Proposal for Final Project of FPGA Image Processing

Xinyu Zhang
xiz368@eng.ucsd.edu

May 3, 2016

What's changed from the Draft

I have spent lots of time reading the paper[1] and its matlab code thoroughly, but painfully saying I still could not understand it to the level of being able to reimplementing it totally and the algorithm is over complex. So I change my plan. I am going to work on convolution neural network.

1 Introduction

There are currently many convolution neural network implementation open source available, but none of them is able to work with FPGA. So here I plan to implement a neural network framework that has following features:

- Configurable
  - Layer Parameter
  - Activation Type
- Easy to Integrate with existing Work
  - Able to import trained model from caffe[2]
  - Modular and easy to add more functionality.
- Not mean to be complete, but has fundamental utilities to do some basic practical work.
  - Layers
    - Convolution Layer
    - Fully Connected Layer
    - Max Pooling Layer
  - Activation Type

        * Sigmoid
    - Output
        * Final Output Callback, give control back to users

The rest of the proposal is organized as follow. In the section 2, I will give a brief introduction of convolution neural network. In the section 3, I will explain some important profiling goals and implementation details. In the section 4, I will give the rough timeline for the project and final delivery plan.

2 Convolution Neural Network

A Convolution Neural Network (CNN) is comprised of one or more convolution layers (often with a subsampling step) and then followed by one or more fully connected layers as in a standard multilayer neural network.

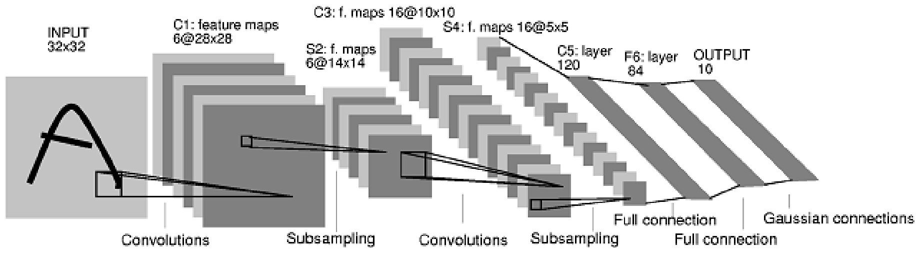

Figure 1: Convolution Neural Network Structure

The architecture of a CNN is designed to take advantage of the 2D structure of an input image (or other 2D input such as a speech signal). This is achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features. Another benefit of CNNs is that they are easier to train and have many fewer parameters than fully connected networks with the same number of hidden units. In this article we will discuss the architecture of a CNN and the back propagation algorithm to compute the gradient with respect to the parameters of the model in order to use gradient based optimization. See the respective tutorials on convolution and pooling for more details on those specific operations.

3 Essentials in the Project

3.1 Profiling

3.1.1 Convolution Layer

There are a couple of existing ways to implement a efficient convolution, like

- Fast Fourier Transform [3]

- 3D Convolution as Matrix Multiplication [4]
- Blocked Convolution (Loop Tiling)

I will implement a basic version and then leverage line/window buffers. If later on I still have time, I will try the above advanced tricks, and for every version, I will keep benchmarks.

3.1.2 Max Pooling

Max Pooling is image subsampling, which is a common operation in the image processing but not provided in the HLS OpenCv [5]. I will try to leverage line/window buffer to implement the image subsampling in O(n) and pixel-streaming way.

3.1.3 Fully Connected Layer

Fully connected layer is basically a inner product operation, which is supposed to be straightforward to optimize.

3.1.4 Sigmoid

Sigmoid needs exponential operation, which is not obvious to implement in the hardware. But I think HLS $< math.h >$ will handle the complexity.

3.1.5 Global Optimization

Since neural network is pipelined in nature, so it would be possible to write the whole neural network in a streaming way like this [6].

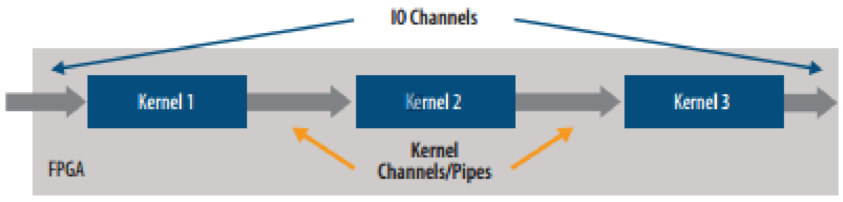

Figure 2: Pipeline

3.2 Configurable

Neural Network implementation is good only when the code could be reused based on different network parameters, like different number of layers, different activation functions. And C++ Template functionality[7] is very good for achieving this goal.

To make it configurable here could be easier than that in GPU/CPU, since we only need to implement forward network for FPGA. The backward network contains way more parameters, though FPGA contains its own difficulties.

3.3 Modular

To make the code modular is crucial. If someone uses your code and wants to customize based on their needs, it's easy for them to just write their own layer class and inherit built-in class, but not dive into your code.

3.4 Caffe Converter

To make this a real-world project, it has to be able to work together with existing popular neural network library and import their pretrained network model. Caffe[2] is arguably the most popular convolution neural network library at presents and they used google protocol buffer[8] to store their trained model. Here I need to implement a function to parse the data from the trained model.

3.5 Callback to the User

When using neural network, the program may uses its output as input to other procedure. In order to meet this need, a final callback to provide neural network result to users will be implemented.

3.6 Running on the real device

Port the simulated code into real device could be a really good learning experience, and will pave the way for future usage.

4 Timeline and Delivery

4.1 Schedule

Assuming the final due date is June 07, I give my rough timeline for this project.

1. (May 03 - May 07) Do surveys on technologies be used, and design the code architecture.
2. (May 08 - May 27) Implement the main functionality.
   (a) Caffe Converter
   (b) Structure and Base Class
   (c) Layers
   (d) Misc (Like Final Callback)
3. (May 28 - June 02) Run and Test the Neural Network, make it work in the simulated environment.

4. (June 03 - June 06) Port it to the real FPGA.

4.2 Delivery

In the best case, the final delivery will be an usable convolution neural network framework for FPGA in Vivado HLS C[9].

References

[1] D. Ramanan X. Zhu. Face detection, pose estimation, and landmark localization in the wild. *CVPR*, 2012.

[2] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[3] Wikipedia. Fast fourier transform — wikipedia, the free encyclopedia, 2016. [Online; accessed 2-May-2016].

[4] https://yunmingzhang.wordpress.com/2015/03/13/how-does-3d-convolution-really-works-in-caffe-a-detailed-analysis/.

[5] http://www.wiki.xilinx.com/hls+video+library.

[6] http://www.embedded-vision.com/platinum-members/altera/embedded-vision-training/videos/pages/may-2015-embedded-vision-summit.

[7] Wikipedia. Template (c++) — wikipedia, the free encyclopedia, 2016. [Online; accessed 2-May-2016].

[8] https://developers.google.com/protocol-buffers/.

[9] http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.