High-Level Synthesizable Deep Learning Routines with Python/PyTorch Interfaces

Version 0 Revision 6

Dec. 17, 2020 (July 23, 2020)

Future Design Systems, Inc.
www.future-ds.com / contact@future-ds.com

Abstract

This document addresses Deep Learning Routines that are High-Level synthesizable. In addition to the C routines, Python Interfaces for PyTorch are included in order to verify functionality of the routines along with PyTorch deep learning framework.

Table of Contents

1 Overview

DLR (*Deep Learning Routines*) as a part of **DPU** (*Deep Learning Processing Unit*) is a collection of high-level synthesizable C routines for deep learning inference network. **DLIP** (*Deep Learning Intellectual Property*) is HDL (Hardware Description Language) version of DLR.

DLR contain followings[1]:

- ✧ Convolution layer (strict convolution, matrix-multiplication-based convolution)
- ✧ Activation layer (ReLU, Leaky ReLU, hyperbolic tangent, sigmoid)
- ✧ Pooling layer (max, average)
- ✧ Fully connected layer (linear layer)
- ✧ De-convolution layer (transposed convolution layer)
- ✧ Batch normalization layer
- ✧ Concatenation layer

The routines have following highlights:

- ✧ Fully synthesizable C code
- ✧ Highly parameterized to be adopted wide range of usages
- ✧ C, Python, PyTorch, Verilog test-benches
- ✧ FPGA verified using Future Design Systems' CON-FMC

It should be noted that the routines are not optimized to get a higher performance since the routines are for hardware implementation not for computation. In addition to this the routines are only for inference not for training.

1.1 Verification flow

Figure 1 shows an overall verification flow, in which following interfaces are supported.

- Standard C/C++
- PyTorch C++ Front-End
- PyTorch

[1] Some are under development and more will be added.

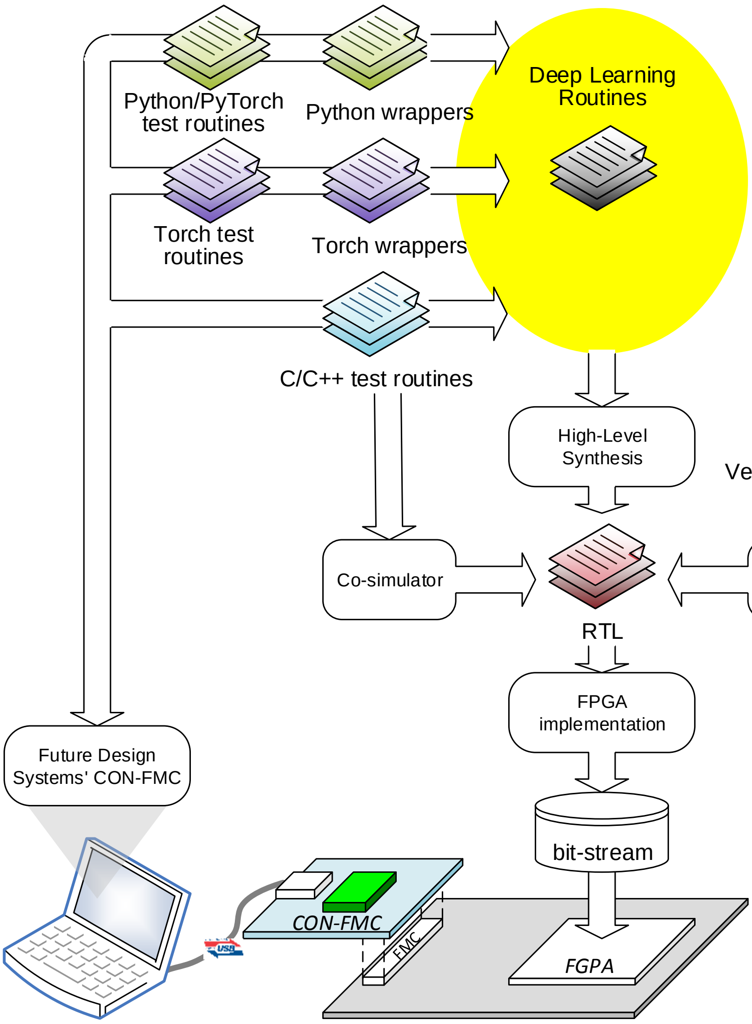

Figure 1: Verification flow

2 License

DLR (Deep Learning Routines) and its associated materials are licensed with the 2-clause BSD license to make the program and library useful in open and closed source products independent of their licensing scheme.

3 Convention and background

3.1 Naming convention

This routine uses following coding conventions.

- ✧ 4 spaces for indentation, not tab
- ✧ Variables uses underscores separating words.
  - char item_delimiter;
- ✧ Functions starts with upper case character and follows carmel-case.
  - void Convolution2dWrapper(int stride);
- ✧ Macros uses all capital letter with underscores separating words.
  - #define MAX_NUM 128
- ✧ Temporary variable starts underscore or ‘t_’ prefix.
  - int t_tmp;
- ✧ Output comes first in the function argument list.
- ✧ Python wrapper functions follow format of DRL of C.
- ✧ PyTorch wrapper functions follow format of PyTorch nn.functional.

3.2 Multi-dimensional array

This routine uses row-major contiguous array layout, which is the same as that of the standard C language.

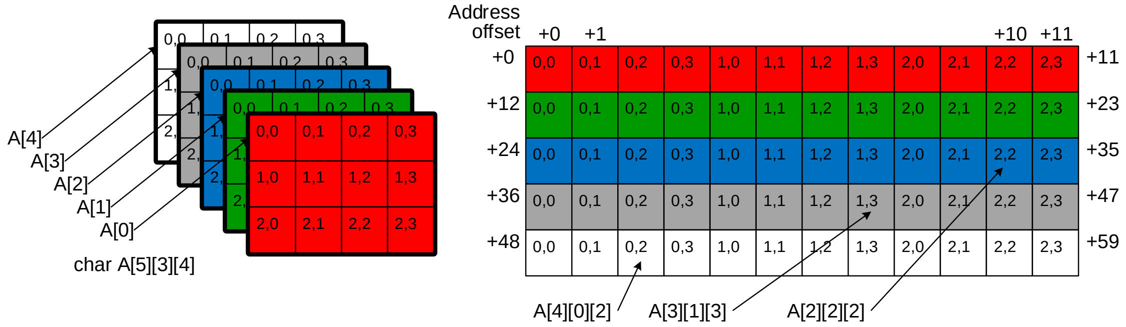

Figure 2: Multi-dimensional array (one-byte data case)

3.3 Python wrapper

Python wrapper uses Ctypes (a foreign function library for Python) to call C routine.

DLIP (Deep Learning IP) Python wrapper uses Python NumPy array to carry array data. As a result, PyTorch tensor should be converted to NumPy arrary in order to be called from PyTorch.

3.4 PyTorch wrapper

PyTorch wrapper supports torch.nn.funtional that is stateless functional version.

3.5 Hardware interface

DLIP (Deep Learning IP) has two AMB AXI bus interfaces; one is for accessing internal registers and the other is for access shared memory.

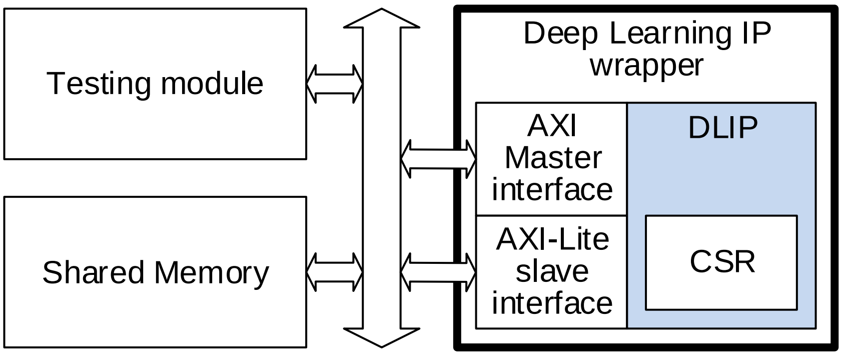

Figure 3: Deep Learning IP HW structure including test bench

3.6 Software interface

Following explains the structure of CSR (Control and Status Register) in the DLIP.

```
// ctl
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/COH)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x10 : Data signal of shared_mem
//        bit 31~0 - shared_mem[31:0] (Read/Write)
// 0x14 : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on
Handshake)
```

Following shows an example to control DLIP through CSR.

Wait for ready before start	Let DLIP runs and wait for completion

#define WAIT_FOR_READY {\
   int ap_idle, ap_idle_r;\
   unsigned int ap_addr;\
	#define GO_AND_WAIT_COMPLETE {\
	   int ap_done, ap_done_r;\
	   int ap_start, ap_data;\

ap_addr = ADDR_CSR;\
while (1) {\
  MEM_READ(ap_addr, ap_idle_r);\
  ap_idle = (ap_idle_r >> 2) && 0x1;\
  if (ap_idle) break;\
}}
	unsigned int ap_addr;\
	ap_addr = ADDR_CSR;\
	ap_data = 0x1;\
	MEM_WRITE(ap_addr, ap_data);\
	while (1) {\
	  MEM_READ(ap_addr, ap_done_r);\
	  ap_done = (ap_done_r >> 1) && 0x1;\
	  if (ap_done) break;\
	}}

4 Activation

4.1 ReLU

‘ActivationReLU()’ applies Rectified Linier Unit over an input data.

```
#include "activation_relu.hpp"
```

```
template<class TYPE=float>
void ActivationReLu
(           TYPE      *out_data
   , const TYPE      *in_data
   , const uint32_t  size
   , const uint16_t  channel
   #if !defined(__SYNTHESIS__)
   , const int       rigor=0
   , const int       verbose=0
   #endif
);
```

Templates:

- TYPE: data type of out_data, in_data
  - int, float, double, and so on[2].

Arguments:

- out_data: resultant data in contiguous memory and can be any dimension
  - channel x size x size
- in_data: input array in contiguous memory and can be any dimension
  - channel x size x size
- size: the number of elements per channel
- channel: the number of channels
- rigor: check values rigorously when 1
- verbose: print more message when 1

Array size must keep following rules.

- ‘out_data’ and ‘in_array’ must have ‘channel x size’ elements.

[2] Fixed-point data type will be supported.

4.1.1 Python wrapper

'ActivationReLu()' wrapper gets NumPy arguments for array and 'out_data' carries calculated result. It returns 'True' on success or 'False' on failure.

```
ActivationReLu( out_data # any dimension
              , in_data  # any dimension
              , rigor=False
              , verbose=False)
```

4.1.2 PyTorch wrapper

'relu()' PyTorch wrapper gets PyTorch tensor arguments for array and returns calculated result. It calls Python wrapper after converting PyTorch tensor to NumPy array.

```
relu( input # any dimension
              , rigor=False
              , verbose=False)
```

4.2 Leaky ReLU

'ActivationLeakyReLU()' applies Rectified Linier Unit over an input data.

```
#include "activation_leakyrelu.hpp"
```

```
template<class TYPE =float>
void ActivationLeakyReLu
(           TYPE     *out_data
   , const TYPE     *in_data
   , const uint32_t  size
   , const uint16_t  channel
   , const uint32_t  negative_slope=0x3DCCCCCD
   #if !defined(__SYNTHESIS__)
   , const int       rigor=0
   , const int       verbose=0
   #endif
);
```

Templates:

- ✧ DTYPE: data type of out_data, in_data
  - ▫ int, float, double, and so on[3].

Arguments:

- ✧ out_data: resultant data in contiguous memory and can be any dimension
  - ▫ channel x size x size
- ✧ in_data: input array in contiguous memory and can be any dimension
  - ▫ channel x size x size

[3] Fixed-point data type will be supported.

- ✧ size: the number of elements per channel
- ✧ channel: the number of channels
- ✧ negative_slope: apply when in_data[] is negative; bit pattern of float 0.01 by default.
  - □ Note that it should carry bit-pattern
- ✧ rigor: check values rigorously when 1
- ✧ verbose: print more message when 1

Array size must keep following rules.

- ● 'out_data' and 'in_array' must have 'channel x size' elements.

4.2.1 Python wrapper

'ActivationLeakyReLu()' wrapper gets NumPy arguments for array and 'out_data' carries calculated result. It returns 'True' on success or 'False' on failure.

```
ActivationLeakyReLu( out_data # any dimension
                   , in_data  # any dimension
                   , negative_slope
                   , rigor=False
                   , verbose=False)
```

4.2.2 PyTorch wrapper

'leaky_relu()' PyTorch wrapper gets PyTorch tensor arguments for array and returns calculated result. It calls Python wrapper after converting PyTorch tensor to NumPy array.

```
leaky_relu( input  # any dimension
          , negative_slope
          , rigor=False
          , verbose=False)
```

4.3 Hyperbolic tangent

'ActivationTanh()' applies hyperbolic tangent over an input data.

```
#include "activation_tanh.hpp"
```

```
template<class TYPE=float>
void ActivationTanh
(         TYPE      *out_data
   , const TYPE     *in_data
   , const uint32_t  size
   , const uint16_t  channel
   #if !defined(__SYNTHESIS__)
   , const int       rigor=0
   , const int       verbose=0
```

```
    #endif
);
```

Templates:

- ✧ DTYPE: data type of out_data, in_data
    - □ int, float, double, and so on[4].

Arguments:

- ✧ out_data: resultant data in contiguous memory and can be any dimension
    - □ channel x size x size
- ✧ in_data: input array in contiguous memory and can be any dimension
    - □ channel x size x size
- ✧ size: the number of elements per channel
- ✧ channel: the number of channels
- ✧ rigor: check values rigorously when 1
- ✧ verbose: print more message when 1

Array size must keep following rules.

- ‘out_data’ and ‘in_array’ must have ‘channel x size’ elements.

4.3.1 Python wrapper

‘ActivationTanh()’ wrapper gets NumPy arguments for array and ‘out_data’ carries calculated result. It returns ‘True’ on success or ‘False’ on failure.

```
ActivationTanh ( out_data # any dimension
               , in_data  # any dimension
               , rigor=False
               , verbose=False)
```

4.3.2 PyTorch wrapper

‘tanh()’ PyTorch wrapper gets PyTorch tensor arguments for array and returns calculated result. It calls Python wrapper after converting PyTorch tensor to NumPy array.

```
tanh( input # any dimension
               , rigor=False
               , verbose=False)
```

4.4 Sigmoid

‘ActivationSigmoid()’ applies sigmoid over an input data.

[4] Fixed-point data type will be supported.

```
#include "activation_sigmoid.hpp"
```

```
template<class TYPE=float>
void ActivationSigmoid
(          TYPE     *out_data
   , const TYPE     *in_data
   , const uint32_t  size
   , const uint16_t  channel
   #if !defined(__SYNTHESIS__)
   , const int       rigor=0
   , const int       verbose=0
   #endif
);
```

Templates:
- ✧ DTYPE: data type of out_data, in_data
  - ◻ int, float, double, and so on[5].

Arguments:
- ✧ out_data: resultant data in contiguous memory and can be any dimension
  - ◻ channel x size x size
- ✧ in_data: input array in contiguous memory and can be any dimension
  - ◻ channel x size x size
- ✧ size: the number of elements per channel
- ✧ channel: the number of channel
- ✧ rigor: check values rigorously when 1
- ✧ verbose: print more message when 1

Array size must keep following rules.

- ‘out_data’ and ‘in_array’ must have ‘channel x size’ elements.

4.4.1 Python wrapper

‘ActivationSigmoid()’ wrapper gets NumPy arguments for array and ‘out_data’ carries calculated result. It returns ‘True’ on success or ‘False’ on failure.

```
ActivationSigmoid( out_data # any dimension
                 , in_data  # any dimension
                 , rigor=False
                 , verbose=False)
```

4.4.2 PyTorch wrapper

‘sigmoid()’ PyTorch wrapper gets PyTorch tensor arguments for array and returns calculated result. It calls Python wrapper after converting PyTorch tensor to NumPy array.

---

[5] Fixed-point data type will be supported.

```
sigmoid( input # any dimension
       , rigor=False
       , verbose=False)
```

5 Concatenation

5.1 Concat 2D

‘Concat2d()’ combines two 2-dimentional arrays.

```
#include “concat_2d.hpp”
```

```
template<class TYPE=float>
void Concat2d
(       TYPE    *out_data
  , const TYPE    *in_dataA
  , const TYPE    *in_dataB
  , const uint16_t  in_rowsA
  , const uint16_t  in_colsA
  , const uint16_t  in_rowsB
  , const uint16_t  in_colsB
  , const uint8_t   dim
  #if !defined(__SYNTHESIS__)
  , const int      rigor=0
  , const int      verbose=0
  #endif
);
```

Templates:

- ✧ DTYPE: data type of out_data, in_data
  - □ int, float, double, and so on[6].

Arguments:

- ✧ out_data[][]: resultant 2D array
  - □ in_rowsA x (in_colsA+in_colsB) when dim is 1
  - □ (in_rowsA+in_rowsB) x in_colsA when dim is 0
- ✧ in_dataA[in_rowA][in_colA]: input array (rows:height, cols:width)
- ✧ in_dataB[in_rowB][in_colB]: input array (rows:height, cols:width)
- ✧ dim: append array B at the end of array A when dim is 0; append each row of array B at the end of each row of array A when dim is 1.
- ✧ rigor: check values rigorously when 1
- ✧ verbose: print more message when 1

Array size must keep following rules.

- ‘out_data’, ‘in_arrayA’ and ‘in_dataB’ are 2 dimensional array.
- ‘dim’ can be 0 or 1.

[6] Fixed-point data type will be supported.

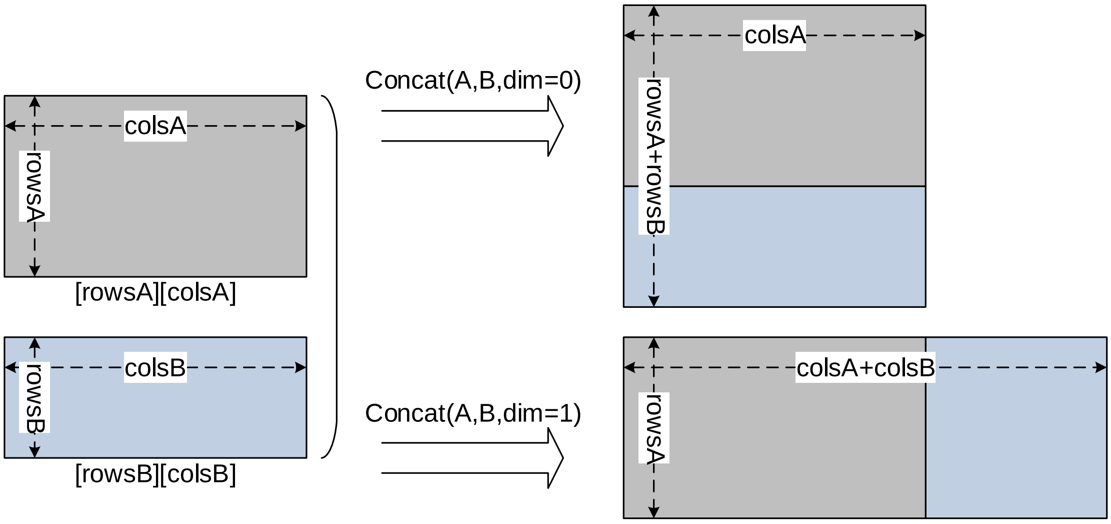

Figure 4: Concat2d example

5.1.1 Python wrapper

'Concat2d()' wrapper gets NumPy arguments for array and 'out_data' carries calculated result. It returns 'True' on success or 'False' on failure.

```
Concat2d( out_data  # in_rowsA x (in_colsA+in_colsB) when dim is 1
                    # (in_rowsA+in_rowsB) x in_colsA when dim is 0
        , in_dataA  # in_rowsA x in_colsA
        , in_dataB  # in_rowsB x in_colsB
        , dim=0     # out_size
        , rigor=False
        , verbose=False)
```

5.1.2 PyTorch wrapper

'cat()' PyTorch wrapper gets PyTorch tensor arguments for array and returns calculated result. It calls Python wrapper after converting PyTorch tensor to NumPy array.

```
cat( (inputA
     , inputB
     , dim=0
     , rigor=False
     , verbose=False)
```

6 Convolution

6.1 Convolution2d

‘Convolution2d()’ applies a 2D convolution over an input data composed of several planes (channels).

```
#include “convolution_2d.hpp”
```

```
template<class TYPE=float>
void Convolution2d
(          TYPE    *out_data
   , const TYPE    *in_data
   , const TYPE    *kernel
   , const TYPE    *bias
   , const uint16_t  out_size
   , const uint16_t  in_size
   , const uint8_t   kernel_size
   , const uint16_t  bias_size
   , const uint16_t  in_channel
   , const uint16_t  out_channel
   , const uint8_t   stride
   , const uint8_t   padding=0
   #if !defined(__SYNTHESIS__)
   , const int       rigor=0
   , const int       verbose=0
   #endif
)
```

Templates:

- DTYPE: data type of out_data, in_data, kernel, bias
  - int, float, double, and so on[7].

Arguments:

- out_data[out_channel][out_size][out_size] resultant array in square
- in_data[in_channel][in_size][in_size] input feature array in square
- kernel[in_channel][out_channel][kernel_size][kernel_size] filter array in square
- bias[bias_size]: bias for output and each element corresponds each output channel
- out_size: the number of elements in width direction (column)
- in_size: the number of elements in width direction (column)
- kernel_size: the number of elements in width direction (column)
- bias_size: the number of elements in bias (zero by default)
- in_channel: the number of input channels
- out_channel: the number of output channels, i.e., filters; the number of channels produced by the convolution.
- strid: stride of the convolution (one by default)
- padding: the number of zero-paddings on both sides (zero by default, not implemented yet)
- rigor: check values rigorously when 1
- verbose: print more message when 1

[7] Fixed-point data type will be supported.

‘minibatch’, ‘dilation’, and ‘goups’ are not supported, but ‘minibatch’ is supported through PyTorch wrapper.

Array size must keep following rules.

- ‘out_data’, ‘in_data’, ‘kernel’ are square matrix
- ‘out_size’ = (((in_size-kernel_size+2*padding)/stride)+1)
- ‘kernel_size’ should be odd number
- ‘bias_size’ should be the same as ‘out_channel’, i.e., the number of kernels
- ‘stride’ should be >=1
- ‘padding’ should be >=0 and will be floor(kernel_size/2) normal case

Each data arrays are row-major contagious and kernel array can be seen as 4-dimentional array of [out_channel x in_chnnel x kerne_size x kernel_size].

Figure 5 shows an example case of three input channels and two output channels[8], in which biases are two and kernels are ‘out_channel x in_channel’.

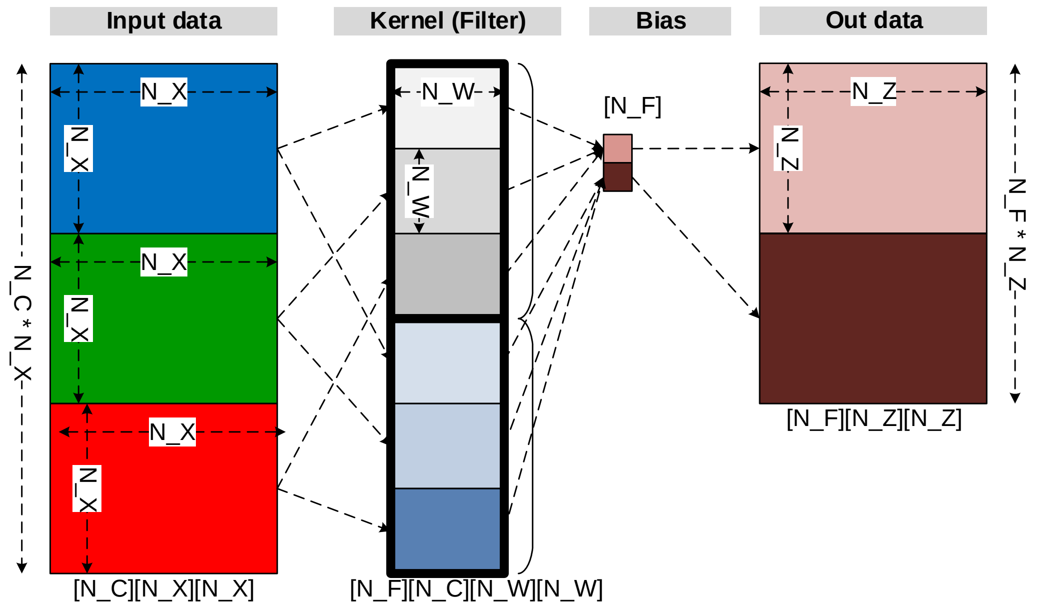

Figure 5: Convolution2d example with 3 input channels, 2 output channels

6.1.1 Python wrapper

‘Convolution2d()’ wrapper gets NumPy arguments for array and ‘out_data’ carries calculated result. It returns ‘True’ on success or ‘False’ on failure.

'IIIIj

```
Convolution2d( out_data    # out_channel x out_size x out_size
             , in_data     # in_channel x in_size x in_size
             , kernel      # out_channel x in_channel x kernel_size x kernel_size
             , stride=1
             , padding=0
             , bias=0
             , rigor=False
             , verbose=False)
```

6.1.2 PyTorch wrapper

'conv2d()' PyTorch wrapper gets PyTorch tensor arguments for array and returns calculated result. It calls Python wrapper after converting PyTorch tensor to NumPy array.

```
conv2d( input # minibatch x in_channel x in_size x in_size
      , weight # out_channel x in_channel x kernel_size x kernel_size
      , bias=None # out_channel
      , stride=1
      , padding=0
      , dilation=1
      , groups=1
      , rigor=False
      , verbose=False)
```

7 Deconvolution (Transposed convolution)

7.1 deconvolution2d

'DeConvolution2d()' applies a 2D transposed convolution over an input signal composed of several input planes (channels).

```
#include "deconvolution_2d.hpp"
```

```
template<class TYPE=float>
void Deconvolution2d
(           TYPE        *out_data
  , const TYPE          *in_data
  , const TYPE          *kernel
  , const TYPE          *bias
  , const uint16_t  out_size
  , const uint16_t  in_size
  , const uint8_t   kernel_size
  , const uint8_t   bias_size
  , const uint16_t  in_channel
  , const uint16_t  out_channel
  , const uint8_t   stride
  , const uint8_t   padding=0
  #if !defined(__SYNTHESIS__)
  , const int       rigor=0
  , const int       verbose=0
  #endif
```

```
);
```

Templates:
- ✧ DTYPE: data type of in_data, kernel, bias, out_data; int, float, double, and so on[9].

Arguments:
- ✧ out_data[out_channel][out_size][out_size] resultant array in square
- ✧ in_data[in_channel][in_size][in_size] input feature array in square
- ✧ kernel[in_channel][out_channel][kernel_size][kernel_size] filter array in square
- ✧ out_size: the number of elements in width direction (column)
- ✧ in_size: the number of elements in width direction (column)
- ✧ kernel_size: the number of elements in width direction (column)
- ✧ in_channel: the number of input channels
- ✧ out_channel: the number of output channels, i.e., filters; the number of channels produced by the convolution.
- ✧ strid: stride of the convolution (one by default)
- ✧ padding: the number of zero-paddings on both sides (zero by default, not implemented yet)
- ✧ bias[bias_size]: bias for output and each element corresponds each output channel
- ✧ bias_size: the number of elements in bias (zero by default)

'paddng', 'minibatch', 'dilation', 'goups' are not supported yet.

Array size must keep following rules.

- 'out_data', 'in_data', 'kernel' are squre matrix
- 'out_size' = (((in_size-kernel_size+2*padding)/stride)+1)
- 'kernel_size' should be odd number
- 'stride' should be positive and larger than 0 (should not be 0)
- 'padding' should be positive including 0 and will be floor(kernel_size/2)
- 'bias_size' should be 'out_channel'.

7.1.1 Python wrapper

'Deconvolution2d()' wrapper gets NumPy arguments for array and 'out_data' carries calculated result. It returns 'True' on success or 'False' on failure.

```
Deconvolution2d( out_data # any dimension
               , in_data  # any dimension
               , weight
               , bias=None
               , stride=1
               , padding=0
               , output_padding=0
```

[9] Fixed-point data type will be supported.

```
                 , groups=1
                 , dilation=1
                 , rigor=False
                 , verbose=False)
```

7.1.2 PyTorch wrapper

'conv_transpose2d()' PyTorch wrapper gets PyTorch tensor arguments for array and returns calculated result. It calls Python wrapper after converting PyTorch tensor to NumPy array.

```
conv_transpose2d( input # any dimension
                 , weight
                 , bias=None
                 , stride=1
                 , padding=0
                 , output_padding=0
                 , groups=1
                 , dilation=1
                 , rigor=False
                 , verbose=False)
```

8 Linear (Fully connected)

8.1 Linear 1D

'Linear1d()' applies a 1D vector multiplication over an input data.

```
#include "linear_1d.hpp"
```

```
template< class TYPE=float
        , int ReLu=0
        , const int LeakyReLu=0
        , const uint32_t negative_slope=0x3DCCCCCD
        >
void Linear1d
(           TYPE    *out_data
   , const TYPE    *in_data
   , const TYPE    *weight
   , const TYPE    *bias
   , const uint16_t  out_size
   , const uint16_t  in_size
   , const uint16_t  bias_size
   #if !defined(__SYNTHESIS__)
   , const int      rigor=0
   , const int      verbose=0
   #endif
);
```

Templates:

- ✧ DTYPE: data type of out_data, in_data, kernel, bias, out_data

- int, float, double, and so on[10].

- ReLu: add ReLU activation at the end of linear operation (it should be 0 when LeakyReLU is 1)
- LeakyReLu: Leaky-ReLU activation at the end of linear operation (it should be 0 when ReLU is 1)
- negative_slope: Slope to be applied for negative input with 0.001 by default and only valid when LeakyReLU is 1.

Arguments:

- out_data[out_size]: resultant 1D array
- in_data[in_size]: input feature 1D array
- weight[out_size][in_size]: weights
  - weight should be transposed before multiplying to in_data
- bias[out_size]: bias 1D array
- out_size: the number of elements of out_data[]
- in_size: the number of elements of in_data[]
- bias_size: the number of elements of bias[]
- rigor: check values rigorously when 1
- verbose: print more message when 1

Array size must keep following rules.

- ‘out_data’ and ‘in_data’ are 1 dimensional array.
- ‘weight’ is 2 dimensional array
- ‘weight’ should be ‘out_size x in_size’
- ‘bias’ should be 1 dimensional array
- ‘bias_size’ should be the same as ‘out_size’

8.1.1 Python wrapper

‘Linear1d()’ wrapper gets NumPy arguments for array and ‘out_data’ carries calculated result. It returns ‘True’ on success or ‘False’ on failure.

```
Linear1d( out_data    # out_size
          , in_data     # in_size
          , weight # out_size x in_size
          , bias # out_size
          , rigor=False
          , verbose=False)
```

8.1.2 PyTorch wrapper

‘linear()’ PyTorch wrapper gets PyTorch tensor arguments for array and returns calculated result. It calls Python wrapper after converting PyTorch tensor to NumPy array.

[10] Fixed-point data type will be supported.

```
linear( input # minibatch x 1 x in_size
          , weight # out_size x in_size
          , bias # out_size
          , rigor=False
          , verbose=False)
```

8.2 Linear ND

'LinearNd()' applies a N-D matrix multiplication over an input data.

```
#include "linear_nd.hpp"
```

```
template<class TYPE=float, int ReLu=0>
void LinearNd
(        TYPE    *out_data
   , const TYPE    *in_data
   , const TYPE    *weight
   , const TYPE    *bias
   , const uint16_t  out_size
   , const uint16_t  in_size
   , const uint16_t  bias_size
   , const uint8_t   ndim
   #if !defined(__SYNTHESIS__)
   , const int       rigor=0
   , const int       verbose=0
   #endif
);
```

Templates:

- DTYPE: data type of out_data, in_data, kernel, bias, out_data
    - int, float, double, and so on[11].
- ReLu: add ReLU activation at the end of linear operation

Arguments:

- out_data[ndim][out_size]: resultant 1D array
- in_data[ndim][in_size]: input feature 1D array
- weight[out_size][in_size]: weight
    - weight should be transposed before multiplying to in_data
- bias[out_size]: bias 1D array
- out_size: the number of elements of out_data[]
- in_size: the number of elements of in_data[]
- bias_size: the number of elements of bias[]
- ndim: dimension of in_data
- rigor: check values rigorously when 1
- verbose: print more message when 1

Array size must keep following rules.

- 'out_data' and 'in_data' are 2 dimensional array.

[11] Fixed-point data type will be supported.

- 'weight' is 2 dimensional array
- 'weight' should be 'out_size x in_size'
- 'bias' should be 1 dimensional array
- 'bias_size' should be the same as 'out_size'

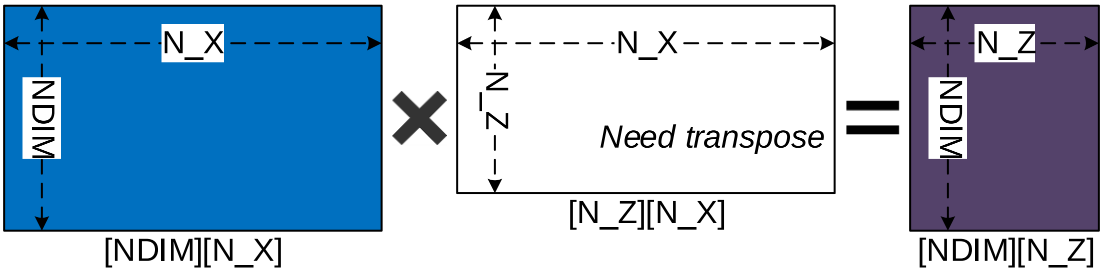

Figure 6: Matrix multiplication over 2 dimensional array case

8.2.1 Python wrapper

'LinearNd()' wrapper gets NumPy arguments for array and 'out_data' carries calculated result. It returns 'True' on success or 'False' on failure.

```
LinearNd( out_data    # ndim x out_size
        , in_data     # ndim x in_size
        , weight      # out_size x in_size
        , bias        # out_size
        , rigor=False
        , verbose=False)
```

8.2.2 PyTorch wrapper

'linearNd()' PyTorch wrapper gets PyTorch tensor arguments for array and returns calculated result. It calls Python wrapper after converting PyTorch tensor to NumPy array.

```
linearNd( input   # minibatch x ndim x in_size
        , weidht # out_size x in_size
        , bias    # out_size
        , rigor=False
        , verbose=False)
```

9 Normalization

9.1 Batch normalization 2D

'Batchnorm()' applies a batch normalization for each channel across a batch of data.

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta$$

where ‘y’ is outut, ‘x’ is input, ‘E[x]’ is mean, ‘Var[x]’ is variance, ‘gamma’ is scaling factor, ‘beta’ is shift factor (bias), ‘epsilon’ is value for numerical stability.

```
#include “norm_2d_batch.hpp”
```

```
template< class TYPE=float
            , int LeakyReLu=0
            , int negative_slope1000=100>
void Norm2dBatch
(           TYPE     *out_data
   , const TYPE     *in_data
   , const TYPE     *running_mean
   , const TYPE     *running_var
   , const TYPE     *scale
   , const TYPE     *bias
   , const uint32_t  in_size
   , const uint16_t  scale_size
   , const uint16_t  bias_size
   , const uint16_t  in_channel
   , const float     epsilon=1E-5
   #if !defined(__SYNTHESIS__)
   , const int       rigor=0
   , const int       verbose=0
   #endif
);
```

Templates:

- DTYPE: data type of out_data, in_data, running_mean, running var, scale and bias
  - int, float, double, and so on.
- LeakyReLu: apply Leaky ReLU at the end of this operation
- negative_slope1000: 100 means 0.1

Arguments:

- out_data[in_channel][in_size][in_size] resultant array in square
- in_data[in_channel][in_size][in_size] input feature array in square
- running_mean: running_mean[in_channel], mean of each channel
- running_var: running_var[in_channel], standard variance of each channel
- scale: scale[in_channel], scale factor (by default, 1)
- bias: bias[in_channel], shift factor (by default, 0)
- in_size: the number of elements in width direction (column)
- scale_size: the number of elements in scale[], it can be 0 or in_channel
- bais_sie: the number of elements in bias[], it can be 0 or in_channel
- in_channel: the number of input channels
- epsilon: numeric stability (by default, 1e-5)

Array size must keep following rules.

- 'out_data' and 'in_data' are the same size
- scale_size, bias_size can be 0 or in_channel.

9.1.1 Python wrapper

'Norm2dBatch()' wrapper gets NumPy arguments for array and 'out_data' carries calculated result. It returns 'True' on success or 'False' on failure.

```
Norm2dBatch( out_data    # out_size
             , in_data    # in_size
             , rigor=False
             , verbose=False)
```

9.1.2 PyTorch wrapper

'batch_norm()' PyTorch wrapper gets PyTorch tensor arguments for array and returns calculated result. It calls Python wrapper after converting PyTorch tensor to NumPy array.

```
batch_norm( input # minibatch x 1 x in_size
             , rigor=False
             , verbose=False)
```

10 Pooling

10.1 Pooling2dMax

'Pooling2dMax()' applies a 2D max pooling over an input data composed of several input planes (channels).

```
#include "pooling_2d_max.hpp"
```

```
template< class TYPE=float
        , const int ReLu=0
        , const int LeakyReLu=0
        , const uint32_t negative_slope=0x3DCCCCCD
        >
void Pooling2dMax
(           TYPE     *out_data
    , const TYPE     *in_data
    , const uint16_t  out_size
    , const uint16_t  in_size
    , const uint8_t   kernel_size
    , const uint16_t  channel
    , const uint8_t   stride
    , const uint8_t   padding=0
    , const int       ceil_mode=0
```

```
   #if !defined(__SYNTHESIS__)
   , const int      rigor=0
   , const int      verbose=0
   #endif
);
```

Templates:

- DTYPE: data type of out_data, in_data, kernel, bias, out_data
  - int, float, double, and so on[12].
- ReLU: apply ReLU at the end of this operation when 1
- LeakyReLu: apply leaky ReLU at the end of this operation when 1
- negative_slope: Slope to be applied for negative input and only valid when LeakyReLU is 1. 0.01 by default.

Arguments:

- out_data[channel][out_size][out_size] resultant array in square
- in_data[channel][in_size][in_size] input feature array in square
- out_size: the number of elements in width direction (column)
- in_size: the number of elements in width direction (column)
- kernel_size: the number of elements in width direction (column)
- channel: the number of input/output channels
- strid: stride of the convolution (one by default)
- padding: the number of zero-paddings on both sides (zero by default, not implemented yet)
- ceil_mode: use ceil() instead of floor to calculate out_size
- rigor: check values rigorously when 1
- verbose: print more message when 1

‘minibatch’ is not supported.

Array size must keep following rules.

- ‘out_data’, ‘in_data’, ‘kernel’ are square matrix
- ‘out_size’ = floor(((in_size-kernel_size+2*padding)/stride)+1) when ‘ceil_mode’ is 0 by default. Otherwise, ceil() is used instead.
- ‘kernel_size’ should be odd number
- ‘stride’ should be >=1
- ‘padding’ should be >=0 and will be floor(kernel_size/2) normal case

10.1.1 Python wrapper

‘Pooling2dMax()’ wrapper gets NumPy arguments for array and ‘out_data’ carries calculated result. It returns ‘True’ on success or ‘False’ on failure.

[12] Fixed-point data type will be supported.

```
Pooling2dMax( out_data    # out_channel x out_size x out_size
            , in_data     # in_channel x in_size x in_size
            , kernel_size
            , stride=1
            , padding=0
            , ceil_mode=False
            , rigor=False
            , verbose=False)
```

10.1.2 PyTorch wrapper

'max_pool2d()' PyTorch wrapper gets PyTorch tensor arguments for array and returns calculated result. It calls Python wrapper after converting PyTorch tensor to NumPy array.

```
max_pool2d( input # minibatch x in_channel x in_size x in_size
          , kernel_size
          , stride=1
          , padding=0
          , ceil_mode=False
          , groups=1
          , rigor=False
          , verbose=False)
```

10.2 Pooling2dAvg

'Pooling2dAvg()' applies a 2D average pooling over an input data composed of several input planes (channels).

```
#include "pooling_2d_avg.hpp"
```

```
template<class TYPE=float>
void Pooling2dAvg
(          TYPE    *out_data
  , const TYPE    *in_data
  , const uint16_t  out_size
  , const uint16_t  in_size
  , const uint8_t   kernel_size
  , const uint16_t  channel
  , const uint8_t   stride
  , const uint8_t   padding=0
  , const int      ceil_mode=0
  #if !defined(__SYNTHESIS__)
  , const int      rigor=0
  , const int      verbose=0
  #endif
  )
```

Templates:

- ✧ DTYPE: data type of out_data, in_data, kernel, bias, out_data

  - int, float, double, and so on[13].

Arguments:

- out_data[channel][out_size][out_size] resultant array in square
- in_data[channel][in_size][in_size] input feature array in square
- out_size: the number of elements in width direction (column)
- in_size: the number of elements in width direction (column)
- kernel_size: the number of elements in width direction (column)
- channel: the number of input/output channels
- strid: stride of the convolution (one by default)
- padding: the number of zero-paddings on both sides (zero by default, not implemented yet)
- ceil_mode: use ceil() instead of floor to calculate out_size
- rigor: check values rigorously when 1
- verbose: print more message when 1

‘minibatch’ is not supported.

Array size must keep following rules.

- ‘out_data’, ‘in_data’, ‘kernel’ are square matrix
- ‘out_size’ = floor(((in_size-kernel_size+2*padding)/stride)+1) when ‘ceil_mode’ is 0 by default. Otherwise, ceil() is used instead.
- ‘kernel_size’ should be odd number
- ‘stride’ should be >=1
- ‘padding’ should be >=0 and will be floor(kernel_size/2) normal case

10.2.1 Python wrapper

‘Pooling2dAvg()’ wrapper gets NumPy arguments for array and ‘out_data’ carries calculated result. It returns ‘True’ on success or ‘False’ on failure.

```
Pooling2dAvg( out_data    # out_channel x out_size x out_size
            , in_data     # in_channel x in_size x in_size
            , kernel_size
            , stride=1
            , padding=0
            , ceil_mode=False
            , rigor=False
            , verbose=False)
```

10.2.2 PyTorch wrapper

‘avg_pool2d()’ PyTorch wrapper gets PyTorch tensor arguments for array and returns calculated result. It calls Python wrapper after converting PyTorch tensor to NumPy array.

[13] Fixed-point data type will be supported.

```
avg_pool2d( input # minibatch x in_channel x in_size x in_size
          , kernel_size
          , stride=1
          , padding=0
          , ceil_mode=False
          , groups=1
          , rigor=False
          , verbose=False)
```

11 Project: LeNet-5

11.1 LeNet-5

LeNet-5 is a convolutional network designed for handwritten and machine-printed character recognition. Figure 7 shows network and its data size. Table 1 shows a summary of the network and its parameters.

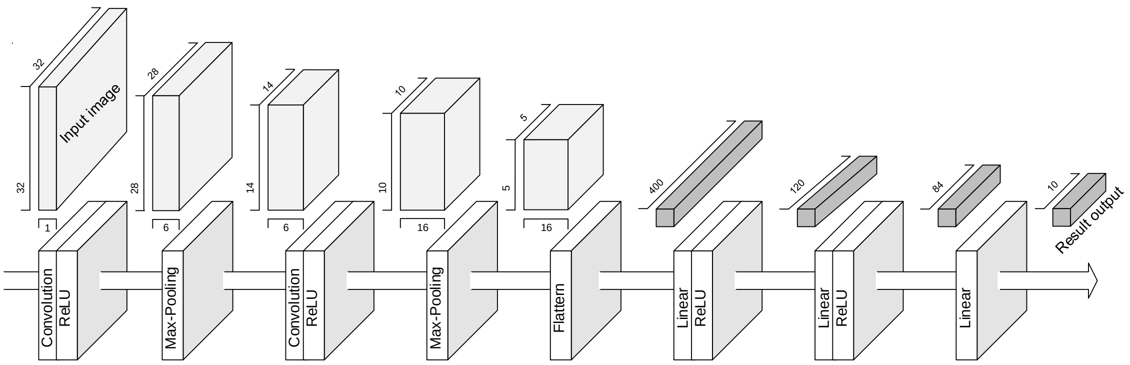

Figure 7: LeNet-5 network

Table 1: LeNet-5 summary

Layer		feature map (channel x W x H)		kernel size	stride	padding	Activation	Parameters (bias+weight)	Feature map
input	image	1	32x32	.	.	.	.	.	1,024
1	CONV	6	28x28	5x5	1	0	ReLU	6+150	4,704
2	MaxPOOL	6	14x14	2x2	2	0	.	0	1,176
3	CONV	16	10x10	5x5	1	0	ReLU	16+2,400	1,600
4	MaxPOOL	16	5x5	2x2	2	0	.	0	400
5	FC	120	1x1	5x5	1	0	ReLU	120+48,000	120
6	FC	.	84	.	.	.	ReLU	84+10,080	84
7	FC	.	10	.	.	.	softmax	0	10
								61,706	9,118

11.1.1 LeNet-5 C++ version

Following code shows an example implementation of LeNet-5 network model using DLR deep learning routines, where ‘lenet5_params.h’ should contain trained results of weights and biases. Flatten step before the first linear layer is not required for DLR since its data array can be seen linear array.

```
#if !defined(DTYPE)
#define DTYPE float
#endif

#include "dlr.hpp"
#include "lenet5_params.h"  // come from LeNet-5.pytorch

void lenet5(       DTYPE classes[10]
          , const DTYPE image  [32][32]
          #if !defined(__SYNTHESIS__)
          , const int rigor
          , const int verbose
          #endif
          )
{
             DTYPE     c1_out_data[6][28][28];
    const DTYPE   (*c1_in_data)[32][32]=(DTYPE (*)[32][32])image; // [1][32][32]
    const DTYPE   (*c1_kernel)[1][5][5]=(DTYPE (*)[1][5][5])conv1_weight; // [6][1][5][5]
    const DTYPE   (*c1_bias)=conv1_bias; // [6]
    const uint16_t  c1_out_size=28;
    const uint16_t  c1_in_size=32;
    const uint8_t   c1_kernel_size=5;
    const uint16_t  c1_bias_size=6;
    const uint16_t  c1_in_channel=1;
    const uint16_t  c1_out_channel=6;
    const uint8_t   c1_stride=1;
    const uint8_t   c1_padding=0;

    Convolution2d<DTYPE> (
          (DTYPE *)c1_out_data
        , (DTYPE *)c1_in_data
        , (DTYPE *)c1_kernel
        , (DTYPE *)c1_bias
        ,          c1_out_size
        ,          c1_in_size
        ,          c1_kernel_size
        ,          c1_bias_size
        ,          c1_in_channel
        ,          c1_out_channel
        ,          c1_stride
        ,          c1_padding
        #if !defined(__SYNTHESIS__)
        ,          rigor
        ,          verbose
        #endif
    );

          DTYPE     p1_out_data[6][14][14];
    const DTYPE   (*p1_in_data)[28][28]=c1_out_data; // [6][28][28]
    const uint16_t  p1_out_size=14;
    const uint16_t  p1_in_size=28;
    const uint8_t   p1_kernel_size=2;
    const uint8_t   p1_channel=6;
    const uint8_t   p1_stride=2;
    const uint8_t   p1_padding=0;
    const int       p1_ceil_mode=0;

    Pooling2dMax<DTYPE, 1> ( // ReLU embedded
          (DTYPE *)p1_out_data    // out_channel x out_size x out_size
        , (DTYPE *)p1_in_data     // in_channel x in_size x in_size
```

```
    ,       p1_out_size    // only for square matrix
    ,       p1_in_size     // only for square matrix
    ,       p1_kernel_size // only for square matrix
    ,       p1_channel
    ,       p1_stride
    ,       p1_padding
    ,       p1_ceil_mode
    #if !defined(__SYNTHESIS__)
    ,       rigor
    ,       verbose
    #endif
);

    DTYPE     c2_out_data[16][10][10];
const DTYPE   (*c2_in_data)[14][14]=p1_out_data; // [6][14][14]
const DTYPE   (*c2_kernel)[6][5][5]=(DTYPE (*)[6][5][5])conv2_weight; // [16][6][5][5]
const DTYPE   (*c2_bias)=conv2_bias; // [16]
const uint16_t  c2_out_size=10;
const uint16_t  c2_in_size=14;
const uint8_t   c2_kernel_size=5;
const uint16_t  c2_bias_size=16;
const uint16_t  c2_in_channel=6;
const uint16_t  c2_out_channel=16;
const uint8_t   c2_stride=1;
const uint8_t   c2_padding=0;

Convolution2d<DTYPE> (
      (DTYPE *)c2_out_data
    , (DTYPE *)c2_in_data
    , (DTYPE *)c2_kernel
    , (DTYPE *)c2_bias
    ,       c2_out_size
    ,       c2_in_size
    ,       c2_kernel_size
    ,       c2_bias_size
    ,       c2_in_channel
    ,       c2_out_channel
    ,       c2_stride
    ,       c2_padding
    #if !defined(__SYNTHESIS__)
    ,       rigor
    ,       verbose
    #endif
);

    DTYPE     p2_out_data[16][5][5];
const DTYPE   (*p2_in_data)[10][10]=c2_out_data; // [16][10][10]
const uint16_t  p2_out_size=5;
const uint16_t  p2_in_size=10;
const uint8_t   p2_kernel_size=2;
const uint8_t   p2_channel=16;
const uint8_t   p2_stride=2;
const uint8_t   p2_padding=0;
const int       p2_ceil_mode=0;

Pooling2dMax<DTYPE, 1> ( // ReLU embedded
      (DTYPE *)p2_out_data
    , (DTYPE *)p2_in_data
    ,       p2_out_size
    ,       p2_in_size
    ,       p2_kernel_size
    ,       p2_channel
    ,       p2_stride
    ,       p2_padding
```

```
    ,       p2_ceil_mode
    #if !defined(__SYNTHESIS__)
    ,       rigor
    ,       verbose
    #endif
);

    DTYPE     f1_out_data[120];
const DTYPE   (*f1_in_data)[5][5]=p2_out_data; // [16][5][5]
const DTYPE   (*f1_weight)[400]=(DTYPE (*)[400])fc1_weight; // [120][400]
const DTYPE   (*f1_bias)=fc1_bias; // [120]
const uint16_t  f1_out_size=120;
const uint16_t  f1_in_size=16*5*5;
const uint16_t  f1_bias_size=120;

Linear1d<DTYPE, 1> ( // ReLU embedded
      (DTYPE *)f1_out_data
    , (DTYPE *)f1_in_data
    , (DTYPE *)f1_weight
    , (DTYPE *)f1_bias
    ,       f1_out_size
    ,       f1_in_size
    ,       f1_bias_size
    #if !defined(__SYNTHESIS__)
    ,       rigor
    ,       verbose
    #endif
);

    DTYPE     f2_out_data[84];
const DTYPE   (*f2_in_data)=f1_out_data; // [120]
const DTYPE   (*f2_weight)[120]=(DTYPE (*)[120])fc2_weight; // [84][120]
const DTYPE   (*f2_bias)=fc2_bias; // [84]
const uint16_t  f2_out_size=84;
const uint16_t  f2_in_size=120;
const uint16_t  f2_bias_size=84;

Linear1d<DTYPE, 1> ( // ReLU embedded
      (DTYPE *)f2_out_data
    , (DTYPE *)f2_in_data
    , (DTYPE *)f2_weight
    , (DTYPE *)f2_bias
    ,       f2_out_size
    ,       f2_in_size
    ,       f2_bias_size
    #if !defined(__SYNTHESIS__)
    ,       rigor
    ,       verbose
    #endif
);

    DTYPE   (*f3_out_data)=classes; // [10]
const DTYPE   (*f3_in_data)=f2_out_data; // [84]
const DTYPE   (*f3_weight)[84]=(DTYPE (*)[84])fc3_weight; // [10][84]
const DTYPE   (*f3_bias)=fc3_bias; // [10]
const uint16_t  f3_out_size=10;
const uint16_t  f3_in_size=84;
const uint16_t  f3_bias_size=10;

Linear1d<DTYPE, 0> ( // ReLU not embedded
      (DTYPE *)f3_out_data
    , (DTYPE *)f3_in_data
    , (DTYPE *)f3_weight
    , (DTYPE *)f3_bias
```

```
        ,       f3_out_size
        ,       f3_in_size
        ,       f3_bias_size
        #if !defined(__SYNTHESIS__)
        ,       rigor
        ,       verbose
        #endif
    );
}
```

11.1.2 LeNet-5 C PyTorch version

Following code shows an example of LeNet-5 network using DLR PyTorch wrapper functions.

```
import torch
import python.torch as dlr

def lenet5_infer( input
            , c1_kernel, c1_bias
            , c2_kernel, c2_bias
            , f1_weight, f1_bias
            , f2_weight, f2_bias
            , f3_weight, f3_bias
            , softmax=True
            , pkg=dlr
            , rigor=False):
    """ LeNet-5 inference network """
    z = pkg.conv2d( input=input, weight=c1_kernel, bias=c1_bias, stride=1
            , padding=0, dilation=1, groups=1, rigor=rigor )
    z = pkg.relu( input=z, rigor=rigor )
    z = pkg.max_pool2d( input=z, kernel_size=2, stride=2, padding=0,
                ceil_mode=False, rigor=rigor )

    z = pkg.conv2d( input=z, weight=c2_kernel, bias=c2_bias, stride=1
            , padding=0, dilation=1, groups=1, rigor=rigor )
    z = pkg.relu( input=z, rigor=rigor )
    z = pkg.max_pool2d( input=z, kernel_size=2, stride=2, padding=0,
                ceil_mode=False, rigor=rigor )

    z = z.view(z.shape[0], -1)
    z = pkg.linear( input=z, weight=f1_weight, bias=f1_bias, rigor=rigor )
    z = pkg.relu( input=z, rigor=rigor )
    z = pkg.linear( input=z, weight=f2_weight, bias=f2_bias, rigor=rigor )
    z = pkg.relu( input=z, rigor=rigor )
    z = pkg.linear( input=z, weight=f3_weight, bias=f3_bias, rigor=rigor )
    if softmax: z = F.softmax( input=z, dim=1)
    else: z = pkg.relu( input=z )
    return z
```

12 Troubleshooting

To be added.

13 Acknowledgement

This work was supported partially by KARI[14] (Korea Aerospace Research Institute) under the "Development of deep learning hardware accelerator for microsatellite[15]" (Contract 2019110A35C-00).

Some part of this work was carried out partially by Handong Global University[16] 2020 Summer Internship program.

Some part of this work was supported partially by ETRI[17] (Electronics and Telecommunications Research Institute) under the "Development and training of FPGA-based AI semiconductor design environment[18]" (Contract EA20202206).

14 References

[1] Future Design Systems, CON-FMC User Manual, FDS-TD-2018-03-001, 2018.
[2] NumPy, https://numpy.org
[3] Ctypes., https://docs.python.org/3/library/ctypes.html
[4] PyTorch, https://pytorch.org

Wish list

- ☐ Sofmax, upsampling

Index

---

[14] 한국항공우주연구원, https://www.kari.re.kr

[15] 초소형위성용 딥러닝 알고리즘 하드웨어 가속기 개발

[16] 한동대학교, https://www.handong.edu, Chaeeon Lim, Yoonseong Lim, Geunsu Song.

[17] 전자통신연구원, https://www.etri.re.kr

[18] FPGA 기반 AI 반도체 설계환경 개발 및 교육

Revision history

- ☐ 2020.12.17: Minor correction
- ☐ 2020.11.12: License scheme changed from proprietary to open.
- ☐ 2020.10.20: ‘channel’ argument added for activation functions
- ☐ 2020.10.01: Major update.
- ☐ 2020.09.01: First version released.
- ☐ 2020.08.31: More IP have been added.
- ☐ 2020.06.23: Initial version released
- ☐ 2020.03.10: Started by Ando Ki (adki@future-ds.com)

– End of document –