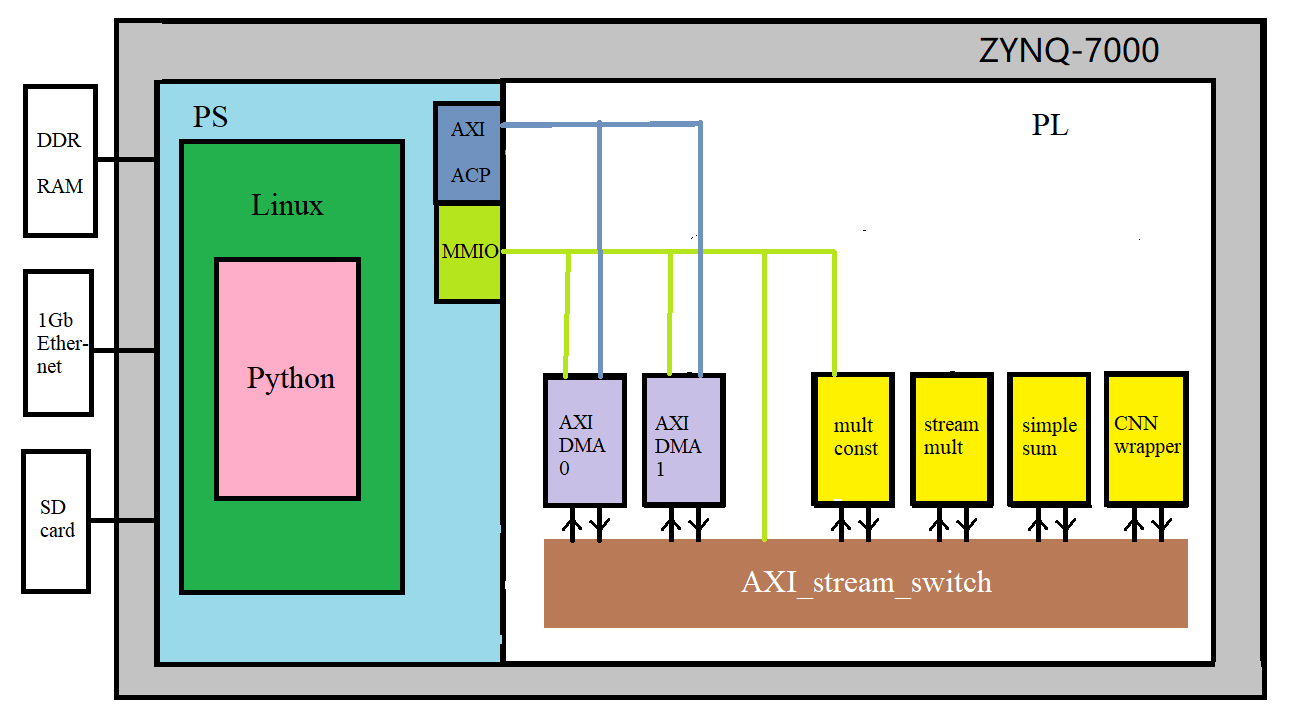
# 分类器（PYNQ- [Classification](https://github.com/louisliuwei/PYNQ-Classification)）

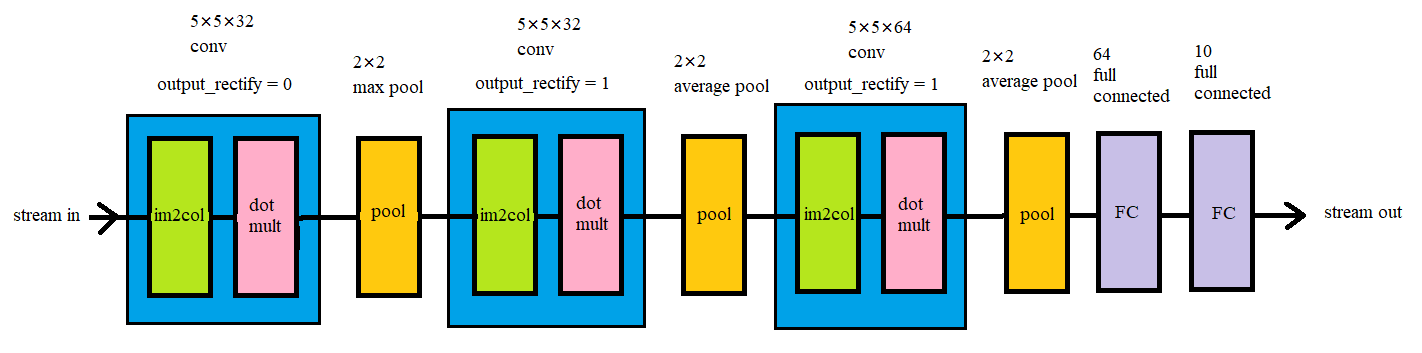
## 

## Overlay 简介（Overlay功能/外设/系统架构/基本原理）

PYNQ-Classifcation是一个软件上基于caffe和Lasagne，硬件加速的CNN分类器项目，它使用python语言在PYNQ框架下进行软件编程和硬件驱动。



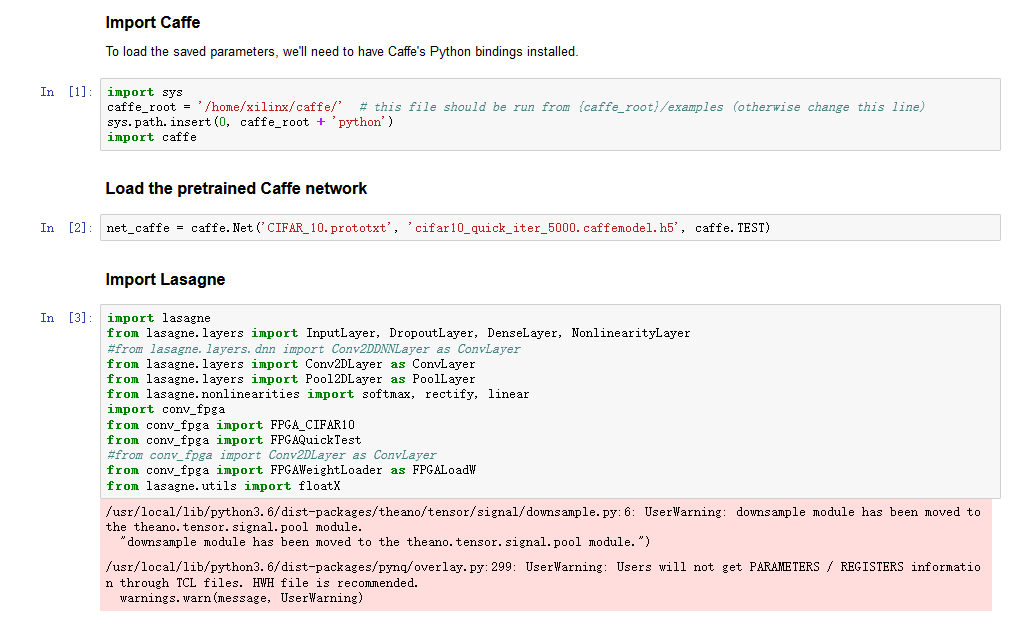
上图为工程项目示意图。通过软硬件划分，在PS端中负责Linux和通信，而PL端负责运算加速。在PYNQ中加载bit时，PL端的控制部分将以IP核形式呈现。通过Python，对IP核的参数配置来实现控制。



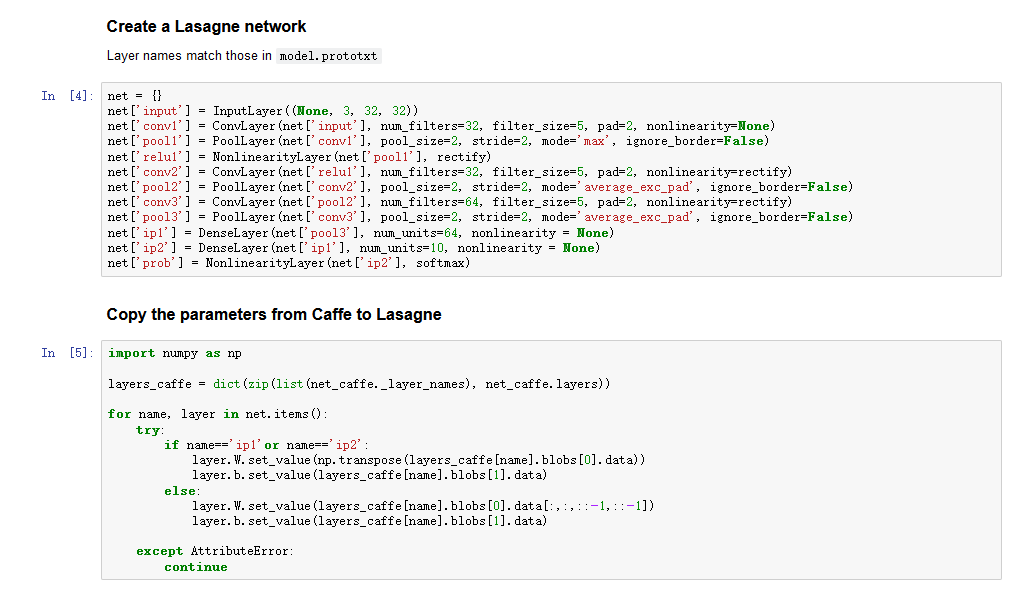
此项目中采用的CNN结构示意图(以CIFAR\_10为例子)，本项目中的卷积层由两个子层构成，一个是im2col子层，它的作用是将输入矩阵中的卷积窗口展开乘一维向量，另一个是点乘子层，它的作用是将im2col子层输出的向量和展开成一维的卷积核相乘加。卷积层有一个参数是output\_rectify，这对应的是当前卷积层是否应用Relu非线性层，0为否，1为是。池化层(pool)有两种选项，一是最大值池化，二是平均值池化。

## 示例Notebook(以CIFAR\_10为例)

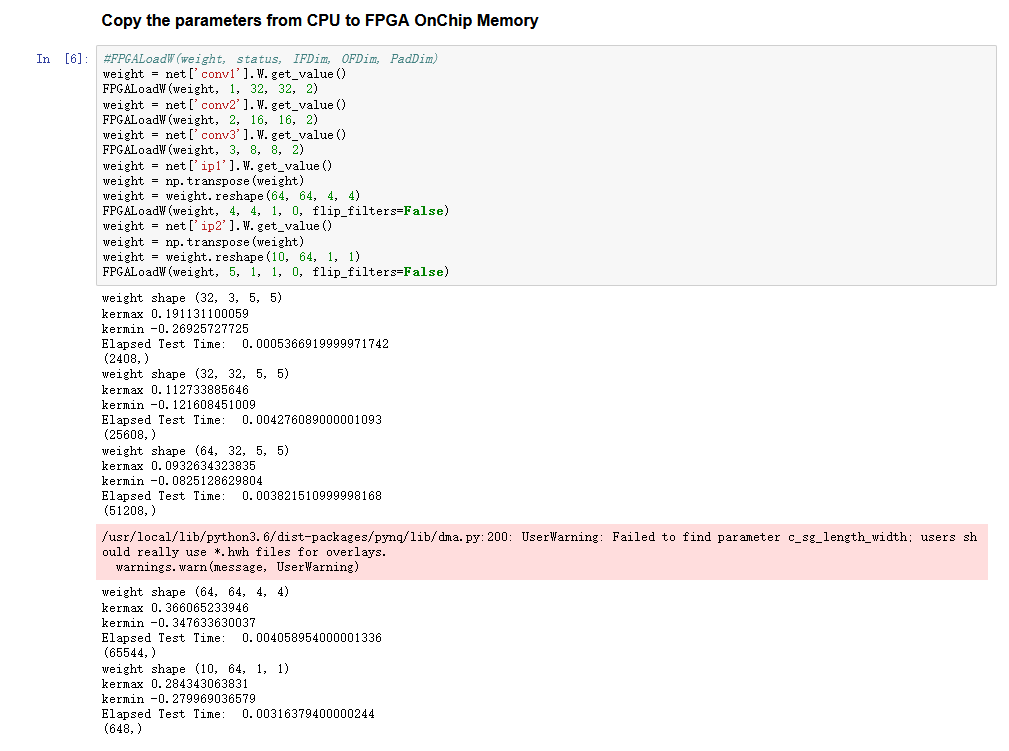
打开PYNQ-Classification/python\_notebooks/Theano/CIFAR\_10/Using a Caffe Pretrained Network - CIFAR10.ipynb。



开始时将库都包含进工程，并且读取caffe中预先训练好的CIFAR\_10模型。



然后建立一个Lasagne的网络结构，这个网络结构和硬件中的结构类似，硬件中去掉了第一个ReLU层。将训练好的模型参数导入到这个网络中。



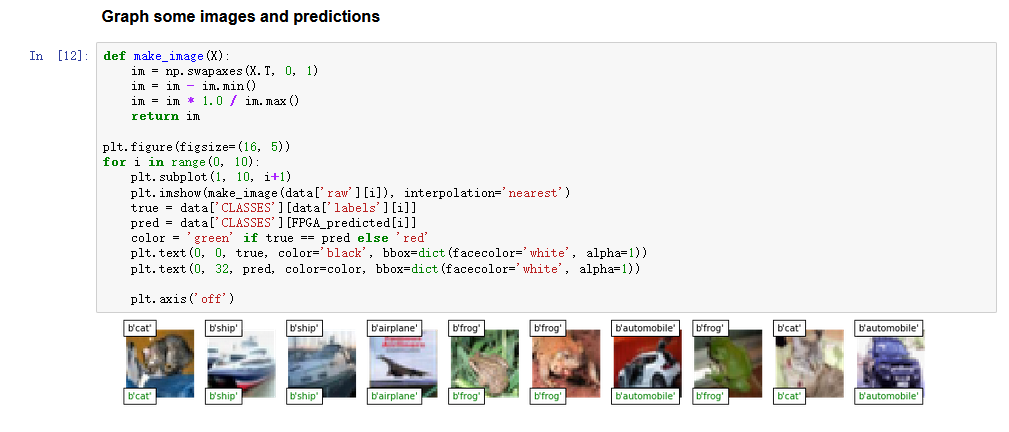
下一步我们将模型参数导入硬件，使用FPGALoadW函数来实现这一步。函数会打印出一些有关信息。



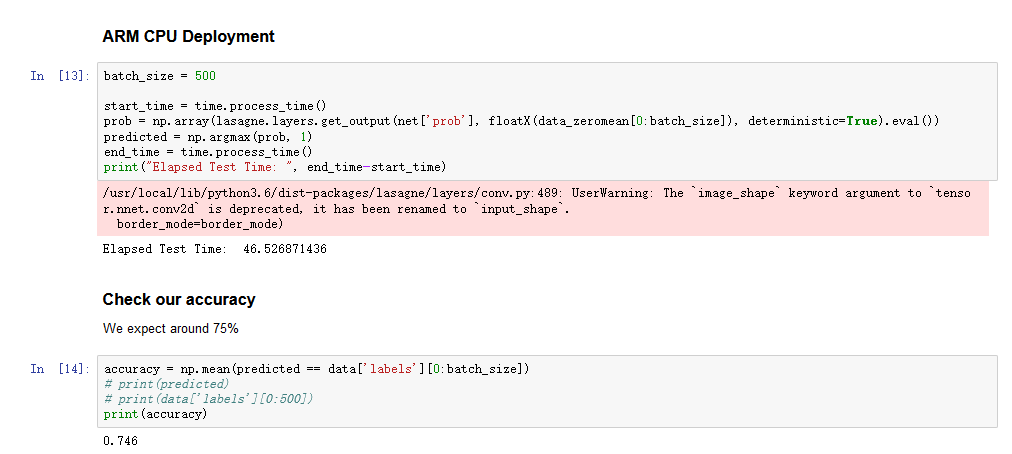
接下来导入CIFAR\_10数据集。



这里我们构建一张新的网络，把原来的中间层换成我们自定义的FPGA\_CIFAR10层，这一层把原来的中间层全部包含了进去，并在FPGA上实现。然后输入数据集，得到结果，结果保存在FPGA\_predicted中。最后显示了准确度，达到72.8%。



这里我们可以看到这个网络的部分识别结果。



这里再用CPU运行一遍这个网络，可以看到它用时46.5秒，比使用FPGA完成卷积慢了很多。

## Overlay详解（Vivado需升级到2018.3版本）

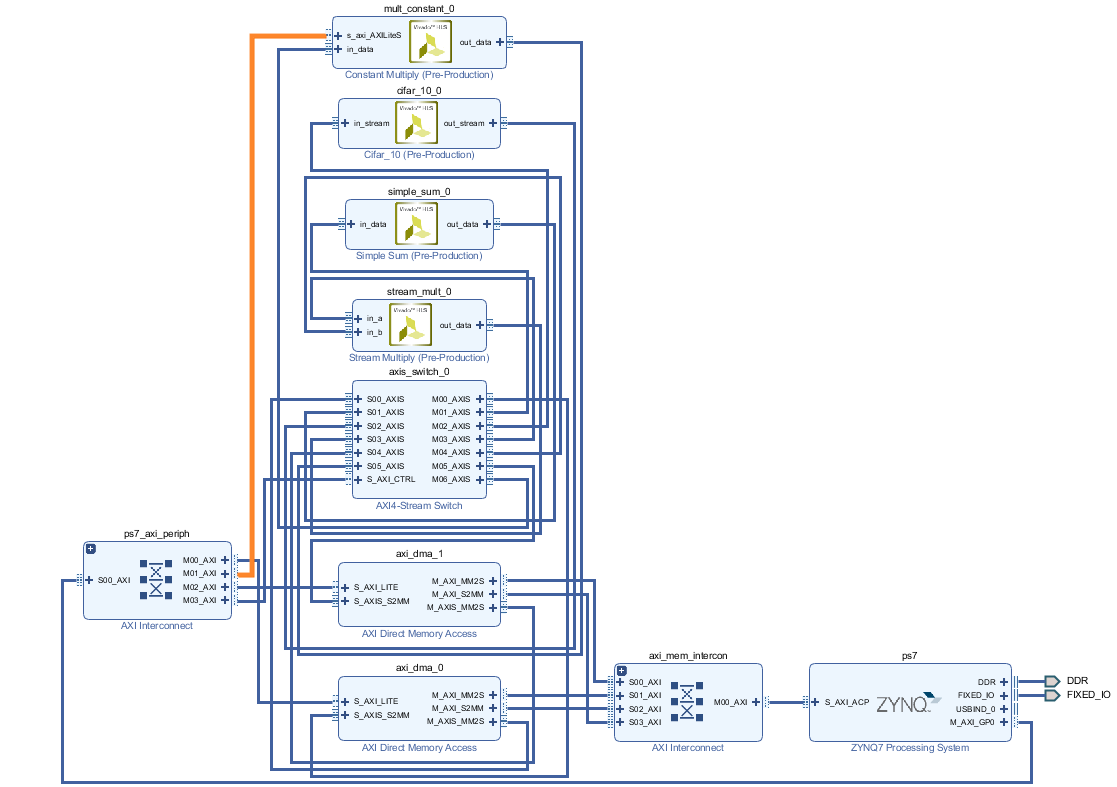
* PS与PL功能划分

PS部分主要是用Python运行caffe和Lasagne,构建用户交互的接口界面，并且通过PYNQ框架控制PL中各IP的运行。

PL部分则发挥FPGA高并行高吞吐量的特性，实现关键算法的加速，本例中主要指CNN的正向传播过程。

* Vivado工程block design介绍

具体硬件设计是这样的：



本项目中，例化两个AXI\_DMA进行数据传输，图中黄色的4个HLS编写的IP是算法加速部分，它们的输入输出接口均为AXI\_stream。AXI\_DMA和算法加速模块通过AXI\_stream\_switch相连。其中mult\_const模块是将输入数据乘以一个可配置的常数，stream\_mult模块是将两组数据对应相乘，simple\_sum模块是将输入数据求和。CNN\_wrapper封装了整个CNN结构，在这个项目中，它就是cifar\_10\_wrapper。AXI\_DMA、AXI\_stream\_switch和mult\_const模块用接入PS端的MMIO，通过AXI\_lite总线来配置它们的寄存器。

两个AXI\_DMA通过PS提供的AXI\_ACP接口与PS的内存控制器相连，可以不依赖CPU传输数据。它们可以将内存中某一区域的数据，以AXI\_stream协议数据流输出，也可以将AXI\_stream协议数据流写回内存。

DMA模块与AXI\_stream\_switch模块相连，AXI\_stream\_switch是AXI\_stream流的交换机，由于AXI\_stream协议不含地址，所以需要依靠交换机控制数据流的流向。AXI\_stream\_switch模块可以通过AXI\_lite接口（接入PS的MMIO）配置。

Constant Multiply模块用于将数据流中的数据与一个常数相乘，这个常数可以通过AXI\_lite接口配置。

Simple Sum模块用于将数据流中的数据求和。

Stream Multiply模块可以将两股数据流中的元素对应相乘，使用时，控制数据流交换机使它的两个输入端口与两个DMA输出，使它的两个输出端口与两个DMA的输入相连。

Cifar\_10模块就是CNN加速器，它的输入输出都是数据流。它将在下一节被介绍。

* PL侧关键算法（加速器）解释

为了解释Cifar\_10模块，我们先来看看它的HLS代码（CIFAR\_10\_wrapper.cpp）：

#include"CIFAR\_10\_wrapper.h"

void cifar\_10(

hls::stream<AXI\_DMA\_IO> &in\_stream,

hls::stream<AXI\_DMA\_IO> &out\_stream

){

#pragma HLS DATAFLOW

#pragma HLS INTERFACE axis port=in\_stream

#pragma HLS INTERFACE axis port=out\_stream

#pragma HLS INTERFACE ap\_ctrl\_none port=return

// Defining Interconnections

hls::stream<AXI\_VAL> connect\_0, connect\_1, connect\_2, connect\_3, connect\_4, connect\_5, connect\_6, connect\_7, connect\_8, connect\_9, connect\_10, connect\_11;

#pragma HLS STREAM variable=connect\_11 depth=50

#pragma HLS STREAM variable=connect\_10 depth=50

#pragma HLS STREAM variable=connect\_9 depth=50

//#pragma HLS STREAM variable=connect\_8 depth=50

#pragma HLS STREAM variable=connect\_7 depth=50

#pragma HLS STREAM variable=connect\_6 depth=50

//#pragma HLS STREAM variable=connect\_5 depth=50

#pragma HLS STREAM variable=connect\_4 depth=50

#pragma HLS STREAM variable=connect\_3 depth=50

//#pragma HLS STREAM variable=connect\_2 depth=50

#pragma HLS STREAM variable=connect\_1 depth=50

#pragma HLS STREAM variable=connect\_0 depth=50

它的结构和简介中的一样，每一层之间的连接通过一个深度50的FIFO连接。在本项目中，一般的卷积层被分成了两个子层。SCIG是im2col子层，它将输入矩阵的卷积窗口展开乘一维向量，送给SMM；SMM是点乘子层，它将卷积窗口中的数据和卷积核相卷积，得到结果。池化层pool和全连接层FC和通常情况一致。

// void SCIG<KerDim\_curr, IFMCH\_curr, IFMDim\_curr, OFMCH\_curr, OFMDim\_curr, PadDim\_curr = 0>(in, out)

// void SMM<A\_COL\_MAX, A\_ROW\_MAX, B\_COL\_MAX>(in\_stream\_a, out\_stream, layer\_id, output\_rectify = 1, FACTOR)

// void pool<pool\_size, In\_CH\_MAX, IFMDim\_MAX> (in, out, layer\_id, pool\_mode, // 0 for max pooling, 1 for average pooling, const bool output\_rectify = 1)

AXI\_DMA\_SLAVE(in\_stream, connect\_0);

SCIG<5, 3, 32, 32, 32, 2>(connect\_0, connect\_1);

SMM<1, 75, 32>(connect\_1, connect\_2, 1, 0, 25);

pool<2, 32, 32>(connect\_2, connect\_3, 1, 0, 1);

SCIG<5, 32, 16, 32, 16, 2>(connect\_3, connect\_4);

SMM<1, 800, 32>(connect\_4, connect\_5, 2, 1, 25);

pool<2, 32, 16>(connect\_5, connect\_6, 2, 1, 0);

SCIG<5, 32, 8, 64, 8, 2>(connect\_6, connect\_7);

SMM<1, 800, 64>(connect\_7, connect\_8, 3, 1, 25);

pool<2, 64, 8>(connect\_8, connect\_9, 3, 1, 0);

FC<1, 1024, 64>(connect\_9, connect\_10, 4, 0, 32);

FC<1, 64, 10>(connect\_10, connect\_11, 5, 0, 16);

AXI\_DMA\_MASTER(connect\_11, out\_stream);

}

接下来我们接下来看一下它接受的数据流的结构：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8~… |
| Status | Batch\_size | ConvKernelDim | IFMCH | IFMDim | OFMCH | OFMDim | PadDim | Data |

Status ：0的时候意味着工作状态，此时不会更新权重，其他值时候意味着更新权重的层号(layer\_ID),这一层会更新权重。layer\_ID在每一层例化时作为参数输入确定。

Batch\_size：输入图片的个数。

ConvKernelDim：卷积核的边长。

IFMDim：输入矩阵的边长（Input Feature Matrices Dimension）

IFMCH：输入矩阵的通道数（Input Feature Matices Channels）

OFMDim：输出矩阵的边长（Outnput Feature Matrices Dimension）

OFMCH：输出矩阵的通道数（Output Feature Matices Channels）

Data: 输入的数据

每一层都会收到以上数据头，来确定本层的工作状态。最后CNN输出的结果也会包含这个数据头，真正的数据从第8位开始。

* PS侧环境准备（如安装第三方库安装步骤）

PYNQ版本v2.4

Python版本 3.6.5

1、安装caffe

-安装依赖

'''

sudo apt-get install libprotobuf-dev libleveldb-dev libsnappy-dev libopencv-dev libhdf5-serial-dev protobuf-compiler

sudo apt-get install --no-install-recommends libboost-all-dev

sudo apt-get install libgflags-dev libgoogle-glog-dev liblmdb-dev

sudo apt-get install libopenblas-dev libopenblas-base

'''

-安装protobuf3

'''

sudo pip3 install protobuf

'''

-由于PYNQ-Z2内存只有512MB，编译caffe需要更多内存，插入闪存盘作为虚拟内存，容量大小4GiB证实可行

'''

mkswap /dev/sda

swapon /dev/sda

'''

-下载caffe

'''

cd /home/xilinx

git clone https://github.com/BVLC/caffe.git

cp /home/xilinx/jupyter\_notebooks/PYNQ-Classification/tools/CAFFE\_ON\_PYNQ/Makefile.config caffe #复制Makefile.config到caffe根目录

cd caffe

'''

-编译caffe

'''

make all

make test

make runtest

'''

-弹出闪存盘

'''

swapoff /dev/sda

'''

-安装pycaffe

'''

cd python

'''

修改requirements.txt中的python-dateutil>=1.4,<2为python-dateutil

'''

for req in $(cat requirements.txt); do sudo pip3 install $req; done

echo "export PYTHONPATH=$(pwd):$PYTHONPATH " >> ~/.bash\_profile

source ~/.bash\_profile

cd ..

make pycaffe

'''

在.bashrc最后添加如下内容

export PYTHONPATH=/home/xilinx/caffe/python

完成后，就可以运行样例

/home/xilinx/jupyter\_notebooks/PYNQ-Classification/python\_notebooks/Theano/CIFAR\_10/ Using a Caffe Pretrained Network - CIFAR10.ipynb

和

/home/xilinx/jupyter\_notebooks/PYNQ-Classification/python\_notebooks/Theano/Lenet/Using a Caffe Pretrained Network - LeNet5.ipynb

2、安装Lasagne

'''

cd /home/xilinx

git clone https://github.com/Lasagne/Lasagne.git

cd Lasagne

pip3 install Theano

'''

删除requirements-dev.txt 内的-r requirements.txt这一行

'''

pip install Lasagne==0.1

'''

* Overlay API介绍

在此工程中，通过

FPGALoadW(weight, status, IFDim, OFDim, PadDim)

来往FPGA内写入参数，weight是权重，status是layer\_ID。

工程中按照Lasagne的规范定义了FPGA\_CIFAR10和FPGA\_LENET两个层类，可以直接整合进Lasagne构建的网络内。用法类似：

FPGA\_net = {}

FPGA\_net['input'] = InputLayer((None, 3, 32, 32))

FPGA\_net['cifar10'] = FPGA\_CIFAR10(FPGA\_net['input'])

FPGA\_net['prob'] = NonlinearityLayer(FPGA\_net['cifar10'], softmax)

然后按照Lasagne的用法即可。

对于FPGA的配置等操作，通过DMAWrapper类，StreamingSwitch类操作，这两个类在prepare\_execution和execute\_hardware这两个函数调用。而这两个函数在hardware\_function装饰的double,mult,total这三个函数里调用。

配置IP都利用MMIO读写寄存器，而DMA的传输需要使用xlnk来手动分配管理内存，再利用DMA类中的writechannel和readchannel的transfer方法启动数据传输。对DMA硬件的驱动在prepare\_execution和execute\_hardware这两个函数以及DMAWrapper类中实现。具体细节参看样例中的conv\_fpga.py。

## 完整工程与源码（百度网盘链接）