primitiv Documentation

Release 0.4.0

primitiv Developer Group

Feb 22, 2019

Contents:

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.

CHAPTER 1

Installing primitiv

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 requries 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)
  - CLBlast 1.2.0

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 hardwares, 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:

```
$ ce /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
```

1.4 Installing primitiv with OpenCL

1.4.1 Installing OpenCL C++ Headers

```
$ git clone https://github.com/KhronosGroup/OpenCL-CLHPP.git
$ cd OpenCL-CLHPP
$ mkdir build
$ cd build
$ cmake .. [OPTIONS]  # See: https://github.com/KhronosGroup/OpenCL-CLHPP
$ make && make install
```

1.4.2 Installing CLBlast

```
$ apt install wget
$ wget https://github.com/CNugteren/CLBlast/archive/1.2.0.tar.gz -O ./clblast.tar.gz
$ mkdir clblast
$ cd clblast
$ tar xf ../clblast.tar.gz --strip-components 1
$ mkdir build
$ cd build
$ cmake .. [OPTIONS]  # See: https://github.com/CNugteren/CLBlast
$ make && make install
```

1.4.3 Configuring primitiv with OpenCL

The following command configures to build the OpenCL backend using system libraries.

```
$ cmake .. -DPRIMITIV_USE_OPENCL=ON
```

The build process tries to find the OpenCL library, the OpenCL C++ headers, and the CLBlast library by default. You can also specify the explicit locations of their libraries if searching failed or you want to switch them:

```
$ cmake .. \
  -DOpenCL_INCLUDE_DIR=/path/to/opencl/include \
  -DOpenCL_LIBRARY=/path/to/libOpenCL.so \
  -DCLHPP_INCLUDE_DIR=/path/to/clhpp/include \
  -DCLBLAST_ROOT=/path/to/clblast/prefix \
  -DPRIMITIV_USE_OPENCL=ON
```

CHAPTER 2

primitiv C++ Tutorials

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:

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

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-separatable*, 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.
```

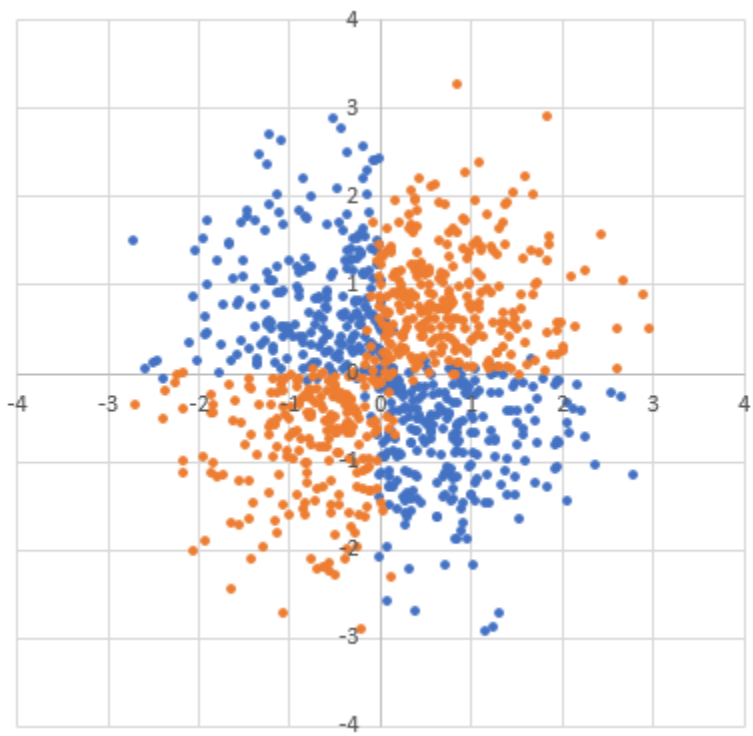
(continues on next page)

(continued from previous page)

```
  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:

$$y := \tanh(W_{hy}\boldsymbol{h} + b_y),$$
$$\boldsymbol{h} := \tanh(W_{xh}\boldsymbol{x} + \boldsymbol{b}_h),$$

where $y \in \mathbb{R}$ is an output value to be fit to $f(x_1, x_2)$, $\boldsymbol{x} := (x_1\ x_2)^\top \in \mathbb{R}^2$ is an input vector, $\boldsymbol{h} \in \mathbb{R}^N$ represents the N-dimentional *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 $\boldsymbol{b}_h \in \mathbb{R}^N$.

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 ths 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}, $\boldsymbol{b}_h$, W_{hy} and b_y. We first specify these parameters as `Parameter` objects:

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

(continues on next page)

(continued from previous page)

```
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);
```

(continues on next page)

(continued from previous page)

```
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
...
```

(continues on next page)

(continued from previous page)

```
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.

CHAPTER 3

Library Designs

This section describes concepts and designs of primitiv.

3.1 Header and Library Files

TBD.

3.2 Operation Rules with Minibatching

TBD.

3.3 Lazy Evaluation

TBD.

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.

3.9 Nodes and Tensors

TBD.

CHAPTER 4

primitiv Reference

This section contains low-level information about primitiv.

4.1 primitiv API Reference

4.1.1 Devices

Base Class

class Device : **public** primitiv::mixins::DefaultSettable<*Device*>, primitiv::mixins::Nonmovable<*Device*>

Interface of the *Tensor* provider.

Subclassed by *primitiv::devices::CUDA*, primitiv::devices::CUDA16, *primitiv::devices::Eigen*, *primitiv::devices::Naive*, *primitiv::devices::OpenCL*

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 udpated.

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 Naive : **public** primitiv::*Device*

Device class for the naive function implementations on CPU.

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.

DeviceType **type**() **const**

Retrieves the type of the device.

Return A DeviceType value.

class Eigen : **public** primitiv::*Device*

Device class for the Eigen3 backend.

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.

DeviceType **type**() **const**

Retrieves the type of the device.

Return A DeviceType value.

class CUDA : **public** primitiv::*Device*

Device class for *CUDA*.

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.

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 OpenCL : **public** primitiv::*Device*
Device class for *OpenCL*.

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.

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 functions

Functions

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:

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

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

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

Parameters

- `x`: A variable representing an argument x.
- `a`: A scaling factor α.
- `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()` end `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()` end `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 defice.

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 defice.

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:

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

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.

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 exponentation 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 exponentation 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 exponentation 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 exponentation 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:

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

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

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

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:

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

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 Var**>**
std::vector<type_traits::Identity<Var>> **split**(**const** Var &*x*, std::uint32_t *dim*, std::uint32_t *n*)

Splits a given variable into specified number of partitions along an axis.

Return A list of `n` variables. Each variable is identical with: `split(x, dim, n)[i] == slice(x, dim, L(i), U(i))`, where `L(i) := i * x.shape()[dim] / n` and `U(i) := (i + 1) * x.shape()[dim] / n`.

Parameters

- `x`: A variable representing an original data.

- `dim`: Axis to be processed.
- `n`: The number of resulting partitions. `n` should be able to divide `x.shape()[dim]` without residue.

Exceptions

- `primitiv::Error`: `n` can not divide `s.shape()[dim]` without residue.

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:

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

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 $X^\top$.

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> **flip**(**const** Var &*x*, std::uint32_t *dim*)

Flips elements along an axis. Following examples show how this function work:

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

Return A new variable representing $\mathrm{flip}(x)$.

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> **permute_dims**(**const** Var &*x*, **const** std::vector<std::uint32_t> &*perm*)

Permutes dimensions of a tensor.

$$
\begin{aligned}
x &:= \left(\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}, \begin{pmatrix} 11 & 14 \\ 12 & 15 \\ 13 & 16 \end{pmatrix}, \begin{pmatrix} 21 & 24 \\ 22 & 25 \\ 23 & 26 \end{pmatrix}, \begin{pmatrix} 31 & 34 \\ 32 & 35 \\ 33 & 36 \end{pmatrix}\right), \\
\mathrm{permute\_dims}(x, [1, 2, 0]) &= \left(\begin{pmatrix} 1 & 11 & 21 & 31 \\ 4 & 14 & 24 & 34 \end{pmatrix}, \begin{pmatrix} 2 & 12 & 22 & 32 \\ 5 & 15 & 25 & 35 \end{pmatrix}, \begin{pmatrix} 3 & 13 & 23 & 33 \\ 6 & 16 & 26 & 36 \end{pmatrix}\right), \\
\mathrm{permute\_dims}(x, [2, 0, 1]) &= \left(\begin{pmatrix} 1 & 2 & 3 \\ 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{pmatrix}, \begin{pmatrix} 4 & 5 & 6 \\ 14 & 15 & 16 \\ 24 & 35 & 36 \\ 34 & 35 & 36 \end{pmatrix}\right), \\
\mathrm{permute\_dims}(x, [0, 1, 2]) &= x.
\end{aligned}
$$

Return A new variable.

Parameters

- `x`: A variable representing an original data.
- `perm`: A list of dimensions for specifying permutation.

template <typename Var>
type_traits::Identity<Var> **permute_dims** (**const** Var &*x*, **const** std::initializer_list<std::uint32_t> *perm*)

Permutes dimensions of a tensor.

Return A new variable.

Parameters

- `x`: A variable representing an original data.
- `perm`: A list of dimensions for specifying permutation.

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> **abs** (**const** Var &*x*)

Applies an elementwise absolute function.

Return A variable representing $|x|$.

Parameters

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

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:

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

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

Parameters

- x: A variable representing an argument x.

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

Applies an elementwise softplus function:

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

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

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:

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

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

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:

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

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

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:

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

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

Parameters

- x: A variable representing an argument x.
- a: A scaling factor α.

template <typename Var**>**
type_traits::Identity<Var> **elu** (**const** Var &*x*, float *a*)

Applies an elementwise exponential linear unit (ELU) function:

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

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

Parameters

- `x`: A variable representing an argument x.
- `a`: A scaling factor α.

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

Retrieves maximum values along an axis. Following examples show how this function work:

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

Return A new variable.

Parameters

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

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

Retrieves minimum values along an axis. Following examples show how this function work:

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

Return A new variable.

Parameters

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

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

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

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

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:

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

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:

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

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:

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

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*.

namespace batch

Functions

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:

$$\begin{aligned} m_x &:= \frac{1}{B}\sum_{i=1}^{B} x_i, \\ v_x &:= \frac{B}{B-1}\left(\frac{1}{B}\sum_{i=0}^{B} x_i^2 - m_x^2\right), \\ \text{batch :: normalize}(x) &:= \frac{x-m_x}{\sqrt{v_x+\epsilon}}, \end{aligned}$$

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.

template <typename Var**>**
type_traits::Identity<Var> **pick** (**const** Var &*x*, **const** std::vector<std::uint32_t> &*ids*)

Selects items from the batch by IDs.
Return A new variable.
Parameters
- `x`: A variable representing an original data.
- `ids`: List of batch IDs. Each value must be lower than `x.shape().batch()`.

template <typename Var**>**
type_traits::Identity<Var> **slice** (**const** Var &*x*, std::uint32_t *lower*, std::uint32_t *upper*)

Extracts a specific range $[L, U)$ of items along the batch axis.
Return A new variable.
Parameters
- `x`: A variable representing an original data.
- `lower`: Lower bound L of the batch.
- `upper`: Upper bound U of the batch.

template <typename Var**>**
std::vector<type_traits::Identity<Var>> **split** (**const** Var &*x*, std::uint32_t *n*)

Splits a given variable into specified number of partitions along the batch.
Return A list of `n` variables. Each variable is identical with: `batch::split(x, n)[i] == batch::slice(x, L(i), U(i))`, where `L(i) := i * x.shape().batch() / n` and `U(i) := (i + 1) * x.shape().batch() / n`.
Parameters
- `x`: A variable representing an original data.
- `n`: The number of resulting partitions. `n` should be able to divide `x.shape().batch()` without residue.

Exceptions
- `primitiv::Error`: `n` can not divide `s.shape().batch()` without residue.

template <typename Container**>**
type_traits::Reduce<Container> **concat** (**const** Container &*xs*)

Concatenates multiple variables along the batch axis.
Return A new variable.
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> **concat**(**const** Container &*xs*)

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.

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

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

$$
\begin{aligned}
x &:= \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}, \begin{pmatrix} 11 & 14 & 17 \\ 12 & 15 & 18 \\ 13 & 16 & 19 \end{pmatrix}, \\
\mathrm{batch} :: \mathrm{sum}(x) &= \begin{pmatrix} 12 & 18 & 24 \\ 14 & 20 & 26 \\ 16 & 22 & 28 \end{pmatrix}.
\end{aligned}
$$

Return A new variable.

Parameters

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

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 μ of the normal distribution.
- `sd`: The standard deviation σ 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 μ of the normal distribution.
- `sd`: The standard deviation σ 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 μ of the normal distribution.
- `sd`: The standard deviation σ 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 μ of the normal distribution.
- `sd`: The standard deviation σ 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 μ of the normal distribution.
- `sd`: The standard deviation σ 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 μ of the log-normal distribution.
- `sd`: The parameter σ 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 μ of the log-normal distribution.
- `sd`: The parameter σ 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 μ of the log-normal distribution.
- `sd`: The parameter σ 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 μ of the log-normal distribution.
- `sd`: The parameter σ 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 μ of the log-normal distribution.
- `sd`: The parameter σ 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 μ of the Gumbel distribution.
- `beta`: The scale parameter β 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 μ of the Gumbel distribution.
- `beta`: The scale parameter β 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 μ of the Gumbel distribution.
- `beta`: The scale parameter β 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 μ of the Gumbel distribution.
- `beta`: The scale parameter β 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 μ of the Gumbel distribution.
- `beta`: The scale parameter β of the Gumbel distribution.

4.1.3 Graph

class Graph : **public** primitiv::mixins::DefaultSettable<*Graph*>, primitiv::mixins::Nonmovable<*Graph*>

Computation 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.

std::vector<*Node*> **add_operator**(std::unique_ptr<Operator> &&*op*, **const** std::vector<*Node*> &*args*)

Adds an operator into the graph.

Return New *Node* objects of resulting values.

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 Initializer : primitiv::mixins::Nonmovable<*Initializer*>
Abstract class to provide parameter initialization algorithms.

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 Constant : **public** primitiv::*Initializer*
Initializer to generate a same-value tensor.

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 Uniform : **public** primitiv::*Initializer*

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

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 Normal : **public** primitiv::*Initializer*

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

Public Functions

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

Creates a new `Normal` initializer.

Parameters

- `mean`: Mean μ of the normal distribution.
- `sd`: Standard deviation σ of the normal distribution.

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

Provides an initialized tensor.

Parameters

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

class Identity : **public** primitiv::*Initializer*

Identity matrix 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 XavierUniform : **public** primitiv::*Initializer*

The Xavier matrix initialization with the uniform distribution.

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 XavierNormal : **public** primitiv::*Initializer*
The Xavier matrix initialization with the normal distribution.

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 XavierUniformConv2D : **public** primitiv::*Initializer*
The Xavier initialization with the uniform distribution for conv2d filters.

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 XavierNormalConv2D : **public** primitiv::*Initializer*
The Xavier initialization with the normal distribution for conv2d filters.

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 Model : primitiv::mixins::Nonmovable<*Model*>
Set of parameters and specific algorithms.

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 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 Optimizer : primitiv::mixins::Nonmovable<*Optimizer*>
Abstract class for parameter optimizers.

Subclassed by *primitiv::optimizers::AdaDelta*, *primitiv::optimizers::AdaGrad*, *primitiv::optimizers::Adam*, *primitiv::optimizers::MomentumSGD*, *primitiv::optimizers::RMSProp*, *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 SGD : **public** primitiv::*Optimizer*

Simple stochastic gradient descent.

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 MomentumSGD : **public** primitiv::*Optimizer*

Stochastic gradient descent with momentum.

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 AdaGrad : **public** primitiv::*Optimizer*

AdaGrad optimizer.

Public Functions

primitiv::optimizers::AdaGrad::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 RMSProp : **public** primitiv::*Optimizer*
RMSProp Optimizer.

Public Functions

primitiv::optimizers::RMSProp::RMSProp(float eta = 0.01, float alpha = 0.9, float eps =
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 AdaDelta : **public** primitiv::*Optimizer*
AdaDelta optimizer. https://arxiv.org/abs/1212.5701

Public Functions

primitiv::optimizers::AdaDelta::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 Adam : **public** primitiv::*Optimizer*
Adam optimizer. https://arxiv.org/abs/1412.6980

Public Functions

primitiv::optimizers::Adam::Adam(float alpha = 0.001, float beta1 = 0.9, float beta2 =
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 Parameter : primitiv::mixins::Nonmovable<*Parameter*>
Class to manage a trainable tensor 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 opotional statistics tensor specified by given name.

Return A tensor.

Parameters

- `name`: Name of the statistics.

Tensor &**`stats`**(**`const`** std::string &*name*)

Returns the current opotional statistics tensor specified by given name.

Return A tensor.

Parameters

- `name`: Name of the statistics.

4.1.9 Shape

class 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 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 a re 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. FindCuda options can also be used.

PRIMITIV_USE_CUDNN Default value: `OFF`

Enables cuDNN as the backend of few CUDA functions. This option introduces a dependency to the cuDNN library v5.0 or later. 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

Index

P