
Creates a new *RMSProp* object.

### Parameters

- eta: Learning rate.
- alpha: Decay factor of moment.
- eps: Bias of power.

float **eta** () **const**  
Returns the hyperparameter eta.

**Return** The value of eta.

float **alpha** () **const**  
Returns the hyperparameter alpha.

**Return** The value of alpha.

float **eps** () **const**  
Returns the hyperparameter eps.

**Return** The value of eps.

**class** `primitiv::optimizers::AdaDelta`  
*AdaDelta optimizer. <https://arxiv.org/abs/1212.5701>*

Inherits from *primitiv::Optimizer*

## Public Functions

**AdaDelta** (float *rho* = 0.95, float *eps* = 1e-6)  
Creates a new *AdaDelta* object.

### Parameters

- rho: Decay factor of RMS operation.

- `eps`: Bias of RMS values.

float **rho** () **const**  
Returns the hyperparameter rho.

**Return** The value of rho.

float **eps** () **const**  
Returns the hyperparameter eps.

**Return** The value of eps.

**class** `primitiv::optimizers::Adam`  
*Adam* optimizer. <https://arxiv.org/abs/1412.6980>  
Inherits from *primitiv::Optimizer*

## Public Functions

**Adam** (float *alpha* = 0.001, float *beta1* = 0.9, float *beta2* = 0.999, float *eps* = 1e-8)  
Creates a new *Adam* object.

### Parameters

- *alpha*: Learning rate.
- *beta1*: Decay factor of momentum history.
- *beta2*: Decay factor of power history.
- *eps*: Bias of power.

float **alpha** () **const**  
Returns the hyperparameter alpha.

**Return** The value of alpha.

float **beta1** () **const**  
Returns the hyperparameter beta1.

**Return** The value of beta1.

float **beta2** () **const**  
Returns the hyperparameter beta2.

**Return** The value of beta2.

float **eps** () **const**  
Returns the hyperparameter eps.

**Return** The value of eps.

## 4.1.8 Parameter

**class** `primitiv::Parameter`  
Class to manage a trainable tensor parameter.  
Inherits from `primitiv::mixins::Nonmovable< Parameter >`

## Public Functions

### **Parameter** ()

Creates an invalid parameter object.

### **Parameter** (const *Shape* &shape, const std::vector<float> &value, *Device* \*device)

Creates a new *Parameter* object.

#### Parameters

- shape: The shape of the parameter. The batch size should be 1.
- value: List of initial values. Order of elements should be the column-major (Fortran) order.
- device: The device object to manage internal memory.

### **Parameter** (const *Shape* &shape, const std::vector<float> &value, *Device* &device)

Creates a new *Parameter* object.

#### Parameters

- shape: The shape of the parameter. The batch size should be 1.
- value: List of initial values. Order of elements should be the column-major (Fortran) order.
- device: The device object to manage internal memory.

### **Parameter** (const *Shape* &shape, const std::vector<float> &value)

Creates a new *Parameter* object.

#### Parameters

- shape: The shape of the parameter. The batch size should be 1.
- value: List of initial values. Order of elements should be the column-major (Fortran) order.

### **Parameter** (const *Shape* &shape, const *Initializer* &initializer, *Device* \*device)

Creates a new *Parameter* object.

#### Parameters

- shape: The shape of the parameter. The batch size should be 1.
- initializer: An *Initializer* object.
- device: The device object to manage internal memory.

### **Parameter** (const *Shape* &shape, const *Initializer* &initializer, *Device* &device)

Creates a new *Parameter* object.

#### Parameters

- shape: The shape of the parameter. The batch size should be 1.
- initializer: An *Initializer* object.
- device: The device object to manage internal memory.

### **Parameter** (const *Shape* &shape, const *Initializer* &initializer)

Creates a new *Parameter* object.

#### Parameters

- shape: The shape of the parameter. The batch size should be 1.
- initializer: An *Initializer* object.

void **init** (**const** *Shape* &shape, **const** std::vector<float> &value, *Device* \*device)  
 Initializes the *Parameter* object.

**Parameters**

- shape: The shape of the parameter. The batch size should be 1.
- value: List of initial values. Order of elements should be the column-major (Fortran) order.
- device: The device object to manage internal memory.

void **init** (**const** *Shape* &shape, **const** std::vector<float> &value, *Device* &device)  
 Initializes the *Parameter* object.

**Parameters**

- shape: The shape of the parameter. The batch size should be 1.
- value: List of initial values. Order of elements should be the column-major (Fortran) order.
- device: The device object to manage internal memory.

void **init** (**const** *Shape* &shape, **const** std::vector<float> &value)  
 Initializes the *Parameter* object.

**Parameters**

- shape: The shape of the parameter. The batch size should be 1.
- value: List of initial values. Order of elements should be the column-major (Fortran) order.

void **init** (**const** *Shape* &shape, **const** *Initializer* &initializer, *Device* \*device)  
 Initializes the *Parameter* object.

**Parameters**

- shape: The shape of the parameter. The batch size should be 1.
- initializer: An *Initializer* object.
- device: The device object to manage internal memory.

void **init** (**const** *Shape* &shape, **const** *Initializer* &initializer, *Device* &device)  
 Initializes the *Parameter* object.

**Parameters**

- shape: The shape of the parameter. The batch size should be 1.
- initializer: An *Initializer* object.
- device: The device object to manage internal memory.

void **init** (**const** *Shape* &shape, **const** *Initializer* &initializer)  
 Initializes the *Parameter* object.

**Parameters**

- shape: The shape of the parameter. The batch size should be 1.
- initializer: An *Initializer* object.

void **load** (**const** std::string &path, bool with\_stats, *Device* \*device)  
 Loads parameters from specified file.

**Parameters**

- path: File path to load parameters.



- `with_stats`: Whether or not to load all additional statistics as well as parameter values if the file has them.
- `device`: The device object to manage internal memory.

void **load** (**const** std::string &*path*, bool *with\_stats*, *Device* &*device*)  
 Loads parameters from specified file.

#### Parameters

- `path`: File path to load parameters.
- `with_stats`: Whether or not to load all additional statistics as well as parameter values if the file has them.
- `device`: The device object to manage internal memory.

void **load** (**const** std::string &*path*, bool *with\_stats*)  
 Loads parameters from specified file.

#### Parameters

- `path`: File path to load parameters.
- `with_stats`: Whether or not to load all additional statistics as well as parameter values if the file has them.

void **load** (**const** std::string &*path*)  
 Loads parameters from specified file.

#### Parameters

- `path`: File path to load parameters.

void **save** (**const** std::string &*path*, bool *with\_stats*) **const**  
 Saves current parameters into specified file.

#### Parameters

- `path`: File path to save parameters.
- `with_stats`: Whether or not to save all additional statistics as well as parameter values if the parameter object has them.

void **save** (**const** std::string &*path*) **const**  
 Saves current parameters into specified file.

#### Parameters

- `path`: File path to save parameters.

bool **valid** () **const**  
 Returns whether the parameter is valid or not.

**Return** true or false w.r.t. the parameter is valid or not.

void **reset\_gradient** ()  
 Set all gradients to 0.

void **add\_stats** (**const** std::string &*name*, **const** *Shape* &*shape*)  
 Adds a new optional statistics tensor.

**Remark** All elements in the new statistics tensor is initialized by 0.

#### Parameters

- `name`: Name of the statistics.
- `shape`: *Shape* of the tensor.

**bool** `has_stats (const std::string &name) const`  
 Checks whether the statistics with name `name` exists or not.

**Return** true if the entry exists, false otherwise.

**Parameters**

- `name`: Name of the statistics.

*Shape* **shape () const**  
 Returns the shape of the parameter.

**Return** *Shape* object.

*Device* **&device () const**  
 Returns the *Device* object to manage the internal memory.

**Return** Pointer of the *Device* object.

**const** *Tensor* **&value () const**  
 Returns the values of the parameter.  
**Return** A tensor representing the parameter tensor.

*Tensor* **&value ()**  
 Returns the values of the parameter.  
**Return** A tensor representing the parameter tensor.

**const** *Tensor* **&gradient () const**  
 Returns the current gradient of the parameter.  
**Return** A tensor representing the gradient of the value.

*Tensor* **&gradient ()**  
 Returns the current gradient of the parameter.  
**Return** A tensor representing the gradient of the value.

**const** *Tensor* **&stats (const std::string &name) const**  
 Returns the current optional statistics tensor specified by given name.  
**Return** A tensor.

**Parameters**

- `name`: Name of the statistics.

*Tensor* **&stats (const std::string &name)**  
 Returns the current optional statistics tensor specified by given name.

**Return** A tensor.

**Parameters**

- `name`: Name of the statistics.

### 4.1.9 Shape

**class** `primitiv::Shape`

Data structure to represent the shape of the node.

Examples: `Shape() == Shape({1, 1, 1, ...}, 1)`: scalar `Shape({}) == Shape({1, 1, 1, ...}, 1)`: scalar `Shape({n}) == Shape({n, 1, 1, ...}, 1)`: column vector `Shape({n, m}) == Shape({n, m, 1, ...}, 1)`: matrix `Shape({...}, k)`: k-parallelized data (mini-batch)

#### Public Functions

**Shape** ()

Creates a new scalar *Shape* object.

**Shape** (std::initializer\_list<std::uint32\_t> *dims*, std::uint32\_t *batch* = 1)

Creates a new *Shape* object.

##### Parameters

- *dims*: List of the dimension sizes.
- *batch*: Batch size.

**Shape** (const std::vector<std::uint32\_t> &*dims*, std::uint32\_t *batch* = 1)

Creates a new *Shape* object.

##### Parameters

- *dims*: List of the dimension sizes.
- *batch*: Batch size.

std::uint32\_t **operator** [] (std::uint32\_t *i*) const

Returns the size of the i-th dimension.

**Return** Size of the i-th dimension.

##### Parameters

- *i*: Dimension number to check.

const std::vector<std::uint32\_t> **dims** () const

Returns the dimension array.

**Return** Copy of the dimension array.

std::uint32\_t **depth** () const

Returns the depth (length of non-1 dimensions) of the shape.

**Return** The depth of the shape.

std::uint32\_t **batch** () const

Returns the batch size.

**Return** Batch size.

std::uint32\_t **volume** () const

Returns the number of elements in each sample. This value is equal to the product of all dimensions.

**Return** Number of elements.

`std::uint32_t lower_volume (std::uint32_t dim) const`

Returns the number of elements in 1 to specified dim.

**Return** `dims[0] * dims[1] * ... * dims[dim-1]`

**Parameters**

- `dim`: Upper bound of the dimension.

`std::uint32_t size () const`

Returns the number of elements in all samples of the mini-batch. This value is equal to `batch () * volume ()`.

**Return** Number of elements.

`std::string to_string () const`

Returns a string representation of the shape. The format is: “[n,m,...]xk”

**Return** Encoded string.

`bool operator== (const Shape &rhs) const`

Compares this and other shape.

**Return** true if this and rhs are same, false otherwise.

**Parameters**

- `rhs`: *Shape* object to compare.

`bool operator!= (const Shape &rhs) const`

Compares this and other shape.

**Return** true if this and rhs are not same, false otherwise.

**Parameters**

- `rhs`: *Shape* object to compare.

`bool has_batch () const`

Checks whether the shape has minibatch or not.

**Return** true if the shape has minibatch, false otherwise.

`bool has_compatible_batch (const Shape &rhs) const`

Checks whether two batch size is compatible (broadcastable) or not.

**Return** true if both batch size is compatible, false otherwise.

**Parameters**

- `rhs`: *Shape* object to compare.

`bool is_scalar () const`

Checks whether the shape is a scalar or not.

**Return** true if the shape is a scalar, false otherwise.

`bool is_column_vector () const`

Checks whether the shape is a column vector or not.

**Return** true if the shape is a column vector, false otherwise.

`bool is_matrix () const`

Checks whether the shape is a vector or a matrix, or not.

**Return** true if the shape is a vector or a matrix, false otherwise.

bool **has\_same\_dims** (const *Shape* &rhs) const

Checks whether two shapes have completely same dimensions.

**Return** true if both shape have same dimensions, false otherwise.

**Parameters**

- rhs: *Shape* object to compare.

bool **has\_same\_loo\_dims** (const *Shape* &rhs, std::uint32\_t dim) const

Checks whether two shapes have same dimensions without an axis. (LOO: leave one out)

**Return** true if both shape have same dimensions regardless the dimension dim, false otherwise.

**Parameters**

- rhs: *Shape* object to compare.
- dim: Dimension to be ignored.

*Shape* **resize\_dim** (std::uint32\_t dim, std::uint32\_t m) const

Creates a new shape which have one different dimension.

**Return** New shape.

**Parameters**

- dim: Dimension to be changed.
- m: New size of the dimension dim.

*Shape* **resize\_batch** (std::uint32\_t batch) const

Creates a new shape which have specified batch size.

**Return** New shape.

**Parameters**

- batch: New batch size.

void **update\_dim** (std::uint32\_t dim, std::uint32\_t m)

Directly updates a specified dimension.

**Parameters**

- dim: Dimension to be updated.
- m: New size of the dimension dim.

void **update\_batch** (std::uint32\_t batch)

Directly updates the batch size.

**Parameters**

- batch: New batch size.

## 4.1.10 Tensor

class primitiv::Tensor

Value with any dimensions.

## Public Functions

### **Tensor()**

Creates an invalid *Tensor*.

### bool **valid()** **const**

Check whether the object is valid or not.

**Return** true if the object is valid, false otherwise.

**Remark** This returns false when the object is created through the default constructor or the object had been moved.

### void **check\_valid()** **const**

Check whether the object is valid or not.

#### **Exceptions**

- `primitiv::Error`: This object is invalid.

### *Shape* **shape()** **const**

Returns the shape of the *Tensor*.

**Return** *Shape* of the *Tensor*.

### *Device* & **device()** **const**

Returns the *Device* object related to the internal memory.

**Return** *Device* object.

### float **to\_float()** **const**

Retrieves one internal value in the tensor.

**Return** An internal float value.

**Remark** This function can be used only when the tensor is a scalar and non-minibatched (i.e., *shape()* == *Shape()*).

### std::vector<float> **to\_vector()** **const**

Retrieves internal values in the tensor as a vector.

**Return** A list of the internal values.

**Remark** Each resulting values are ordered by the column-major order, and the batch size is assumed as the last dimension of the tensor.

### std::vector<std::uint32\_t> **argmax**(std::uint32\_t *dim*) **const**

Retrieves argmax indices along an axis.

**Return** A list of integers that indicates positions of the maximum values.

#### **Parameters**

- *dim*: A specified axis.

### std::vector<std::uint32\_t> **argmin**(std::uint32\_t *dim*) **const**

Retrieves argmin indices along an axis.

**Return** A list of integers that indicates positions of the minimum values.

#### **Parameters**

- *dim*: A specified axis.

void **invalidate** ()  
Invalidates this object.

void **reset** (float *k*)  
Reset internal values using a constant.

**Parameters**

- *k*: A value to be used to initialize each element.

void **reset\_by\_array** (const float \**values*)  
Reset internal values using a vector.

**Remark** Length of *values* should be equal to `shape().size()`. Each element should be ordered by the column-major order, and the batch size is assumed as the last dimension.

**Parameters**

- *values*: Array of values to be used to initialize each element.

void **reset\_by\_vector** (const std::vector<float> &*values*)  
Reset internal values using a vector.

**Remark** *values.size()* should be equal to `shape().size()`. Each element should be ordered by the column-major order, and the batch size is assumed as the last dimension.

**Parameters**

- *values*: List of values to be used to initialize each element.

*Tensor* **reshape** (const *Shape* &*new\_shape*) const  
Returns a tensor which have the same values and different shape.

**Return** A new tensor.

**Parameters**

- *new\_shape*: New shape with batch size 1.

*Tensor* **flatten** () const  
Returns a flattened tensor.

**Return** A new tensor.

*Tensor* &**inplace\_multiply\_const** (float *k*)  
Directly multiplies a constant.

**Return** \*this

**Parameters**

- *k*: A constant to multiply.

*Tensor* &**inplace\_add** (const *Tensor* &*x*)  
Directly adds a value.

**Return** \*this

**Parameters**

- *x*: A tensor to add.

*Tensor* &**inplace\_subtract** (const *Tensor* &*x*)  
Directly subtracts a value.

**Return** \*this

### Parameters

- `x`: A tensor to subtract.

## 4.2 Build Options

### 4.2.1 Standard Options

Users basically can use *CMake 3.1.0* standard options (e.g., `-DCMAKE_INSTALL_PREFIX`) together with the unique options.

### 4.2.2 Unique Options

**PRIMITIV\_BUILD\_C\_API** Default value: `OFF`

Builds C APIs. `libprimitiv_c` library file and headers in the `primitiv/c` directory will also be installed.

**PRIMITIV\_BUILD\_STATIC\_LIBRARY** Default value: `OFF`

Builds static libraries instead of shared objects.

**PRIMITIV\_BUILD\_TESTS** Default value: `OFF`

Builds test binaries and generates `make test` command. This option introduces a dependency to the [Google Test](#). `FindGTest` options can also be used.

**PRIMITIV\_BUILD\_TESTS\_PROBABILISTIC** Default value: `OFF`

Builds test cases that probabilistically fails.

**PRIMITIV\_GTEST\_SOURCE\_DIR** Default value: `" "`

Specifies the source directory of Google Test. If you want to use Google Test provided from Debian/Ubuntu repository, add `-DPRIMITIV_GTEST_SOURCE_DIR=/usr/src/googletest/googletest` together with `-PRIMITIV_BUILD_TESTS=ON` option.

**PRIMITIV\_USE\_CACHE** Default value: `OFF`

Whether or not to use cached values to prevent increasing computation amount. Libraries built with this flag will tend to consume more memory.

**PRIMITIV\_USE\_EIGEN** Default value: `OFF`

Enables Eigen backend (`primitiv::devices::Eigen` class). This option introduces a dependency to the [Eigen3](#) library, and `FindEigen3` options can also be used.

**PRIMITIV\_USE\_CUDA** Default value: `OFF`

Enables CUDA backend (`primitiv::devices::CUDA` class). This option introduces a dependency to the [NVIDIA CUDA Toolkit v8.0](#) or later and [cuDNN library v5.0](#) or later. `FindCuda` and `FindCuDNN` options can also be used.

**PRIMITIV\_USE\_OPENCL** Default value: `OFF`

Enables OpenCL backend (`primitiv::devices::OpenCL` class). This option introduces dependencies to an [OpenCL v1.2](#) implementation and [OpenCL C++ Bindings v2](#). `FindOpenCL` options can also be used, and `cl2.hpp` should be found in `/path/to/include/CL`.



## 4.3 primitiv File Format v0.1

**primitiv File Format** is a common binary format to store/load data used in primitiv. It uses the [MessagePack](#) wire format as the inner binary representation.

### 4.3.1 Legend

```
+-----+      +-----+-----+-----+...
| Type |  =  | Member Type 1 | Member Type 2 |
|       |    | Member Name 1 | Member Name 2 |
+-----+      +-----+-----+-----+...
```

### 4.3.2 Types

```
+-----+      +-----+-----+-----+
| Shape |  =  | array<uint32> | uint32 |
|       |    | dims          | batch  |
+-----+      +-----+-----+-----+
```

In the current version, the `batch` member is always 1 for all `Shape` objects.

```
+-----+      +-----+-----+
| Tensor | =  | Shape | bin  |
|       |    | shape | data |
+-----+      +-----+-----+
```

`data` member has an array of single-precision floating number with the following format:

- Byte order: *Little-endian* (differ than MessagePack's float)
- Array order: *Column-major* (Fortran)
- Batch is treated as the *last dimension* of the shape (if `shape.batch > 1`). I.e., The next data begins just after the previous data according to the *column-major array order*.

```
+-----+      +-----+-----+~~~~~+.....
| Parameter | = | Tensor | uint32 | str      | Tensor      |
|           |   | value  | N      | stat_key[1] | stat_value[1] | N times
+-----+      +-----+-----+~~~~~+.....
```

```
+-----+      +-----+~~~~~+.....
| Model |  =  | uint32 | array<str> | Parameter      |
|       |    | N      | param_key[1] | param_value[1] | N times
+-----+      +-----+~~~~~+.....
```

The key of each parameter represents the *address* of the parameter from the root model. E.g.:

- `param_key == ["foo"]`: Parameter has the name "foo", and is directly owned by the *root model*.
- `param_key == ["foo", "bar"]`: Parameter has the name "bar", and is owned by the submodel "foo".

```
+-----+      +-----+-----+-----+
| Optimizer | = | map<str, uint32> | map<str, float> |
```

		uint_configs	float_configs	
+-----+	+-----+	+-----+	+-----+	

### 4.3.3 File Format

uint32	uint32	uint32	Shape Tensor Parameter Model Optimizer
ver_major	ver_minor	data_type	data

Version numbers are typically equal to following:

- `ver_major == 0`
- `ver_minor == 1`

Following table shows the correspondence between `data_type` and `data`:

data_type	data
0x0	Shape
0x100	Tensor
0x200	Parameter
0x300	Model
0x400	Optimizer

## P

- primitiv::Device (C++ class), 23
- primitiv::Device::copy\_tensor (C++ function), 24
- primitiv::Device::CPU (C++ enumerator), 23
- primitiv::Device::CUDA (C++ enumerator), 23
- primitiv::Device::DeviceType (C++ type), 23
- primitiv::Device::dump\_description (C++ function), 24
- primitiv::Device::EIGEN (C++ enumerator), 23
- primitiv::Device::GROUP\_CPU (C++ enumerator), 23
- primitiv::Device::GROUP\_CUDA (C++ enumerator), 23
- primitiv::Device::GROUP\_FILTER (C++ enumerator), 23
- primitiv::Device::GROUP\_OPENCL (C++ enumerator), 24
- primitiv::Device::inplace\_add (C++ function), 25
- primitiv::Device::inplace\_multiply\_const (C++ function), 24
- primitiv::Device::inplace\_subtract (C++ function), 25
- primitiv::Device::NAIVE (C++ enumerator), 23
- primitiv::Device::new\_tensor\_by\_array (C++ function), 24
- primitiv::Device::new\_tensor\_by\_constant (C++ function), 24
- primitiv::Device::new\_tensor\_by\_vector (C++ function), 24
- primitiv::Device::OPENCL (C++ enumerator), 24
- primitiv::Device::type (C++ function), 24
- primitiv::devices::CUDA (C++ class), 26
- primitiv::devices::CUDA::assert\_support (C++ function), 26
- primitiv::devices::CUDA::check\_support (C++ function), 27
- primitiv::devices::CUDA::CUDA (C++ function), 26
- primitiv::devices::CUDA::dump\_description (C++ function), 26
- primitiv::devices::CUDA::num\_devices (C++ function), 26
- primitiv::devices::CUDA::type (C++ function), 26
- primitiv::devices::Eigen (C++ class), 25
- primitiv::devices::Eigen::dump\_description (C++ function), 26
- primitiv::devices::Eigen::Eigen (C++ function), 26
- primitiv::devices::Eigen::type (C++ function), 26
- primitiv::devices::Naive (C++ class), 25
- primitiv::devices::Naive::dump\_description (C++ function), 25
- primitiv::devices::Naive::Naive (C++ function), 25
- primitiv::devices::Naive::type (C++ function), 25
- primitiv::devices::OpenCL (C++ class), 27
- primitiv::devices::OpenCL::assert\_support (C++ function), 28
- primitiv::devices::OpenCL::check\_support (C++ function), 28
- primitiv::devices::OpenCL::dump\_description (C++ function), 27
- primitiv::devices::OpenCL::num\_devices (C++ function), 27
- primitiv::devices::OpenCL::num\_platforms (C++ function), 27
- primitiv::devices::OpenCL::OpenCL (C++ function), 27
- primitiv::devices::OpenCL::type (C++ function), 27
- primitiv::functions (C++ type), 28
- primitiv::functions::add (C++ function), 29
- primitiv::functions::batch (C++ type), 47
- primitiv::functions::batch::mean (C++ function), 47
- primitiv::functions::batch::normalize (C++ function), 47
- primitiv::functions::batch::sum (C++ function), 47
- primitiv::functions::broadcast (C++ function), 39
- primitiv::functions::concat (C++ function), 35
- primitiv::functions::constant (C++ function), 41, 42
- primitiv::functions::constant\_node (C++ function), 41
- primitiv::functions::constant\_tensor (C++ function), 41
- primitiv::functions::conv2d (C++ function), 40
- primitiv::functions::copy (C++ function), 33
- primitiv::functions::cos (C++ function), 37
- primitiv::functions::divide (C++ function), 30, 31
- primitiv::functions::dropout (C++ function), 46
- primitiv::functions::elu (C++ function), 38
- primitiv::functions::exp (C++ function), 36

primitiv::functions::flatten (C++ function), 36  
 primitiv::functions::identity (C++ function), 42, 43  
 primitiv::functions::identity\_node (C++ function), 42  
 primitiv::functions::identity\_tensor (C++ function), 42  
 primitiv::functions::input (C++ function), 32  
 primitiv::functions::input\_node (C++ function), 32  
 primitiv::functions::input\_tensor (C++ function), 31  
 primitiv::functions::log (C++ function), 36  
 primitiv::functions::log\_softmax (C++ function), 39  
 primitiv::functions::logsumexp (C++ function), 39  
 primitiv::functions::lrelu (C++ function), 38  
 primitiv::functions::matmul (C++ function), 36  
 primitiv::functions::max\_pool2d (C++ function), 41  
 primitiv::functions::mean (C++ function), 44  
 primitiv::functions::multiply (C++ function), 30  
 primitiv::functions::negative (C++ function), 29  
 primitiv::functions::ones (C++ function), 46  
 primitiv::functions::ones\_node (C++ function), 46  
 primitiv::functions::ones\_tensor (C++ function), 45  
 primitiv::functions::parameter (C++ function), 33  
 primitiv::functions::parameter\_node (C++ function), 33  
 primitiv::functions::parameter\_tensor (C++ function), 33  
 primitiv::functions::pick (C++ function), 34  
 primitiv::functions::positive (C++ function), 28  
 primitiv::functions::pow (C++ function), 31  
 primitiv::functions::pown (C++ function), 31  
 primitiv::functions::prelu (C++ function), 38  
 primitiv::functions::random (C++ type), 47  
 primitiv::functions::random::bernoulli (C++ function), 48  
 primitiv::functions::random::bernoulli\_node (C++ function), 48  
 primitiv::functions::random::bernoulli\_tensor (C++ function), 48  
 primitiv::functions::random::gumbel (C++ function), 51, 52  
 primitiv::functions::random::gumbel\_node (C++ function), 51  
 primitiv::functions::random::gumbel\_tensor (C++ function), 51  
 primitiv::functions::random::log\_normal (C++ function), 51  
 primitiv::functions::random::log\_normal\_node (C++ function), 50  
 primitiv::functions::random::log\_normal\_tensor (C++ function), 50  
 primitiv::functions::random::normal (C++ function), 50  
 primitiv::functions::random::normal\_node (C++ function), 49  
 primitiv::functions::random::normal\_tensor (C++ function), 49  
 primitiv::functions::random::uniform (C++ function), 49  
 primitiv::functions::random::uniform\_node (C++ function), 49  
 primitiv::functions::random::uniform\_tensor (C++ function), 48  
 primitiv::functions::relu (C++ function), 37  
 primitiv::functions::reshape (C++ function), 35  
 primitiv::functions::selu (C++ function), 43  
 primitiv::functions::sigmoid (C++ function), 37  
 primitiv::functions::sin (C++ function), 37  
 primitiv::functions::slice (C++ function), 34  
 primitiv::functions::softmax (C++ function), 39  
 primitiv::functions::softmax\_cross\_entropy (C++ function), 40  
 primitiv::functions::softplus (C++ function), 37  
 primitiv::functions::sqrt (C++ function), 36  
 primitiv::functions::stop\_gradient (C++ function), 40  
 primitiv::functions::subtract (C++ function), 29, 30  
 primitiv::functions::sum (C++ function), 38, 43  
 primitiv::functions::tan (C++ function), 37  
 primitiv::functions::tanh (C++ function), 37  
 primitiv::functions::transpose (C++ function), 36  
 primitiv::functions::zeros (C++ function), 45  
 primitiv::functions::zeros\_node (C++ function), 44  
 primitiv::functions::zeros\_tensor (C++ function), 44  
 primitiv::Graph (C++ class), 52  
 primitiv::Graph::add\_operator (C++ function), 52  
 primitiv::Graph::backward (C++ function), 53  
 primitiv::Graph::clear (C++ function), 52  
 primitiv::Graph::dump (C++ function), 53  
 primitiv::Graph::forward (C++ function), 52  
 primitiv::Graph::get\_device (C++ function), 53  
 primitiv::Graph::get\_shape (C++ function), 53  
 primitiv::Graph::num\_operators (C++ function), 53  
 primitiv::Initializer (C++ class), 53  
 primitiv::Initializer::apply (C++ function), 54  
 primitiv::initializers::Constant (C++ class), 54  
 primitiv::initializers::Constant::apply (C++ function), 54  
 primitiv::initializers::Constant::Constant (C++ function), 54  
 primitiv::initializers::Identity (C++ class), 55  
 primitiv::initializers::Identity::apply (C++ function), 55  
 primitiv::initializers::Identity::Identity (C++ function), 55  
 primitiv::initializers::Normal (C++ class), 54  
 primitiv::initializers::Normal::apply (C++ function), 55  
 primitiv::initializers::Normal::Normal (C++ function), 55  
 primitiv::initializers::Uniform (C++ class), 54  
 primitiv::initializers::Uniform::apply (C++ function), 54  
 primitiv::initializers::Uniform::Uniform (C++ function), 54  
 primitiv::initializers::XavierNormal (C++ class), 55  
 primitiv::initializers::XavierNormal::apply (C++ function), 56  
 primitiv::initializers::XavierNormal::XavierNormal (C++ function), 56  
 primitiv::initializers::XavierNormalConv2D (C++ class), 56

primitiv::initializers::XavierNormalConv2D::apply (C++ function), 56

primitiv::initializers::XavierNormalConv2D::XavierNormalConv2D (C++ function), 56

primitiv::initializers::XavierUniform (C++ class), 55

primitiv::initializers::XavierUniform::apply (C++ function), 55

primitiv::initializers::XavierUniform::XavierUniform (C++ function), 55

primitiv::initializers::XavierUniformConv2D (C++ class), 56

primitiv::initializers::XavierUniformConv2D::apply (C++ function), 56

primitiv::initializers::XavierUniformConv2D::XavierUniformConv2D (C++ function), 56

primitiv::Model (C++ class), 57

primitiv::Model::add (C++ function), 57, 58

primitiv::Model::get\_all\_parameters (C++ function), 60

primitiv::Model::get\_parameter (C++ function), 58, 59

primitiv::Model::get\_submodel (C++ function), 59, 60

primitiv::Model::get\_trainable\_parameters (C++ function), 60

primitiv::Model::load (C++ function), 57

primitiv::Model::save (C++ function), 57

primitiv::Node (C++ class), 60

primitiv::Node::argmax (C++ function), 61

primitiv::Node::argmin (C++ function), 61

primitiv::Node::backward (C++ function), 61

primitiv::Node::device (C++ function), 61

primitiv::Node::graph (C++ function), 60

primitiv::Node::operator\_id (C++ function), 61

primitiv::Node::shape (C++ function), 61

primitiv::Node::to\_float (C++ function), 61

primitiv::Node::to\_vector (C++ function), 61

primitiv::Node::valid (C++ function), 60

primitiv::Node::value\_id (C++ function), 61

primitiv::Optimizer (C++ class), 62

primitiv::Optimizer::add (C++ function), 63

primitiv::Optimizer::get\_configs (C++ function), 63

primitiv::Optimizer::get\_epoch (C++ function), 62

primitiv::Optimizer::get\_gradient\_clipping (C++ function), 63

primitiv::Optimizer::get\_learning\_rate\_scaling (C++ function), 62

primitiv::Optimizer::get\_weight\_decay (C++ function), 62

primitiv::Optimizer::load (C++ function), 62

primitiv::Optimizer::reset\_gradients (C++ function), 63

primitiv::Optimizer::save (C++ function), 62

primitiv::Optimizer::set\_configs (C++ function), 63

primitiv::Optimizer::set\_epoch (C++ function), 62

primitiv::Optimizer::set\_gradient\_clipping (C++ function), 63

primitiv::Optimizer::set\_learning\_rate\_scaling (C++ function), 62

primitiv::Optimizer::set\_weight\_decay (C++ function), 62

primitiv::Optimizer::update (C++ function), 63

primitiv::optimizers::AdaDelta (C++ class), 65

primitiv::optimizers::AdaDelta::AdaDelta (C++ function), 65

primitiv::optimizers::AdaDelta::eps (C++ function), 66

primitiv::optimizers::AdaDelta::rho (C++ function), 66

primitiv::optimizers::AdaGrad (C++ class), 64

primitiv::optimizers::AdaGrad::AdaGrad (C++ function), 64

primitiv::optimizers::AdaGrad::eps (C++ function), 65

primitiv::optimizers::AdaGrad::eta (C++ function), 65

primitiv::optimizers::Adam (C++ class), 66

primitiv::optimizers::Adam::Adam (C++ function), 66

primitiv::optimizers::Adam::alpha (C++ function), 66

primitiv::optimizers::Adam::beta1 (C++ function), 66

primitiv::optimizers::Adam::beta2 (C++ function), 66

primitiv::optimizers::Adam::eps (C++ function), 66

primitiv::optimizers::MomentumSGD (C++ class), 64

primitiv::optimizers::MomentumSGD::eta (C++ function), 64

primitiv::optimizers::MomentumSGD::momentum (C++ function), 64

primitiv::optimizers::MomentumSGD::MomentumSGD (C++ function), 64

primitiv::optimizers::RMSProp (C++ class), 65

primitiv::optimizers::RMSProp::alpha (C++ function), 65

primitiv::optimizers::RMSProp::eps (C++ function), 65

primitiv::optimizers::RMSProp::eta (C++ function), 65

primitiv::optimizers::RMSProp::RMSProp (C++ function), 65

primitiv::optimizers::SGD (C++ class), 64

primitiv::optimizers::SGD::eta (C++ function), 64

primitiv::optimizers::SGD::SGD (C++ function), 64

primitiv::Parameter (C++ class), 66

primitiv::Parameter::add\_stats (C++ function), 69

primitiv::Parameter::device (C++ function), 70

primitiv::Parameter::gradient (C++ function), 70

primitiv::Parameter::has\_stats (C++ function), 70

primitiv::Parameter::init (C++ function), 68

primitiv::Parameter::load (C++ function), 68, 69

primitiv::Parameter::Parameter (C++ function), 67

primitiv::Parameter::reset\_gradient (C++ function), 69

primitiv::Parameter::save (C++ function), 69

primitiv::Parameter::shape (C++ function), 70

primitiv::Parameter::stats (C++ function), 70

primitiv::Parameter::valid (C++ function), 69

primitiv::Parameter::value (C++ function), 70

primitiv::Shape (C++ class), 71

primitiv::Shape::batch (C++ function), 71

primitiv::Shape::depth (C++ function), 71

`primitiv::Shape::dims` (C++ function), [71](#)  
`primitiv::Shape::has_batch` (C++ function), [72](#)  
`primitiv::Shape::has_compatible_batch` (C++ function),  
[72](#)  
`primitiv::Shape::has_same_dims` (C++ function), [73](#)  
`primitiv::Shape::has_same_loo_dims` (C++ function), [73](#)  
`primitiv::Shape::is_column_vector` (C++ function), [72](#)  
`primitiv::Shape::is_matrix` (C++ function), [72](#)  
`primitiv::Shape::is_scalar` (C++ function), [72](#)  
`primitiv::Shape::lower_volume` (C++ function), [71](#)  
`primitiv::Shape::operator`  
  `=` (C++ function), [72](#)  
`primitiv::Shape::operator==` (C++ function), [72](#)  
`primitiv::Shape::operator[]` (C++ function), [71](#)  
`primitiv::Shape::resize_batch` (C++ function), [73](#)  
`primitiv::Shape::resize_dim` (C++ function), [73](#)  
`primitiv::Shape::Shape` (C++ function), [71](#)  
`primitiv::Shape::size` (C++ function), [72](#)  
`primitiv::Shape::to_string` (C++ function), [72](#)  
`primitiv::Shape::update_batch` (C++ function), [73](#)  
`primitiv::Shape::update_dim` (C++ function), [73](#)  
`primitiv::Shape::volume` (C++ function), [71](#)  
`primitiv::Tensor` (C++ class), [73](#)  
`primitiv::Tensor::argmax` (C++ function), [74](#)  
`primitiv::Tensor::argmin` (C++ function), [74](#)  
`primitiv::Tensor::check_valid` (C++ function), [74](#)  
`primitiv::Tensor::device` (C++ function), [74](#)  
`primitiv::Tensor::flatten` (C++ function), [75](#)  
`primitiv::Tensor::inplace_add` (C++ function), [75](#)  
`primitiv::Tensor::inplace_multiply_const` (C++ function),  
[75](#)  
`primitiv::Tensor::inplace_subtract` (C++ function), [75](#)  
`primitiv::Tensor::invalidate` (C++ function), [74](#)  
`primitiv::Tensor::reset` (C++ function), [75](#)  
`primitiv::Tensor::reset_by_array` (C++ function), [75](#)  
`primitiv::Tensor::reset_by_vector` (C++ function), [75](#)  
`primitiv::Tensor::reshape` (C++ function), [75](#)  
`primitiv::Tensor::shape` (C++ function), [74](#)  
`primitiv::Tensor::Tensor` (C++ function), [74](#)  
`primitiv::Tensor::to_float` (C++ function), [74](#)  
`primitiv::Tensor::to_vector` (C++ function), [74](#)  
`primitiv::Tensor::valid` (C++ function), [74](#)