可综合 Deep Learning 库设计实现

金济芳 jfjin14@fudan.edu.cn 复旦大学

1. 开发环境

1.1. Deep Learning 训练

Keras: keras 是基于 Python 的 Deep Learning 开源库，涵盖了各种组件。本设计中以 keras 为参考基准，使用 keras 设计模型，并训练网络各层参数。

1.2. 可综合库设计工具

High Level Synthesis: Xilinx HLS 工具作为设计工具，用于代码编写，功能仿真，资源与时序分析等。

SDSoC: 在 HLS 功能验证通过后，使用 SDSoC 进行实际的位流验证。

2. 可综合 Deep Learning 库组件设计

2.1. 数据类型

考虑到用户自定义网络对数据的精度需求，比如分类只要求分类结果正确而不要求每个输出神经元的值非常精确，但是如果使用网络做多项式拟合时精度要求较高。其次，在 HLS 中，数据精度对硬件资源的占用率有很大影响。比如一个 DSP 最多支持 17-bit 的运算， 因此 2 个 float 类型的数据乘法需要 3 个 DSP。HLS 中支持可配置的定点数运算，在满足精度前提下，使用定点数可以降低硬件资源消耗。

因此，库中所有的数据类型定义为 TYPE_T， 可由用户自行 typedef 定义,具体在 configure.h 文件中定义。

所有代码使用 C++模板方法实现，并定义在 **namespace SDAI** 命名空间中。

2.2. 性能配置

HLS 处理多重循环时，需要考虑性能和硬件资源之间的均衡关系。

```
LOOP_1:for(int i = 0;....)
{
    LOOP_2:for(int j = 0;....)
    {
#pragma HLS pipeline
        LOOP_3:for(int k = 0;,,,)
        {
            ......
            ......
        }
    }
}
```

对于循环，一般使用流水#pragma HLS pipeline 进行优化，**该优化指令的特点为:对当前循环层进行 Pipeline，里层的循环层自动被优化展开，外层的循环层无影响**。比如上图中的 LOOP_2 中，假设使用#pragma HLS pipeline，那么 LOOP_3 会被自动优化，一般都是 UNROLL，对 LOOP_1 无作用。

因此，一般而言，将#pragma HLS pipeline 指令运用到越外层，性能越好；越运用到内层，性能越差。当然也可能越到外层性能反而变差的情况。

在各层设计中，各层的循环中均使用了#pragma HLS pipeline 指令对循环进行优化，但是优化的程度都是可配置的，由<LAYER_NAME>_PERF_MODE 宏决定。这些宏统一定义在 configure.h 文件中。优化程度比较粗超的分为 PERF_HIGH, PERF_MEDIAN 和 PERF_LOW 三种，默认所有层的优化程度都是 PERF_HIGH。用户使用时可自行配置。

```
/*
 * @note: configure the performance for each layer, it can be configured as PERF_HIGH, PERF_MEDIAN and PERF_LOW
 */
#define PERF_HIGH                           3
#define PERF_MEDIAN                         2
#define PERF_LOW                            1

#define DENSE_PERF_MODE                     PERF_HIGH
#define CONVOLUTION1D_PERF_MODE             PERF_HIGH
#define CONVOLUTION2D_PERF_MODE             PERF_HIGH
#define EMBEDDING_PERF_MODE                 PERF_HIGH
#define POOLING1D_PERF_MODE                 PERF_HIGH
#define POOLING2D_PERF_MODE                 PERF_HIGH
#define RECURRENT_PERF_MODE                 PERF_MEDIAN
#define RESHAPE_PERF_MODE                   PERF_HIGH
#define UTILS_PERF_MODE                     PERF_HIGH
```

2.3. 存储模型

对于小规模深度学习网络，比如层数较少且每层复杂度较低时，所有的权重和各层的计算中间结果可以缓存到 FPGA 内部的存储资源中，比如 BRAM, Flip-Flops 等。但是对于大型深度学习网络，其权重和各层的计算中间结果规模急剧增大，难以全部缓存到 FPGA 内部存储资源中。基于以上考虑，该可综合 Deep Learning 库设计了两种存储模型。

1) Local 模型: 适用于小型层，将当前层的权重和计算输出结果均暂存到 FPGA 的 BRAM 中。
2) Stream 模型：适用于大型层，将当前层的权重或者计算结果，通过 AXI Master 总线传输。AXI Master 总线可以与 DDR, Flash 等 FPGA 外部存储器连接，通过总线将数据存储到 FPGA 片外。
3) 混合模型：根据数据容量，将部分数据通过 Local 模型缓存到 BRAM 中；部分数据通过 Stream 模型缓存到片外。

对于 Stream 模型，根据各层的特点，可以选择性的将大容量数据缓存到 FPGA 片外。比如较大全连接层，一般其权重数量远大于输入输出数量，因此可以将权重通过 Stream 模型缓存到 FPGA 片外，但是输入输出可以通过 Local 模型缓存到 FPGA BRAM 中。

2.4. 存储优化配置

对于 Stream 模型，使用 AXI Master 总线从 FPGA 片外访问数据，但是 AXI Master 访问速度和带宽有限，容易成为网络性能的瓶颈。基于以上考虑，该可综合 Deep Learning 库设计了三种存储优化模式。

1) OPT_NONE: 无优化。
2) OPT_MEM：局部存储。根据各层对数据访问的特点和范围，一次将部分连续数据通过 AXI Master 总线的 Burst 模式，复制到 FPGA 内部的 BRAM 中。访问 BRAM 速度快于 AXI Master 总线访问， 而且 BRAM 容易被 HLS 编译器自动优化。
3) OPT_BUFFER：包括 LineBuffer 和 WindowBuffer。构造线性缓冲器和窗口缓冲

器，实现对多维数组的优化。

一般而言，对于多维数据和大容量数据，OPT_BUFFER 优化性能较好，但是编译的复杂度也更高，会导致较差的时序和更高的资源利用率。用户可在 configure.h 中自行配置优化方法。

```
#define CONVOLUTION1D_OPT_MODE                  OPT_MEM
#define POOLING1D_OPT_MODE                      OPT_NONE
#define CONVOLUTION2D_OPT_MODE                  OPT_MEM
#define POOLING2D_OPT_MODE                      OPT_MEM
```

2.5. ACTIVATION

- 位置: activation.h
- 日期: 2016/10/8
- 描述：包含一系列常用的激活函数，目前已定义如下:

typedef enum{*LINEAR*, *SIGMOID*, *TANH*, *RELU*, *SOFTSIGN*, *SOFTPLUS*, *SOFTMAX*}ACTIVATION;

激活函数	表达式
LINEAR	$y = x$
SIGMOID	$y = \frac{1}{1+e^{-x}}$
HARDSIGMOID	$y = \mathrm{MAX}(0, \mathrm{MIN}(1, \quad 0.2x + 0.5))$
TANH	$y = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} = 1 - \frac{2}{e^{2x}+1}$
RELU	$y = \max(0, x)$
LEAKYRELU	$y = x \geq 0 \ ? \ x : \mathrm{alpha} * x$
THRESHOLDEDRELU	$y = x \geq \mathrm{theta} \ ? \ x : 0$
SOFTSIGN	$y = \frac{1}{1+\lvert x\rvert}$
SOFTPLUS	$y = \log(1+e^{x})$
SOFTMAX	$\max = MAX(x_i)$ $v_i = exp^{x_i - max}$ $y_i = \frac{v_i}{\sum_i^N v_i}$

2.5.1. 激活函数

- **inline** TYPE_T **activation_fn**(TYPE_T x)

  activation_fn 是顶层的激活函数，根据模板 AC_FN 选择对应的激活函数进行运算。其中 SOFTMAX 比较特殊，它的参数是一维数组， 处理方式略有不同。详细可参照 dense 层中 SOFTMAX 运算过程。

  此外，用户可以自定义各种激活函数， 然后放到 activation_fn 中使用。

2.6. Dense

- 位置: dense.h

- 日期: 2016/10/8
- 描述：实现了全连接层 Dense，即输入与输出神经元之间的全连接。

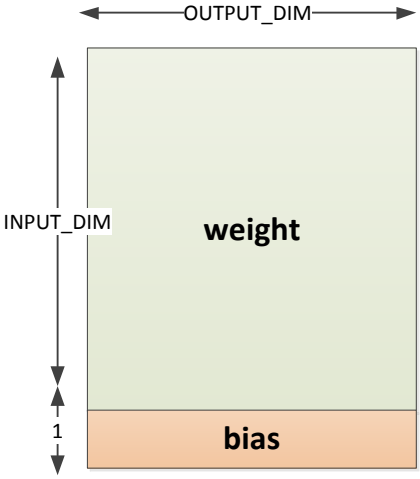
2.6.1. 模板参数

template<int INPUT_DIM, int OUTPUT_DIM, ACTIVATION AC_FN>
class Dense

- INPUT_DIM：输入神经元节点数目
- OUTPUT_DIM：输出神经元节点数目
- AC_FN：输出神经元的激活函数

2.6.2. 成员变量

- TYPE_T weight[**INPUT_DIM** + 1][**OUTPUT_DIM**];
  全连接层 Dense 输入与输出神经元之间的权重(weight)和偏置(bias)，是一个二维数组。权重(weight)大小是 INPUT_DIM x OUTPUT_DIM，偏置(bias)大小是 1 x OUTPUT_DIM，存储方式与 Keras 中 Dense 一致。



- TYPE_T res[**OUTPUT_DIM**];
  全连接层 Dense 输出神经元，经过激活函数后的计算结果，数组大小等于输出神经元的数目 OUTPUT_DIM.

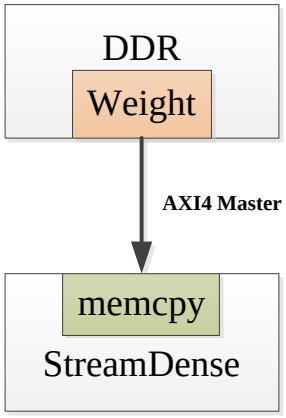
2.6.3. 成员函数

- **void feedforward**(TYPE_T data[**INPUT_DIM**])
  Dense 的推断函数，data 是输入样本数据，长度等于 Dense 的输入神经元数目 INPUT_DIM。经过激活函数后，运算结果保存到 res 中。

2.7. Dense_WeightStream

- 位置: dense.h
- 日期: 2016/11/10
- 描述：全连接层(Fully Connected Layer)最大的特点是，当输入神经元 INPUT_DIM 与输出神经元值 OUTPUT_DIM 都很大时，将会导致权重参数(**INPUT_DIM** + 1)x**OUTPUT_DIM** 非常大。在 Dense Layer 中，所有的 weight 都存储在 FPGA 的 Distributed RAM 中，但是当权重参数数目很大时，FPGA 中的存储资源 BRAM 会成为瓶颈。

Dense_WeightStream Layer 的设计思路是将数目很大的权重放到 FPGA 片外，比如 DDR、Flash 或者其他用户提供的接口。在执行 feedforward 推断函数时，再传入权重参数，而不是一开始就进行初始化。即 Weight 采用 Stream 模式，而输出中间结果采用 Local 模式。



Dense_WeightStream 采用了 AXI4 Mater Interface 处理大权重问题。在 Deep Learning 的各组件中，一般权重的访问都不是顺序的，而是随机访问的，因此 Weight 不能使用 AXI4 Stream 接口，但是 AXI4 Master 是共享内存模型，可实现硬件加速器对 Memory 的随机访问。此外，频繁的 AXI4 Master 的数据传输开销会极大制约性能，因此可使用 memcpy 函数实现小批量的 Burst 模式的数据传输，将 Weight 的小部分连续数据复制到 FPGA 的 BRAM 中，以减少数据传输开销。

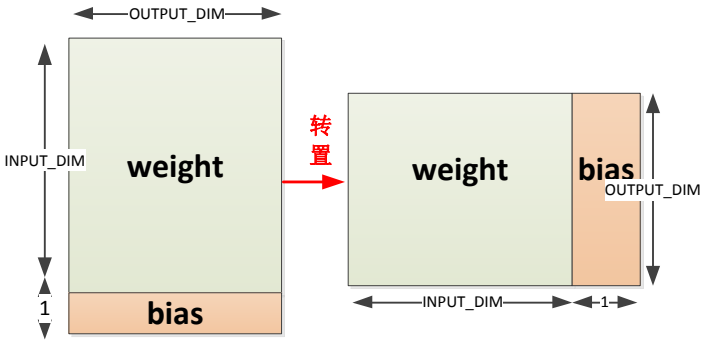
2.7.1. 模板参数

StreamDense 的模板参数与 Dense Layer 一致。

2.7.2. 成员函数

- **void feedforward(volatile** TYPE_T *weight, TYPE_T data[**INPUT_DIM**])

  Dense_WeightStream 的推断函数,，data 是输入数据，weight 是 AXI Master 接口的权重参数， **但是 weight 是 Keras 或者 Dense Layer 中的转置矩阵**。因此用户在定义权重时，需要先自行将其转置，在传入到 StreamDense 中。

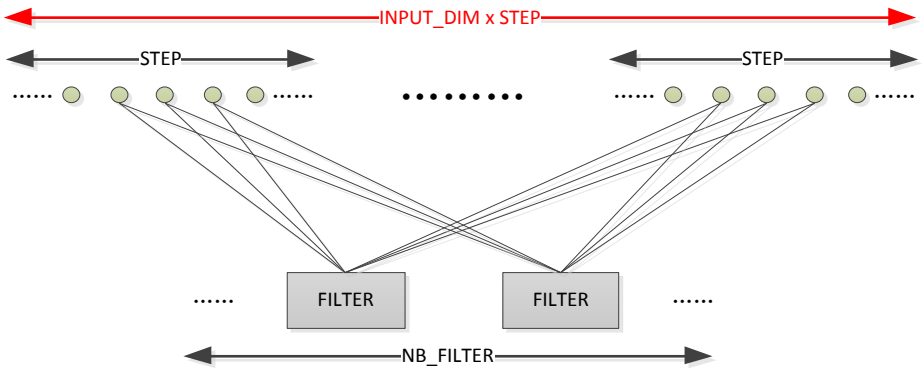


转置的目的是为了实现内层循环中 weight 的顺序访问，因此此时 weight 相当

于一维数组，而不是二维数组。当内存循环中 weight 可顺序访问时，使用 memcpy 函数将连续的数据从 AXI4 Master 接口复制到 FPGA 中的 RAM 中，一方面可以减少 AXI4 Master 数据传输的延时，另一方面使用 FPGA 内的 RAM 有利于编译器 Pipeline 指令的优化.

2.8. Convolution1D

- 位置: convolution1D.h
- 日期: 2016/10/10
- 描述： convolution1D 是实现一维数据序列的卷积层，其基本工作原理如下:



Convolutio1D 可包含多个卷积核(Filter)，数据序列到每个 Filter 的权重(weight)和偏置(bias)都是共享的，因此可以大幅降低权重参数的数目。

2.8.1. 模板参数

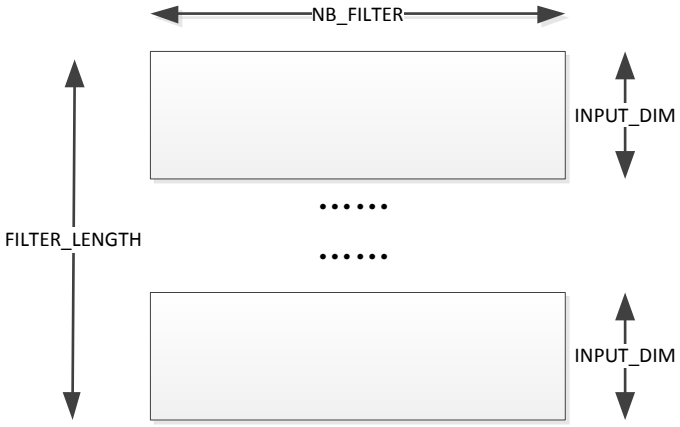
- NB_FILTER: 卷积核的数目。
- FILTER_LENGTH: 卷积核的长度，比如上图中卷积核的长度为 3。
- STEP: 输入的一维数据序列的步长。
- INPUT_DIM：输入数据序列的维数，即数据序列的数目，默认为 1。
- SUBSAMPLE_LENGTH：在输入数据序列上，卷积器每次移动的距离，默认是 1.
- AC_FN：卷积核输出的激活函数，默认值是 LINEAR。
- OUTPUT_DIM：每个卷积核 Filter 输出的维度。这个参数不需要用户输入，它是根据前面几个参数计算得到的。

$$\text{OUTPUT\_DIM} = ((\text{STEP} - \text{FILTER\_LENGTH})/\text{SUBSAMPLE\_LENGTH} + 1)$$

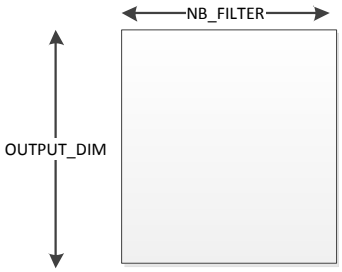
比如当输入数据长度 STEP=10，卷积核长度 FILTER_LENGTH=3，SUBSAMPLE_LENGTH=1 时， 那么每个卷积器会有 OUTPUT_DIM=(10 - 3)/1 + 1=8 个输出。

2.8.2. 成员变量

- TYPE_T weight[**FILTER_LENGTH**][**INPUT_DIM**][**NB_FILTER**];
  Convolution1D 的权重是一个 FILTER_LENGTH x INPUT_DIM x NB_FILTER 的三维数组。



- TYPE_T bias[**NB_FILTER**];
  对应各卷积核的偏置，一维数组，大小为 NB_FILTER.
- TYPE_T res[**OUTPUT_DIM**][**NB_FILTER**];
  Convolution1D 的输出结果是个二维矩阵，因为每个卷积核 Filter 会输出 OUTPUT_DIM 个数据，故 res 是一个 OUTPUT_DIM 行，NB_FILTER 列的二维数组。



2.8.3. 成员函数

- **void feedforward(TYPE_T data[STEP][INPUT_DIM])**
  输入数据序列 data 是长度为 STEP x INPUT_DIM 的二维数组，计算结果保存到 res 二维数组中。

2.9. Convolution1D_DataStream

- 位置: convolution1D.h
- 日期: 2016/11/15
- 描述：Convolution1D_DataStream 原理描述与 Convolution1D 相同。对于卷积层，其输入与输出数据一般是多维的，数据量较大；其权重是采用共享权重的，权重数据量反而较少。因此 Convolution1D_DataStream 将权重使用了 Local 模型，而输入输出使用 Stream 模型。

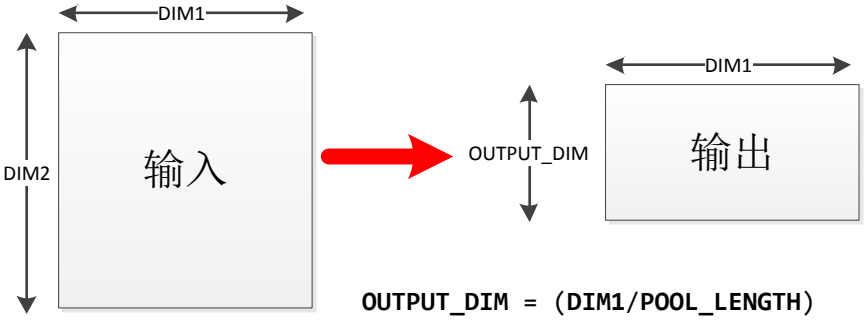
2.10. MaxPooling1D

- 位置: pooling1D.h
- 日期: 2016/10/10
- 描述：实现一维数据的 pooling，采用局部最大值法，一般在 Convolution1D 层后使用。从数据流来看，输入与输出数据都是二维数据，但是 MaxPooling1D 对数据的第一维进行了缩减。

2.10.1. 模板参数

- POOL_LENGTH：pooling 的步长，一般为 2。
- DIM1：输入数据的第一维大小。
- DIM2：输入数据的第二维大小。
- OUTPUT_DIM：输出数据的第一维大小。该参数不需要用户指定，由前面参数计算出。

$$\mathrm{OUTPUT\_DIM} = (\mathrm{DIM1}/\mathrm{POOL\_LENGTH})$$



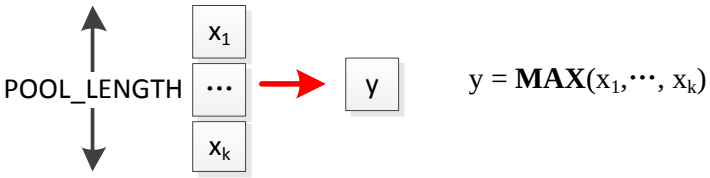
2.10.2. 成员变量

Maxpooling1D 层不需要权重和偏置等参数，但是存在计算过程。

- TYPE_T res[**OUTPUT_DIM**][**DIM2**];
  前面提到，输入数据大小为 DIM1 x DIM2，经过 pooling 后，输出变为 OUTPUT_DIM x DIM2，第一维大小被缩减了。

2.10.3. 成员函数

- **void feedforward**(TYPE_T data[**DIM1**][**DIM2**])
  在第一维数据中，取步长 POOL_LENGTH 个数据中的最大值作为输出结果。



2.11. MaxPooling1D_Stream

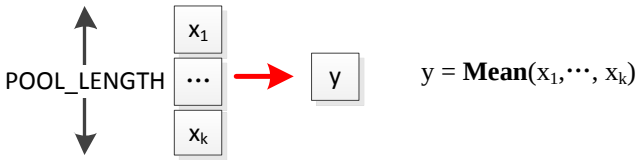
- 位置: pooling1D.h
- 日期: 2016/11/15
- 描述：与 MaxPooling1D 原理相同，只是输入输出采用了 Stream 模式。

2.12. AveragePooling1D

- 位置：pooling1D.h
- 日期: 2016/10/10
- 描述：原理与 MaxPooling1D 类似，一般在 Convolution1D 之后，对数据的第一维根据平均法进行缩减。其成员变量、模板参数与 MaxPooling1D 相同。

2.12.1. 成员函数

- **void feedforward**(TYPE_T data[**DIM1**][**DIM2**])
  在第一维数据中，取步长 POOL_LENGTH 个数据的平均值作为输出结果。



2.13. AveragePooling1D_Stream

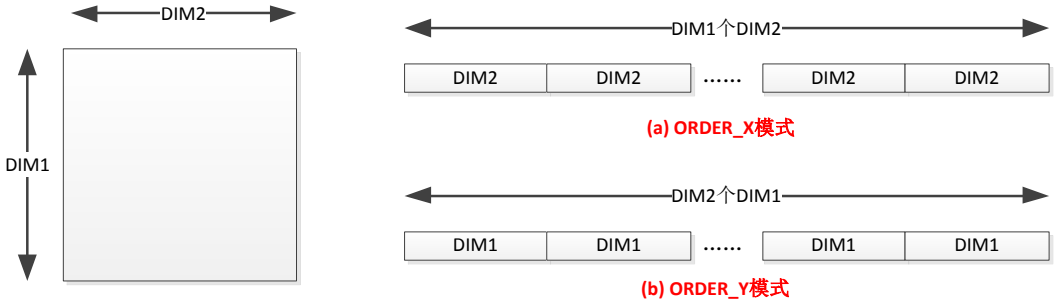
- 位置：pooling1D.h
- 日期: 2016/11/15
- 描述：原理与 AveragePooling1D 相同，输入输出使用 Stream 模式。

2.14. Reshape

- 位置：reshape.h
- 日期:2016/10/13
- 描述：实现各层之间中间结果维数的转换，比如 2D 到 1D 的展开，1D 到 3D 的扩展等。

2.14.1. Reshape2D_1D

- 描述：在设计中，在数组维数需要改变时，可以采用 Reshape 进行适当的转换。Reshape2D_1D 能够实现 2D 数组到 1D 数组的转换。
- 方法：转换模式包括 ORDER_X 和 ORDER_Y 两种方式。
  1) ORDER_X：以水平方向，也就是第二维 DIM2 方向展开。展开后的形式，输出与输入是相同的，因此这也是默认的展开方式。
  2) ORDER_Y：以竖直方向， 也就是第一维 DIM1 方向展开。展开后的形式相当于矩阵的转置。



2.14.2. Reshape3D_1D

- 描述: 实现 3 维数组到 1 维数组的转换。

2.14.3. Reshape_Stream_1D

- 描述：实现 AXI Master 总线到 1 维数组的转换。

2.14.4. Reshape_Stream_2D

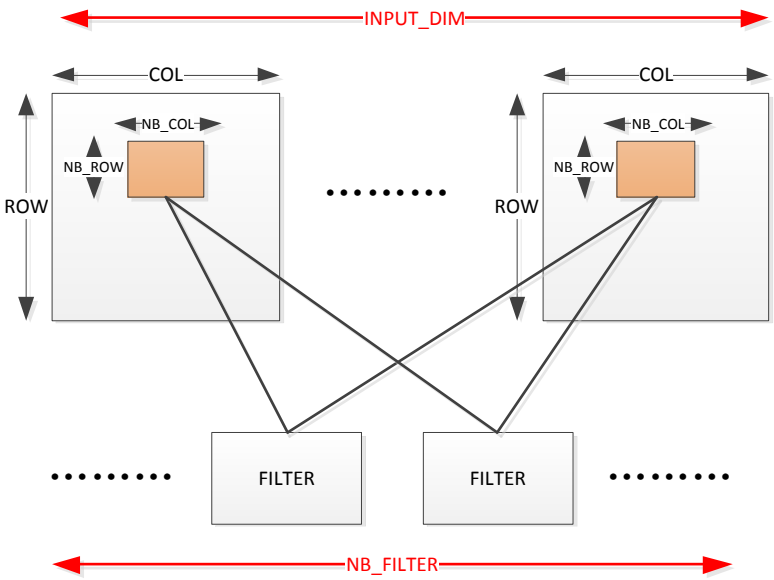
- 描述：实现 AXI Master 总线到 2 维数组的转换。

2.14.5. Reshape_Stream_3D

- 描述：实现 AXI Master 总线到 3 维数组的转换。

2.15. Convolution2D

- 位置:convolution2D.h
- 日期：2016/10/18
- 描述：Convolution2D 是二维数据矩阵的卷积层，其基本原理如下图所示。二维矩阵的大小为 ROW x COL， 输入的维数是 INPUT_DIM。卷积窗口的大小为 NB_ROW x NB_COL。卷积器 Filter 的数目为 NB_FILTER。



2.15.1. 模板参数

- NB_FILTER：输出的卷积器的数目。
- NB_ROW:卷积窗口的高度，一般为 3、5 等奇数。
- NB_COL:卷积窗口的宽度， 一般是 3、5 等奇数。
- ROW: 输入二维数据的高度。
- COL: 输入二维数据的宽度。
- INPUT_DIM：输入数据的维数， 默认值 1.
- ACTIVATION: 卷积器的激活函数， 默认值 LINEAR。
- SUBSAMPLE_ROW: 卷积窗口在竖直方向每次移动的步长，默认值 1.
- SUBSAMPLE_COL: 卷积窗口在水平方向每次移动的步长， 默认值 1.
- OUTPUT_ROW: 输出数据的高度，为(ROW - NB_ROW)/SUBSAMPLE_ROW + 1。
- OUTPUT_COL: 输出数据的宽度， 为(COL - NB_COL)/SUBSAMPLE_COL + 1

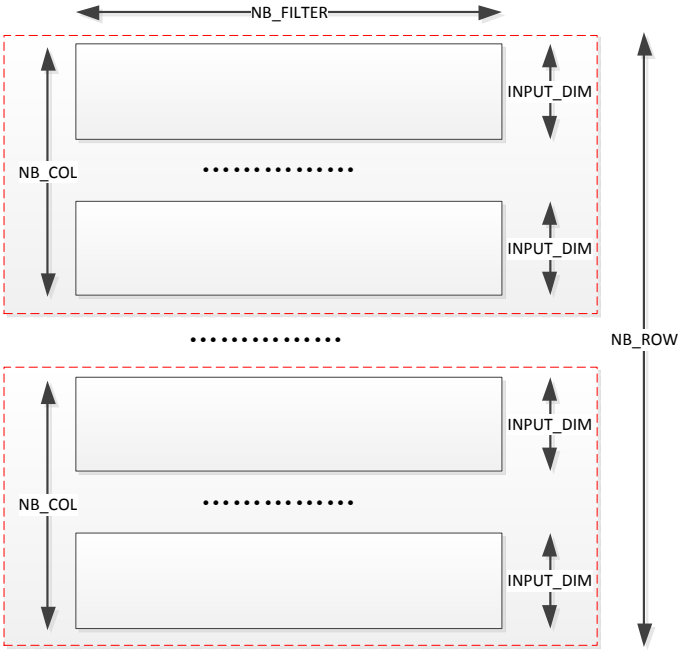
2.15.2. 成员变量

- **TYPE_T weight[NB_ROW][NB_COL][INPUT_DIM][NB_FILTER];**

Convolution2D 的权重是一个 4 维数组，NB_ROW * NB_COL * INPUT_DIM * NB_FILTER。

- 第一维是卷积窗口的高度 NB_ROW.
- 第二维是卷积窗口宽度 NB_COL.
- 第三维是输入数据维数 INPUT_DIM
- 第四维是输出滤波器数目 NB_FILTER.

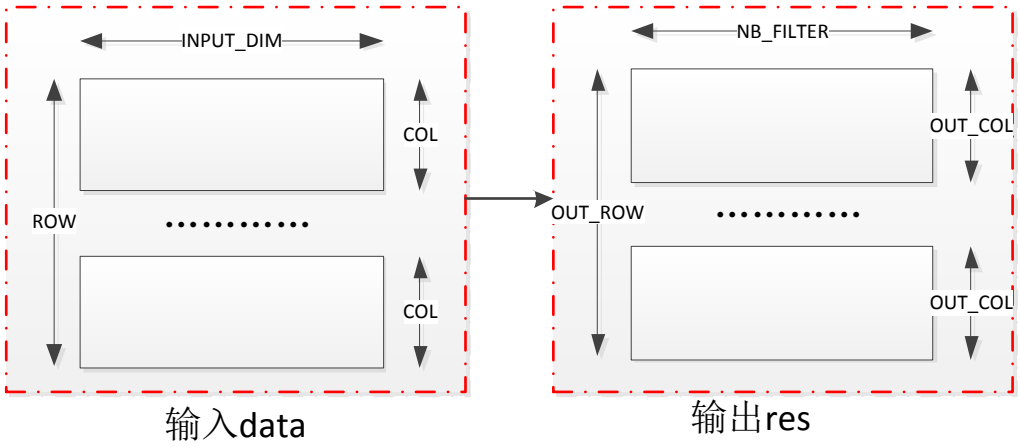
这个 4 维数组的存储方式与 Keras 生成的权重一致。



- **TYPE_T bias[NB_FILTER];**
  每个卷积器的偏置，因此 bias 是长度为 NB_FILTER 的一维数组。
- **TYPE_T res[OUT_ROW][OUT_COL][NB_FILTER];**
  Convolution2D layer 的输出结果，是一个 OUT_ROW * OUT_COL * NB_FILTER 的三维数组。

2.15.3. 成员函数

- **void feedforward**(TYPE_T data[**ROW**][**COL**][**INPUT_DIM**])
  Convolution2D 的 feedforward 函数。输入 data 是 ROW * COL * INPUT_DIM 的三维数组，计算结果保存在 res 中。数据输入输出如下图所示。



2.16. Convolution2D_DataStream

- 位置:convolution2D.h
- 日期：2016/10/18
- 描述：Convolution2D_DataStream 原理与 Convolution2D 相同，权重采用 Local 模式，输入输出数据采用 Stream 模式。

2.17. MaxPooling2D

- 位置：pooling2D.h
- 日期：2016/10/18
- 描述：对 2D 数据进行局部最大值法进行 pooling，实现对二维数据的压缩，一般在 Convolution2D 后使用。

2.17.1. 模板参数

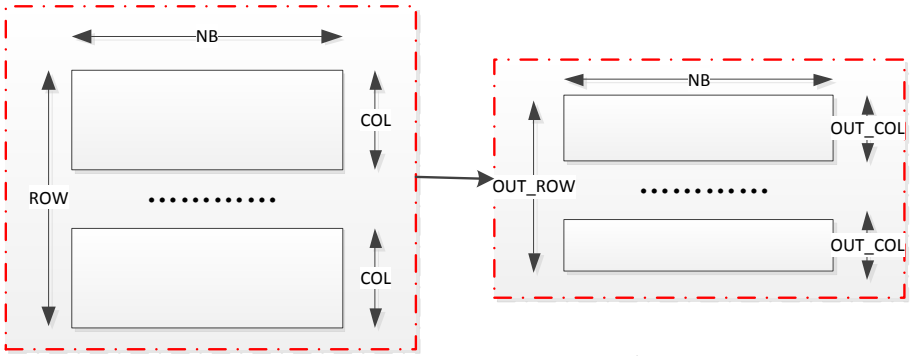
- ROW：输入二维数据的高度。
- COL: 输入二维数据的宽度。
- NB: 输入数据的维数。
- POOL_ROW：竖直方向 pooling 的步长，默认值 2.
- POOL_COL：水平方向 pooling 的步长， 默认值 2.
- OUT_ROW：pooling 后二维数据的高度，为 ROW/POOL_ROW
- OUT_COL：pooling 后二维数据的宽度， 为 COL/POOL_COL。

2.17.2. 成员变量

- TYPE_T res[**OUT_ROW**][**OUT_COL**][**NB**];
  pooling 后的输出结果是 OUT_ROW x OUT_COL x NB 的三维数组。

2.17.3. 成员函数

- **void feedforward**(TYPE_T data[**ROW**][**COL**][**NB**])
  MaxPooling2D 的 feedforward 函数，输入 data 是 ROW x COL x NB 的三维数组，输出保存到 res 中。



2.18. MaxPooling2D_Stream

- 位置：pooling2D.h
- 日期：2016/10/18
- 描述：原理与 MaxPooling2D 相同，输入输出数据采用 Stream 模式。

2.19. AveragePooling2D

- 位置:pooling2D.h
- 日期: 2016/10/18
- 描述：对 2D 数据进行局部平均法法进行 pooling，实现对二维数据的压缩，一般在 Convolution2D 后使用，其模板参数、成员变量等与 MaxPooling2D 类似。

2.19.1. 成员函数

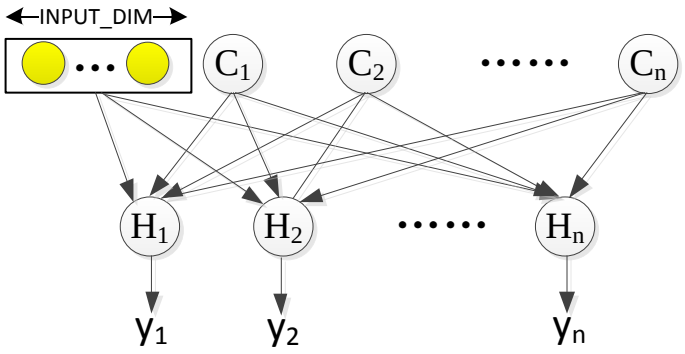
- **void feedforward**(TYPE_T data[**ROW**][**COL**][**NB**])
  该层的 feedforward 函数，输入 data 是 ROW x COL x NB 的三维数组，输出保存到 OUT_ROW x OUT_COL x NB 的三维数组 res 中。

2.20. AveragePooling2D_Stream

- 位置:pooling2D.h
- 日期: 2016/10/18
- 描述：功能与 AveragePooling2D 相同， 输入输出采用 Stream 模式。

2.21. SimpleRNN

- 位置: recurrent.h
- 日期:2016/11/1
- 描述: SimpleRNN 属于”Simple Recurrent Neural Network”。本文中 SimpleRNN 是基于 Elman Neural Network 原理，可参考链接: Simple Recurrent Neural Network。



SimpleRNN 是循环神经网络，其结构与全连接层 Dense 类似， 但是运行原理完全不同。在上图所示的 SimpleRNN 结构图中，真正的**输入元素**只有一个**(但是每个输入元素是 INPUT_DIM 维)**，即矩形内元素，因此实际上 SimpleRNN **每次只接受一个输入元素**。然而循环神经网络中引入了记忆单元，在 SimpleRNN 中就是上下文单元(Context Unit)，对应 C_x。当 SimpleRNN 定义了 n 个输出神经元时，就有 n 个上下文单元。这 n 个上下文单元 C_1-C_n 当做输入神经元处理， 因此 SimpleRNN 有 **n + INPUT_DIM** 个输入神经元，但是只有 INPUT_DIM 个真正意义上的输入神经元，有 n 个输出神经元。

SimpleRNN 的循环运作过程如下:

1) 假设输入向量长度为 M(INPUT_LENGTH)，即 **X** = [$\mathbf{X_0}$， $\mathbf{X_1}$，… $\mathbf{X_M}$]，每个向量元素 $\mathbf{X_i}$ 又是一个长度 INPUT_DIM 的数组。
2) 初始时所有的上下文单元置为 0，即 $C_x = 0$。
3) 向 SimpleRNN 中输入第一个输入元素 $\mathbf{X_0}$，然后按照全连接层计算方法，计算所有输出神经元**经过激活函数后**的输出值 **y** = [y_1,y_2,…y_n]。
4) 使用输出神经元的输出值 **y** 更新上下文单元 **C**，直接替换 $C_i = y_i$。
5) 然后输入第二个输入元素 $\mathbf{X_1}$，按照 3)-4)步骤处理。
6) 重复 3)-5)步骤， 直到输入向量 **X** 全部处理完成。

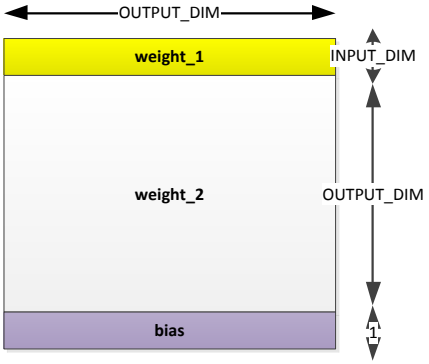
从上述步骤可以看出，SimpleRNN 一共执行了 M 次，即输入向量的长度 INPUT_LENGTH。每次执行过程，就是更新输出神经元输出值 **y** 的过程。

2.21.1. 模板参数

- INPUT_LENGTH: 输入向量的大小 M。
- INPUT_DIM: 输入向量的维度。
- OUTPUT_DIM: 输出神经元的数目 N。
- AC_FN: 输出神经元的激活函数。

2.21.2. 成员变量

- TYPE_T weight[**OUTPUT_DIM** + **INPUT_DIM** + 1][**OUTPUT_DIM**];
  SimpleRNN 中的权重 weight 和偏置 bias，由于输入神经元数目是 OUTPUT_DIM + INPUT_DIM， 输出神经元输出是 OUTPUT_DIM， 采用全连接方式， 因此权重是一个(OUTPUT_DIM + INPUT_DIM) * OUTPUT_DIM 的二维矩阵, 偏置 bias 是 1 * OUTPUT_DIM， 所以合起来是(OUTPUT_DIM + INPUT_DIM + 1) * OUTPUT_DIM 的二维矩阵。其中第一行(黄色)是真正的输入神经元的权重， 后 OUTPUT_DIM 行是上下文单元的权重。**存储方式与 Keras 一致， 不过 Keras 将前 INPUT_DIM 行和后 OUTPUT_DIM 行分开为两个独立的 ARRAY，实际使用时可以直接合并。**



- TYPE_T res[**OUTPUT_DIM**];

  SimpleRNN 的输出神经元的输出结果， 同时也是上下文单元，因为当前上下文单元的值就是上一次输出神经元的输出结果。**在 feedforward 函数中, res 需要被初始化为 0.**

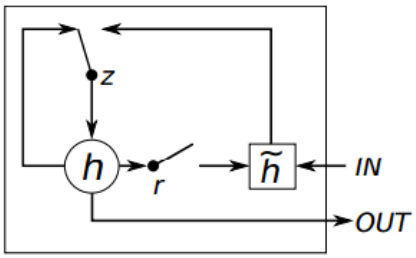
2.21.3. 成员函数

- **void feedforward**(TYPE_T data[**INPUT_LENGTH**][**INPUT_DIM**])

  SimpleRNN 的推断函数，输入向量 data 是 INPUT_LENGTHx INPUT_DIM 的二维数组。整个 SimpleRNN 会被顺序执行 INPUT_LENGTH 次， 最后输出结果保存到 res 中。

2.22. GRU

- 位置: recurrent.h
- 日期: 2016/11/3
- 描述: Gated Recurrent Unit(GRU)是 Gated Recurrent Neural Network 的核心组件，Gated Recurrent Neural Network 是 RNN 的改进版本，其原理可参考: Gated Recurrent Neural Network 说明。其主要的运行过程与 SimpleRNN 类似，对输入向量循环计算。



(b) Gated Recurrent Unit

$$z_t^j = \sigma\left(W_z \mathbf{x}_t + U_z \mathbf{h}_{t-1}\right)^j .$$

$$r_t^j = \sigma\left(W_r \mathbf{x}_t + U_r \mathbf{h}_{t-1}\right)^j .$$

$$\tilde{h}_t^j = \tanh\left(W \mathbf{x}_t + U\left(\mathbf{r}_t \odot \mathbf{h}_{t-1}\right)\right)^j$$

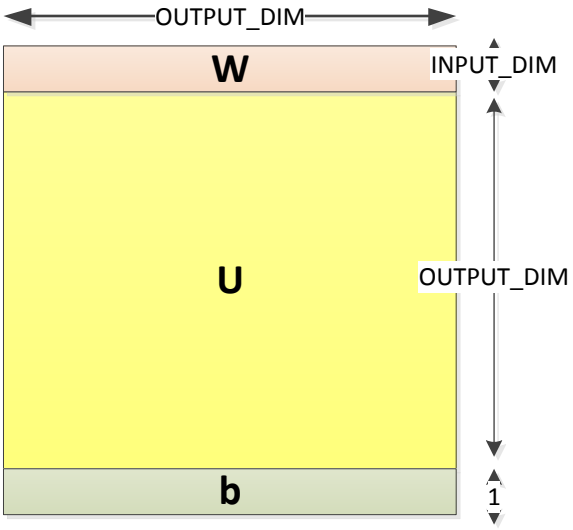
$$h_t^j = (1 - z_t^j) h_{t-1}^j + z_t^j \tilde{h}_t^j .$$

2.22.1. 模板参数

- INPUT_LENGTH: 输入向量的长度。
- INPUT_DIM：输入向量中每个元素的维度。
- OUTPUT_DIM： 输出神经元的数目。
- AC_FN: 输出神经元的激活函数， 默认是 TANH.
- INNER_AC_FN: 内部隐藏神经元的激活函数， 默认是 SIGMOID.

2.22.2. 成员变量

与 SimpleRNN 等其他 Layer 不同， GRU 内部包含 Update Gate, Reset Gate 和 Candidate Activation 三个独立的神经元， 它们分别拥有独立的权重和偏置， 而且三者的权重和偏置的存储方式和大小相同，是(OUTPUT_DIM + INPUT_DIM + 1) * OUTPUT_DIM 的二维数组。



该二维数组中，其实包含了{W, U, b}三部分，对应 GRU 结构**(参考 GRU 公式或者 Keras 中 GRU 源码)**。**W** 是 INPUT_DIM*OUTPUT_DIM, **U** 是 OUTPUT_DIM*OUTPUT_DIM， **b** 是 1*OUTPUT_DIM。下面的 weight_z, weight_r 和 weight_h 的结构都是如上图所示。

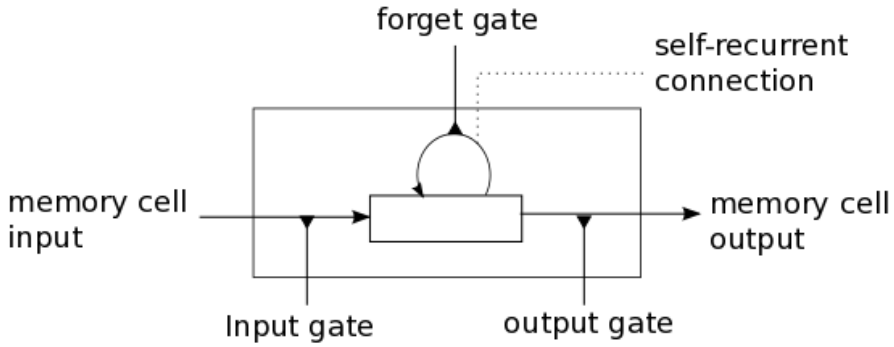
- TYPE_T weight_z[**OUTPUT_DIM** + **INPUT_DIM** + 1][**OUTPUT_DIM**];
  Update Gate 神经元的权重和偏置。
- TYPE_T weight_r[**OUTPUT_DIM** + **INPUT_DIM** + 1][**OUTPUT_DIM**];
  Reset Gate 神经元的权重和偏置。
- TYPE_T weight_h[**OUTPUT_DIM** + **INPUT_DIM** + 1][**OUTPUT_DIM**];
  Candidate Activation 神经元的权重和偏置。
- TYPE_T res[**OUTPUT_DIM**];
  输出神经元经过激活函数后的输出结果。
- TYPE_T rr[**OUTPUT_DIM**];
  Reset Gate 神经元经过激活函数后的值。
- TYPE_T zz[**OUTPUT_DIM**];
  Update Gate 神经元经过激活函数后的值。
- TYPE_T rh[**OUTPUT_DIM**];
  rr 与 res 的 element-wise 运算结果。

2.22.3. 成员函数

- **void feedforward**(TYPE_T data[**INPUT_LENGTH**][**INPUT_DIM**])
  GRU 的推断函数，输入数据 data 是 INPUT_LENGTH x INPUT_DIM 的二维数组。推断函数进行 INPUT_LENGTH 次循环运算，最后将输出结果保存到 res 中。

2.23. LSTM

- 位置: recurrent.h
- 日期: 2016/11/9
- 描述：基于 Long Short Term Momeory(LSTM)单元的 RNN 实现。实际上 LSTM 先与 GRU 出现，GRU 是基于 LSTM 的简化版本。原理可参考: LSTM Networks for Sentiment Analysis



The equations below describe how a layer of memory cells is updated at every timestep t. In these equations :

- x_t is the input to the memory cell layer at time t
- $W_i, W_f, W_c, W_o, U_i, U_f, U_c, U_o$ and V_o are weight matrices
- b_i, b_f, b_c and b_o are bias vectors

First, we compute the values for i_t, the input gate, and $\widetilde{C}_t$ the candidate value for the states of the memory cells at time t :

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$
$$\widetilde{C}_t = tanh(W_c x_t + U_c h_{t-1} + b_c)$$

Second, we compute the value for f_t, the activation of the memory cells' forget gates at time t :

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

Given the value of the input gate activation i_t, the forget gate activation f_t and the candidate state value $\widetilde{C}_t$, we can compute C_t the memory cells' new state at time t :

$$C_t = i_t * \widetilde{C}_t + f_t * C_{t-1}$$

With the new state of the memory cells, we can compute the value of their output gates and, subsequently, their outputs :

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + V_o C_t + b_o)$$
$$h_t = o_t * tanh(C_t)$$

其中， o_t 的计算使用下面简化的计算方法代替上述中的公式:

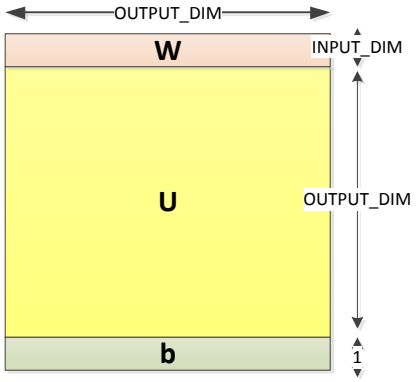
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

2.23.1. 模板参数

- INPUT_LENGTH: 输入向量的长度。
- INPUT_DIM：输入向量中每个元素的维度。
- OUTPUT_DIM： 输出神经元的数目。
- AC_FN: 输出神经元的激活函数， 默认是 TANH.
- INNER_AC_FN: 内部隐藏神经元的激活函数， 默认是 SIGMOID.

2.23.2. 成员变量

与 GRU 类似， LSTM 单元中包含 Input Gate, Memory Cell, Forget Gate 和 Output Gate 四个神经元，它们分别拥有独立的权重和偏置， 而且三者的权重和偏置的存储方式和大小相同，是(OUTPUT_DIM + INPUT_DIM + 1) * OUTPUT_DIM 的二维数组。



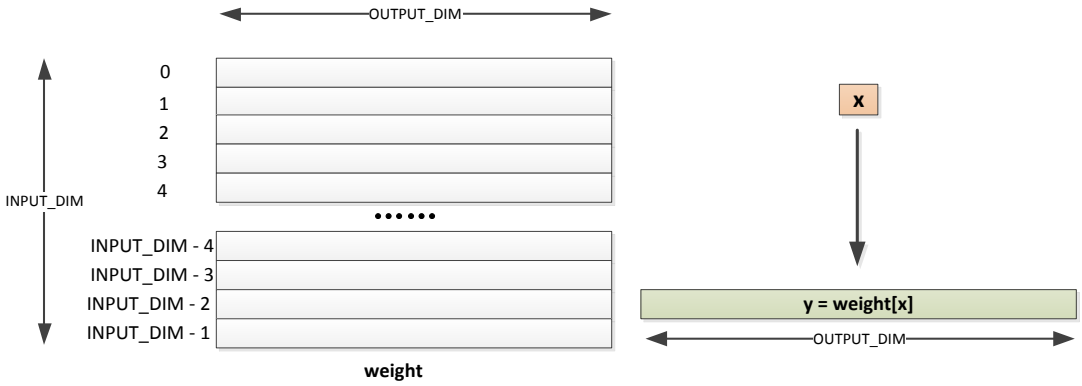
- TYPE_T weight_i[**OUTPUT_DIM** + **INPUT_DIM** + 1][**OUTPUT_DIM**];
  Input Gate 神经元权重和偏置。
- TYPE_T weight_c[**OUTPUT_DIM** + **INPUT_DIM** + 1][**OUTPUT_DIM**];
  Memory Cell 神经元权重和偏置。
- TYPE_T weight_f[**OUTPUT_DIM** + **INPUT_DIM** + 1][**OUTPUT_DIM**];
  Forget Gate 神经元权重和偏置。
- TYPE_T weight_o[**OUTPUT_DIM** + **INPUT_DIM** + 1][**OUTPUT_DIM**];
  Output Gate 神经元权重和偏置。
- TYPE_T res[**OUTPUT_DIM**];
  输出神经元计算结果。
- TYPE_T ct[**OUTPUT_DIM**];
  Memory Cell 运算的中间结果。

2.23.3. 成员函数

- **void feedforward**(TYPE_T data[**INPUT_LENGTH**][**INPUT_DIM**])
  GRU 的推断函数，输入数据 data 是 INPUT_LENGTH x INPUT_DIM 的二维数组。推断函数进行 INPUT_LENGTH 次循环运算，最后将输出结果保存到 res 中。

2.24. Embedding

- 位置: embedding.h
- 日期: 2016/11/9
- 描述：Keras 中的 Embedding Layer，将正整数转换成固定大小的密集向量 Dense Vector。其计算过程本质上是查找表 Look-Up Table，输入数据的每个元素值 v，以 v 为索引号，在权重参数 weight 中查找到对应的输出 Dense Vector。因此 Embedding Layer 可以看成通过查找表进行编码的过程。



注意，Embedding Layer 只能作为网络的第一层，输入数据类型必须是正整数。

2.24.1. 模板参数

- INPUT_DIM: 查找表的长度，因此输入的正整数向量中，最大值不能超过 INPUT_DIM，否则索引值会越界出错。
- OUTPUT_DIM：输出的密集向量 Dense Vector 长度。
- NB_SAMPLES: 输入数据序列的数目。
- INPUT_LENGTH:输入数据序列的长度。

2.24.2. 成员变量

- TYPE_T weight[**INPUT_DIM**][**OUTPUT_DIM**];
  如上图所示，Embedding 的权重参数是 INPUT_DIM x OUTPUT_DIM 的二维数组。
- TYPE_T res[**NB_SAMPLES**][**INPUT_LENGTH**][**OUTPUT_DIM**];
  输出结果是 NB_SAMPLES x INPUT_LENGTH x OUTPUT_DIM 的三维数组， 因此输入数据中的每个元素对应的输出是一个长度为 OUTPUT_DIM 的向量，因此输入是二维数组，输出是三维数组。

2.24.3. 成员函数

- **void feedforward**(TYPE_PINT data[**NB_SAMPLES**][**INPUT_LENGTH**])
  Embedding Layer 的推断函数， 其原理是 weight 当做查找表 Lookup-Table， 然后通过查找表实现输入到输出之间的映射。

2.25. Embedding_DataStream

- 位置: embedding.h
- 日期: 2016/11/9
- 描述：原理与 Embedding 相同，输入输出采用 Stream 模式。

3. 注意事项

3.1. 权重初始化

在所有 Layer 中， 权重参数都是在构造函数中进行初始化的。实例化一个类时， 需要传入权重参数的**一维**数组， 然后在构造函数中，通过拷贝转换成多维数组。比如 Convolution2D 中，将输入的一维数组 WEIGHT 转换成 4 维数组 weight。从传统 C++语法上看， 这种做法是低效率的， 因为存在额外的数据拷贝，相反，直接把一维数组的指

针传递进去就可以了。但是这么做的理由如下：

```
/* initialize the weight and bias */
for( int i = 0; i < NB_ROW; i++)
{
    for( int j = 0; j < NB_COL; j++)
    {
        for( int m = 0; m < INPUT_DIM; m++)
        {
            for(int n = 0; n < NB_FILTER; n++)
                weight[i][j][m][n] = WEIGHT[ i*NB_COL*INPUT_DIM*NB_FILTER + j*INPUT_DIM*NB_FILTER + m*NB_FILTER + n];
        }
    }
}
for( int i = 0; i < NB_FILTER; i++)
{
    bias[i] = BIAS[i];
}
```

1) 在 HLS 编译时候， 其对**多维数组的优化比一维数组的优化效率高**，如果使用一维数组去表示多维数组，编译器不能很好的对 ARRAY 进行 PARTITION，从而导致 Memory Bottle，因此所有 Layer 中的数据都采用多维数组表示，而不是一维数组。
2) HLS 实质上是编译成硬件 RTL，运行方式不是传统的 C++执行方式。构造函数中，将一维数组转换成多维数组的代码， 实际上会被自动优化掉，weight 会被综合成 Distributed RAM，**这段代码是不存在时钟周期消耗的，** 这是个 power-on initialization 的过程。
3) 使用多维数组运算，可以简化代码的可读性和可维护性。

3.2. 顶层 Pipeline 指令

构建完整网络后，对输入数据流进行处理时，一般采用流水方式可实现高性能。在顶层函数的 for 循环中加入约束优化指令#pragma HLS pipeline II=XXX 实现。约束 XXX 值越小，总性能越高， 但是综合编译难度越大，硬件资源占用率也越大。**当深度网络比较复杂时， 不要使用该优化指令， 否则无法编译成功。**

根据经验，XXX 值设定如下，可在性能和资源利用率之间实现较好的平衡。

$$XXX = Len + 1$$

其中 Len 是输入层(也是网络第一层)样本的数据长度。在给输入层准备数据阶段，需要 Len 个时钟周期从数组(通常是 AxiStream 总线)读取数据，然后额外的 1 个时钟周期给每次的迭代一个缓冲。因此根据经验， XXX = Len + 1 是不错的选择。

```
    for( int i = 0; i < NN; i++)
    {
#pragma HLS LOOP_TRIPCOUNT min=10 max=1000
/* the II should be the INPUT_DIM + 1, in this demo,
 * it should be II = STEP + 1= 10 + 1 = 11,
 * so a good balance between the performance and resource utilization can be achieved */
#pragma HLS pipeline II=11


        /* prepare the input data */
        for( int k = 0; k < STEP; k++)
            data[k] = sample[i * STEP + k];

        /* the feedforward process */
        conv1D.feedforward(data);
        pool1.feedforward( conv1D.res );
        pool2.feedforward( pool1.res );
        dense.feedforward( pool2.res );

        /* save the result */
        for( int v = 0; v < NB_CLASS; v++)
            result[i* NB_CLASS + v] = dense.res[v];
    }
```

以 conv1D_v1 工程 top.cpp Neural 函数为例，输入层的数据长度为 STEP=10，因此将 II 设置成 STEP + 1 = 11。(如果 II=10，资源利用率将急剧上升，II <10 综合编译器无法实现。)