Convolution Optimization with Zynq FPGAs

With Application to Convolutional Neural Networks

Michael Losh, Oluwakemi Adabonyan
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: mlosh@oakland.edu, onadabonyan@oakland.edu

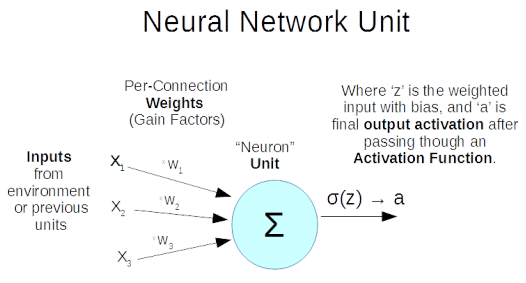
***Abstract*—We have designed a Zynq AXI Full Interface peripheral to calculate vector dot products as a parallel operation in hardware targeting Convolution operations such as found in Convolutional Neural Networks. Execution speed of an example neural network using the dot product peripheral was slightly better than a random-access software-only implementation, but actually slower than a dot-product optimized software-only implementation. The performance limitations are analyzed and possible further enhancements identified.**

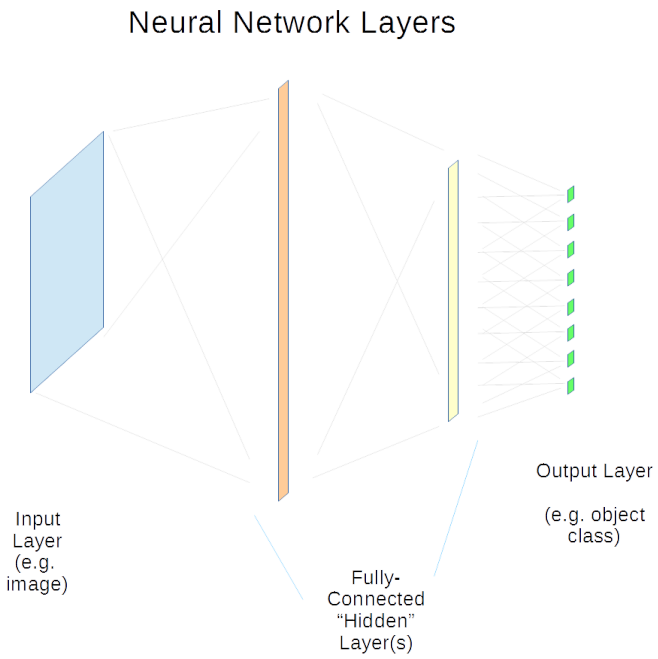
I. Introduction

Neural Networks are arrangements of signal processing units inspired by neural systems of real organisms with many simplifications to make their implementation in a wide variety of applications such as signal filtering, feature detection, and stimulus classification feasible. Neural network models for signal processing and recognition were explored in the 1950s and 1960s, but a lack of scalable training methods and suitable hardware platforms for large networks limited widespread adoption. Interest waned until *backpropagation* "learning" techniques were developed in the 1980s that enabled usefully-sized networks for a variety of real tasks to be trained[1]. Most examples of neural nets in this ear were typically implemented on standard digital computers, although some custom hardware chips for implementing processing units with configurable connections were developed and demonstrated in the 1980s and 1990s (Synaptics Inc. was founded by early researchers in this field). Interest in neural networks plateaued again due in part to attention placed in *Digital Signal Processing* (DSP) and statistical methods of *Machine Learning* (ML) until the late 1990s through today, when a neural network architecture further inspired by the visual cortex of animals was developed and found a much more tractable method of implementing many-layered networks, also known as *deep* networks. This network architecture is called a *Convolutional Neural Net* or CNN because the core feature of it is a form of convolution operator within the early layer(s) of the network. In the 2000s and 2010s, implementations of CNN and other deep networks received a significant boost in training speed and execution speed with the application of highly parallel processing technologies such as found in *graphic processing units* (GPU) and *field-programmable gate arrays* (FPGAs). These approaches are being used for a wide variety of applications in fields as diverse as automotive, financial, marketing, security, and many more.

A. Basics

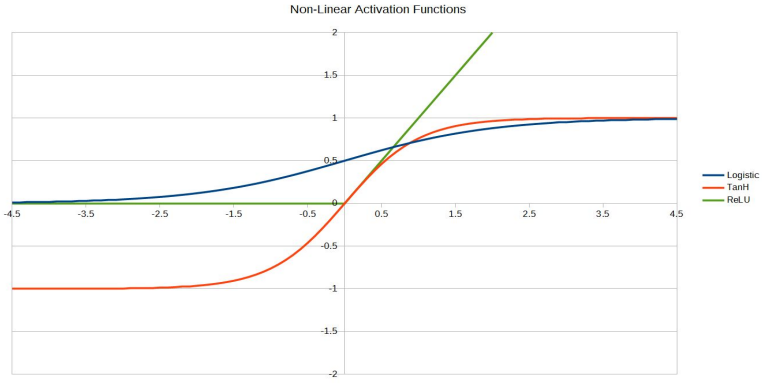
A neuron unit of a neural network is an entity that produces a non-linear response signal to a weighted sum of input signals originating from input-sensing units or earlier neuron units. Units can be organized in a variety of ways.





To keep the organization easier to understand, model, and manage, the network is usually organized into multiple layers, where each unit in one layer only receives inputs from the lower layer, and only provides input to the higher level. The layer connected directly to sensor signals or some other source of information is called the *Input Layer*. The layer proving output signals, such as signal/object classification, or feature location within a larger image, is called the *Output Layer*. Most networks that are part of a meaningful application have one or more additional layers in between the input and output layers. Each of these layers are traditionally called *Hidden Layers*, because they are not directly observing nor providing any outside signal. Units in one layer may be connected to all units in another layer, in which case the layer is called a *Fully-Connected* layer. In other networks, units in a layer may be only connected to a subset of units. This subset is often termed a *Receptive Field*, especially when it is a spatially compact sample of a larger image or data set.

The output activation signal of a neural net unit is typically a scalar value represented by either a real number within a modest range (-10 to +10) or a scaled integer, usually 8 to 32 bits in size. Each above the Input Layer unit receives inputs from one or more other units. The activation signal is modified by a weight factor, usually modeled as a simple proportional factor with a small value, typically -1 to +1 for real-valued signal representations, or a small fixed-point scaled-integer factor. When modelled with simple linear proportional weights, the total weighted input to a unit can be represented by the vector dot product of one vector holding the output activation levels of the connected units and a second vector holding the corresponding weight values: ($z = \mathbf{w} \cdot \mathbf{x}$). This total weighted input value z is then taken as the input to the unit's activation function. While purely linear activation functions are possible, they do not capale of performing sophisticated discrimination and other interesting tasks. Therefore, some form of non-linear function is needed. To date, the most common approaches are a *sigmoid* function, or a rectified-linear function, also known as *ReLU*. The sigmoid



function received a lot of attention because this

S-shaped curve models the activation of biological neurons reasonably well, which do not respond much to low levels of input stimulus, begin to increase in response to some level of input, then begin to saturate as the input level keeps increasing.

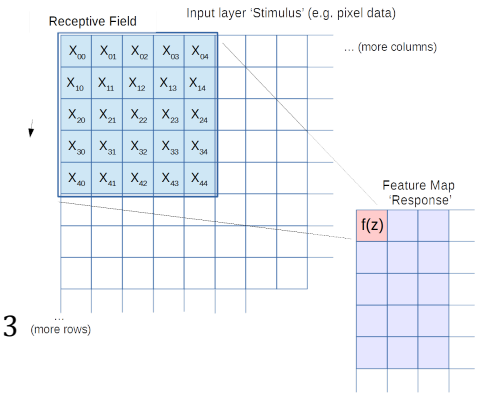
The two varieties of the sigmoid employed are the logistic function $1/(1 + e^{-z})$ whis is alway positive from 0 to 1, and the hyperbolic tangent function $\tanh(z) = (e^{z} - e^{-z}) / (e^{z} + e^{-z})$, which is symmetric about the axis and ranges from -1 to +1. Perhaps more recently, the ReLU function max(0, z) has been shown to be equally if not better-performing in many applications with adequate training with obvious benefits in terms of lower hardware resources and/or execution time cost.

B. Convolutional Neural Networks

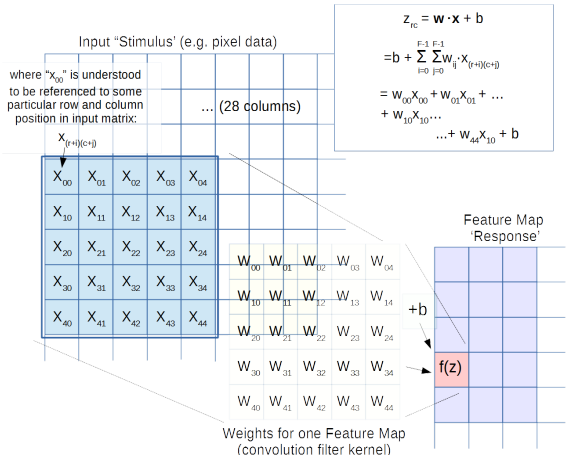
Convolutional neural networks use three basic ideas: *local receptive fields*, *shared weights*, and *pooling* to make deep networks practical to develop and use in practice.

a. Local Receptive Fields

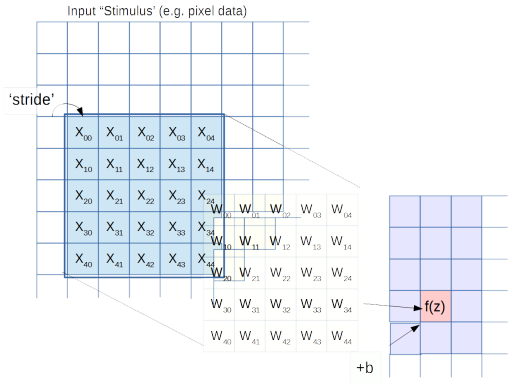
Each input pixel is connected to a layer of hidden neurons. The connections are made in small localized regions of the input image. Each neuron in the first hidden layer is connected to a small region (e.g. 5 by 5) of input pixels, as illustrated here:



The region in the input image is called the *Local Receptive Field* for that hidden neuron. Each of the connections to the input units has a weight, and the hidden neuron itself has an overall bias. This particular hidden neuron can be defined as learning to analyze its particular local receptive field.



In effect, one may imagine this Local Receptive Field sliding over entire input image in small increments, which is known as the ***Stride Length***, and feeding the total weighted input to the units in the first hidden layer. For an input 28 by 28 pixel image, and a 5 by 5 local receptive fields, there will be 24 by 24 neurons in the hidden layer. This is because one can only move the local receptive field 23 neurons across and 23 neurons down from the origin position, moving from the top left of the input image to the bottom right.



b. Shared Weights and Biases

Each hidden neuron in the convolutional layer has a bias and same set of weights as all other units in the same layer connected to their respective local receptive fields. The output is mathematically represented below:

$$a_{rc} = \sigma(z_{rc})$$

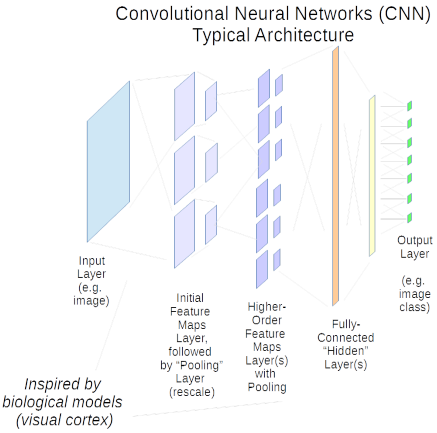
$$z_{rc} = \mathbf{w} \cdot \mathbf{x} + b$$

$$= b + \sum_{i=0}^{F-1} \sum_{j=0}^{F-1} w_{ij} \cdot x_{(r+i)(c+j)}$$

$$= w_{00}x_{00} + w_{01}x_{01} + \ldots + w_{10}x_{10} \ldots + w_{44}x_{10} + b$$

Given some non-uniformity in the weights, the weighted input will be stronger or weaker depending on how well the weights correspond to a particular pattern of activation in the previous layer at the location of the receptive field. We can say that the weights are used to detect a specific feature at each location in the input layer, forming a type of map of where the pattern is strongly present, absent, or anti-correlated The map from the input layer to an hidden layer is usually called a ***Feature Map***. All weights defining the feature map are the **Shared Weights**. A particular ***Shared Bias*** value may be used for all units in the feature map, or individual bias values may be trained. All the shared weights and bias are often said to define a **Filter**.

In practice, most CNNs use multiple layers of convolutional layers, each with multiple feature maps with distinct sets of shared weights. Later layers of CNNs often use fully-connected layers for final output, as illustrated here.



c. Pooling Layers

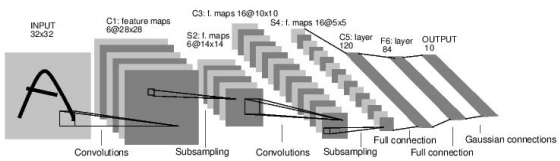
Pooling layers exist alongside the convolutional layers. They are used immediately after convolutional layers to downsample the previous layer's output to improve position invariance (ability to detect a feature wherever it occurs), improve generalization, and to reduce the computing resources and memory by having fewer units and weights in the later layers. Often in CNNs, the *Max Pooling* procedure is used, whereby the maximum activation value within a 2 by 2 region of the convolutional layer is selected and passed on to the next layer. The activation value selected may be adjusted with a bias, but it is common to omit biasing in this layer. For a 28 by 28 output convolutional layer, after max pooling, we have a 14 by 14 neuron layer.

d. Summary of Convolution Steps in LeNet-5

LeNet-5 is a well-known example of a CNN. It begins with the input 32 by 32 pixels. This is followed by a convolution layer of 6 feature maps, each of 5 by 5 local receptive fields. This produces a 6 by 28 by 28 layer of hidden feature

neurons. Following max pooling applied to 2 by 2 regions across the 6 feature maps, the resulting layer is a 6 by 14 by 14 hidden feature neuron layer. Convolution is once again done, and this produces a 10 by 10 convolutional layers of 16 feature maps. Max pooling is repeated to produce a 16 by 5 by 5 hidden feature neuron layer.

The next connection represents a 1D array convolutional layer with 120 feature maps. This is connected to the Fully connected layer of 84 feature maps and finally to the final layer of the connection. This Output, fully connected, layer connects every neuron from the max-pooled layer to every one of the 10 output neurons (digit 0 to 9).



We chose a similar structure for our final project, but simplified the overall network by having fewer feature maps and fewer fully-connected neurons. Our specific design is discussed in the Methodology section which follows below.

II. Methodology

For our final project we wanted to implement hardware acceleration of a CNN. CNNs are highly effective while having modest numbers of neuron connection weights in the early convolutional layers, while the convolution operation itself is repeated many times over the input data and earlier layer outputs. This scenario is ideal for hardware parallelization, which is why GPU coprocessors and FPGA hardware have both been used to implement key parts of CNN architectures. Reconfigurable FPGAs provide another capability that might be useful if not essential for managing very large input datasets: ability to reconfigure the hardware to optimize performance for layers and connections organized in different ways.[2] Such a variation is quite common: lower layers have spatially-compact receptive fields and widely-shared weights, middle layers implement pooling or sample number of lower-level feature maps. Higher layers may be more widely-distributed connections, or even a fully-connected layer.

As described above, there are two key calculations that are performed by neural network units: weighted average of inputs, followed by a nonlinear function. Of these, the function that offers the best opportunity to employ highly parallel computing resources is the weighted input operation. When one considers that the CNN use of shared weights means that the weight values do not need be repeatedly passed to the hardware, there is even more opportunity to make effective use of configurable hardware. For relatively-fixed applications, it is possible to lock in the weights in a custom hardware design through the use of *read-only memory* (ROM) *look-up tables* (LUTs). During class, an intriguing method of processing weighted averages called Distributed Arithmetic was briefly covered and described in some detail in a class handout[3]. We did not attempt to implement this method, but we would encourage future students to consider implementing it as a good method to accelerate weighting factors in a variety of applications.

A. LeNet-5

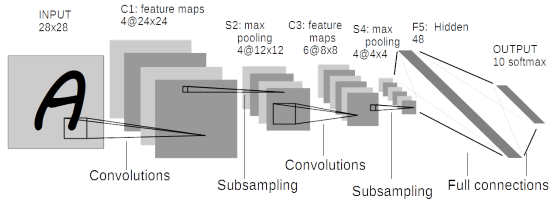
We chose to implement the dot product operation to accelerate a simplified version of *LeNet-5*, a very famous CNN architecture. The LeNet architecture was first introduced by

LeCun et al. in their widely-cited 1998 paper, *Gradient-Based Learning Applied to Document Recognition*.[4] It can be useful for pattern inference. The test dataset is the MNIST dataset described further below, which contains a total of 70,000 grayscale images of handwritten digits as one might find on a postal envelope or paper forms filled out by hand[5] . The LeNet-5 has 3 convolutional layers (C1, C3, C5), 2 subsampling "max pooling" layers (S2, S4) , a fully-connected layer (F6) and an output layer.

In each "feature map", nodes are organized in a 2D image fashion and have shared weights that are strongly stimulated by a specific feature at a corresponding location in the lower layer. Each node in a convolutional layer is identified by (column index, row index, feature map index). Each pooling layer only connects to a small receptive field in its corresponding feature map and implement a simple function such as maximum, or average within the small "pool" of inputs. There is no crossover between the convolution layer and subsampling Layer.

B. Our CNN Design

In our implementation, we kept the input image at its original 28 by 28 pixel size. We convert the 8-bit 0 to 255 grayscale image levels to a fixed-point value ranging from -1.0 for 0 to +1.0 for 255.



Simplified CNN for Handwritten Digit Classification

That input layer is then scanned by four feature map layers where each of their units observes a specific 5 by 5 pixel receptive field region of the original image. The size of 5 of the receptive field in each dimension means that there are 24 (28 - 5 + 1) units by 24 units in each of the four feature maps of the first convolutional layer. These are downsampled through max pooling by a layer with four sets of 12 by 12 units, each of which returns the maximum value of a 2 by 2 receptive field with no overlap (stride of 2).

The second convolutional layer produces six feature maps using 100 weights each (5 by 5 within one feature map, and all four feature maps scanned at the same corresponding location). There are eight distinct receptive field positions along each dimension, for a total of 8 by 8 units, downsampled again through max pooling to produce a total of six 4 by 4 little feature maps. These feature maps detect more-abstract, higher-order "features of features" compared to the previous layer.

Next, all 96 units of the second max pooling layer are fully connected to a one dimensional hidden layer of 48 units. This set of connections has the majority of the network's trainable weights (96 * 48 = 4,608).

Lastly, the ten possible digit classification values are represented by ten more units fully-connected to the 48 hidden layer units. The ten activations are then adjusted by the "soft max" (normalized exponential $e^{z_j}/\Sigma e^{z_k}$) function to emphasize the "winner" classification and provide the result as a rough likelihood percentage.

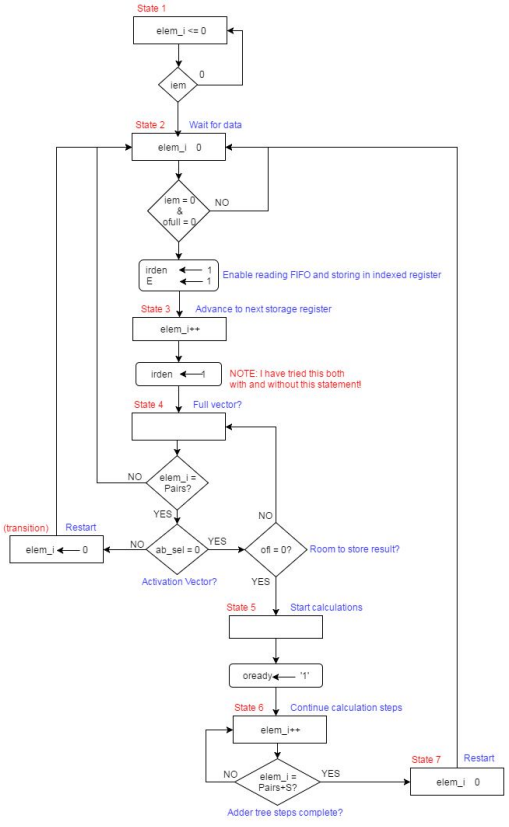
C. Our Hardware Design

To support the first convolutional layer processing, we needed to pass a set of 25 weights to the hardware, once for each feature

map. Then we can send many 5 by 5 pixel sets of the image data plus a bias term to the hardware and receive back a single value as the weighted input for the feature map unit. These 5 by 5 matrix values for both the weights and image pixel levels can be reshaped into a one-dimensional vector with no loss of meaning, as long as each vector has the same positional mapping of elements to their respective matrices. We extended the vector length to 26 elements to allow space for the bias term and to use an even number, since the AXI Full peripheral interface is 32 bits wide bits and we chose to represent weights and unit activations in a 16 bit fixed point format with 10 fractional bits: FX [16 10]. This encoding was determined experimentally to be suitable for our application, as described in a later section. Therefore, we needed to pack the 26 values for each vector into 13 pairs of 16-bit numbers and feed them into the hardware. For later layers, we need 100 or 96 element vectors (5 by 5 by 4, or 4 by 4 by 6) or 48 (hidden layer to output unit). While these were not attempted to be passed to the AXI peripheral, it would be possible to break them up into pieces that are each no more than 26 elements long. An alternative approach would be to make the peripheral support large vectors like this and provide a means to pass it smaller vectors efficiently (e.g. use special address to pass the length, or any write to a special address terminates the vector at the current length).

We chose the AXI Full peripheral bus interface to communicate with the main ARM processor system, as we were familiar with it from earlier class exercises and laboratory assignments and it can support fast DMA memory transfers, which we might want to utilize in the future. We wanted to decouple the sending of weights (w) and unit activation vectors (x), so we determined that our AXI interface would monitor the access write address and and use it to set a flag representing the intended identity (and corresponding register storage location) of the vector. A zero in the address bit 2 represents a unit activation vector, while a 1 in the bit 2 of the address (e.g. base address + 4) indicates the data is a weight vector. This flag is called 'ab_sel' in our VHDL statements, where the "a" vector is the unit activation values vector and the "b" vector is the connection weights vector.

We chose the AXI Full FIFO interface model from class, as it allows us to send the data either as separate pairs of values or through many-value DMA burst without requirement to identify the specific index of each vector element. This means that we needed to implement our own method of counting input value pairs and storing each of them in the correct activation or weight register. We did this by implementing an element count 'elem' in a "red clock" *finite state machine* (FSM).



When we have data available in the FIFO, we enable reading it and latching it into an appropriate register enabled by a demultiplexed select line derived from the counter ‘elem’ and the ‘ab_sel’ vector select flag from the write address as described above.

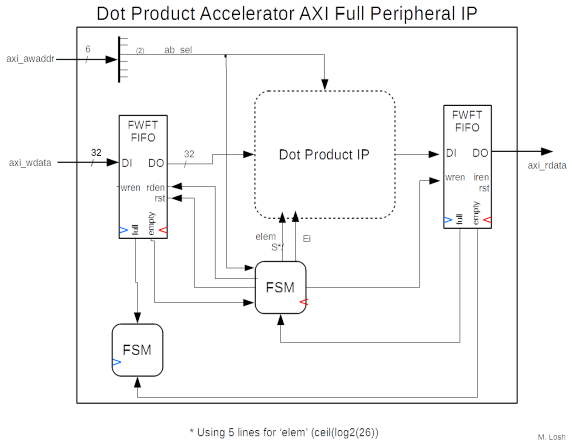
Recalculation of the dot product is only completed when the activation vector is sent and all expected pairs of values received. The outputs of the corresponding weight ‘w’ and activation ’x’ register are fed to a signed multiply operation resulting in a 32-bit product with fixed-point FX [32 20] scaling. The product terms are then added together using an adder tree structure that requires five levels (ceil(log2(26)) = 5). An early version of this adder was un-clocked, but a staged version with registers at each level was implemented.

When all of the input vector is received and the calculations have had enough time to pass through the staged adder tree, we enable the result to be written into the output FIFO, which then makes it available to the main processor system to be read by the embedded application software, which implements the other parts of the network (ReLU function, max pooling, softmax output) and unit/weight selection.

D. Hardware Circuit Details

Dot Product Accelerator AXI Full Peripheral IP

We implemented our hardware design as a packaged *Intellectual Property* (IP) that utilizes the AXI Full interface of the Zynq chip family, which supports interconnecting AXI Masters to Memory-Mapped AXI Slave peripheral co-processors in the *Programmable Logic* (PL) fabric of the chip. Its structure consisted of the Dot Product IP, interfaced with the input and output FIFOs, and two Finite state machines, as was previously used in class assignments. The “Red” FSM is in charge of the *irden, rst, owren, elem, Ei, ab_sel, ofull, iempty* signals. The “Blue” FSM is in charge of the *ifull, oempty* signals. The *ab_sel* signal is sourced from the third bit of the *axi_awaddr* address of the AXI interface. It is used as an address for the weights.

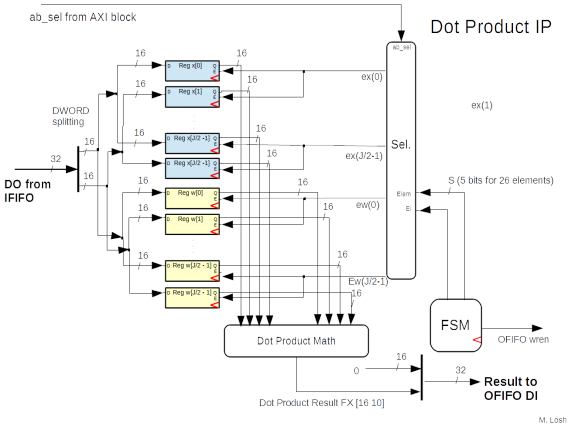


a. DOT PRODUCT IP

This is the heart of the Dot Product Accelerator IP. It basically functions to produce the total weighted input (z) to the non-linear function of a higher-layer neuron unit. This IP design consists of Registers, Input Register logic SEL, Red Finite State Machine, and Dot Product Math.

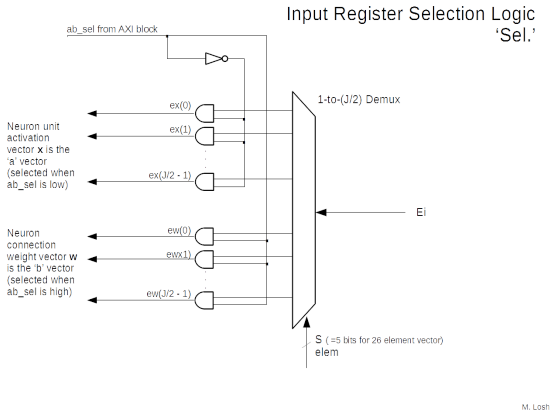
By design, the weights (w) are first pulled from the AXI bus and place in the input FIFO. The IP reads these inputs from the input FIFO and stores them into the designated registers. The second batch of inputs from the FIFO, represent the inputs (x), and are stored in their designated registers. The registers are accessed based on SEL circuit and sent into the Dot Product Math Circuit. The output of this circuit is fed to the output FIFO. The Red FSM issues the Elem index value and Ei enable line that controls the SEL circuit, which in turn enables specific

Registers for the inputs *x* and the weights *w*. Each register is 16 bits, thus 2 registers are enabled simultaneously for each pair of values arriving from the 32 bit-wide FIFO and they are J (26 in the current implementation) in number. The 32 bit word from the input FIFO is split into 2 16 bits and sent to a separate register, either of the input *x* registers or the weights *w* registers.



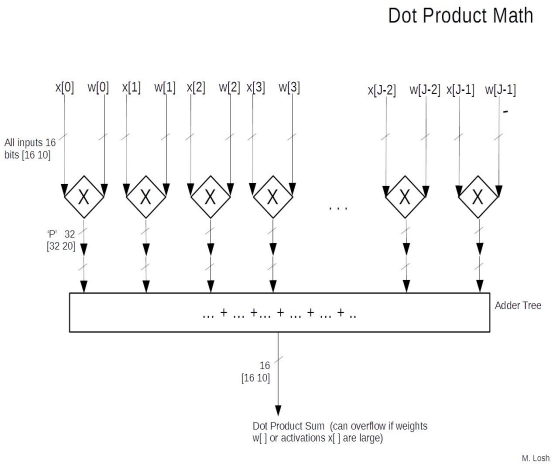
b. SEL

This is an input register selection logic circuit. It functions as a Demultiplexer (1-to-J/2 Demux). The input *Ei*, with an eight bit selection *elem,* produces [J/2] outputs. These outputs are in two sets: one for the inputs *x,* and another for the weights *w.* The *ab_sel* input from the AXI block when true allows a weights output to be true, while the *not ab_sel* signal allows the other set of outputs. That is, if *ab_sel* is 1, one of the weights enable signal is activated, otherwise, one of the other outputs is selected to be active. The corresponding outputs of the Demux, based on the *Ei, Elem* and *ab_sel* are sent to the respective registers in the Dot Product IP. The select line is only active if the general Ei signal is active.

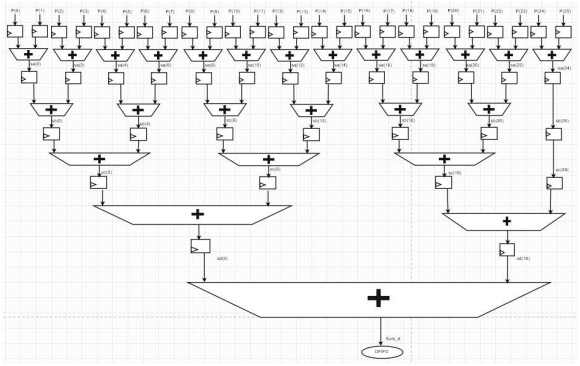


c. DOT PRODUCT MATH

The dot product math circuit consist of the multiplier and adder tree. All [J] inputs, each of 16 bit values are multiplied by their corresponding 16 bit weights value. This serves as the inputs for the adder tree, to produce the Dot Product Sum. The circuit was designed such that the input will only overflow when either the weights *w* or activations *x* are large, and in our demonstration application, value are kept small.



The Adder tree consist of 5 stages, each with registers. This ensures a potentially faster operation through pipelining. The final output of this circuit is sent to the output FIFO.

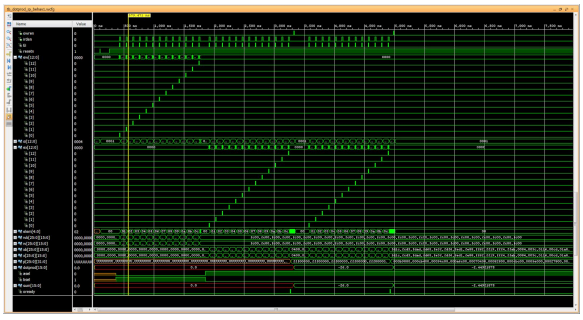


When we had development issues with our AXI peripheral not cooperating with our main processor, we used the tb_AXI testbench provided to the class which implements most of the AXI signals and bus interactions realistically so we could identify timing and handshaking issues. We used this testbench at length to refine the "red" clock Finite State Machine timing and interaction with the remainder of our hardware design.

III. Experimental Setup

A. VHDL Testbenches

To test and validate our hardware design we used two levels of VHDL testbenches. The simple "tb_dotprod_id" test bench allowed us to test the interior portion of the design without the buffer FIFOs. We set up a data value pattern on the simulated "IFIFO out" signal bus and just used clock timing to determine when to change the signal to other values to emulate the way the main processor would feed in data to our circuit through the input FIFO. Similarly, we would monitor the output of the calculations that would normally go to the input side of the output FIFO. This level of testbench was useful to get the register select circuit working that determines where each of the sent vector elements would be stored.



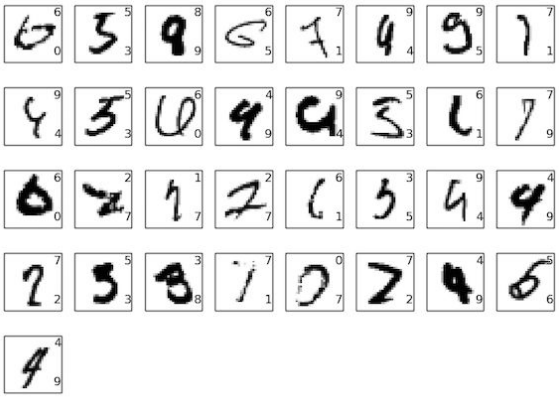
B. MNIST dataset

The MNIST dataset is a well-studied and easily understood dataset used in the computer vision and machine learning literature. This dataset classifies handwritten digits 0-9[6]. With a total of 70,000 images, images are split between 60,000 images in the Training Data and 10,000 images in the Evaluation Data. The writing is taken from examples from hundreds of different writers from two pools: high school students and Census Bureau employees. None of the people used for the training set were used in the evaluation set. The training data is taken from a set Common splits include the standard 60,000/10,000, 75%/25%, and 66.6%/33.3%. During training, it is common for a subset of the entire training set to be kept out of direct network training and used for interim performance evaluation. While the samples are distinct, they originate from the same pool of writers, so one can assume that the trained network will do better with this reserved subset that the evaluation set, which comes from a completely different pool of writers. Our objective is not to produce the best possible handwritten digit classifier possible (or even the best given the constraints on network size), but to provide a test ground for hardware designs

useful for CNN acceleration, so we did not formally test our network against the evaluation set.

Each digit in the MNIST dataset is a 28 by 28 pixel image, but LeCunn et. al padded the images to a size of 32 by 32 pixels to help the network tolerate shifted images, but we used the images as-is, and scaled the 0 to 255 intensity values to a fixed-point range of -1.0 to +1.0 when loading into our network.

Each image in the training and evaluation sets have associated classification labels. In other words, the "correct" interpretation or at least the intended meaning of the digit (some examples are quite sloppy, as you might expect from a wide variety of people).



(From Nielsen[7])

C. Neural Network Training

At first, we attempted to use sample Python code from Michael Nielsen's book Neural Networks and Deep Learning[7]. While the code ran and produced weights, when we imported these into our C program, it did not produce reasonable results, perhaps due to a transposition effect that we did not resolve. We implemented our own back-propagation functions in the C code to train the network weights and bias values. We used some common techniques in the field of neural networks to improve the training process: we adjusted the learning rate from a more aggressive starting rate to a more gradual rate over the training process. We also implemented weight decay where we reduce the magnitude of all weights by a small percentage if the average size of all weights (as defined by root mean square) is above a threshold value. This keeps most weights in the -0.5 to 0.5 range and helps prevent over-generalizing from specific training examples. We used the first 50,000 images of the training set for the training, and the last 10,000 images in the training set as our "validation" set. After many minutes of training on a high-specification laptop (Intel Core i7), we could achieve validation accuracies around 98.0%. +/-0.6% out of 1000-randomly selected examples from the 10,000.

As we intended to implement our dot product accelerator in VHDL, we wanted to have the neural network to perform well with fixed-point numbers. We experimented with various encodings, and counted cases where the weighted input calculations overflowed the allowed encoding range of the number format in question. Because we trained our network using weight decay, the majority of weight values were fairly small, preventing overflow. We determined we had no overflows for a large sample of inputs when we used an FX [16 10] encoding and still keep fairly close to our original accuracy.

After training the network on a PC, the code and weights were copied into the Xilinx SDK environment. The C program for the embedded ARM processor system was adjusted to read an SD card for the the training/validation image set, image labels. The weights and network definition file was formatted as an include file that defines the network specification as a large hard-coded string. This string is parsed just as

an external file would be. In the future it would be convenient to make this network and weight configuration information readable also from the SD card.

IV. Neural Net Testing and Results

Our network implementation in C ran on our Zynq Zybo board with an SD card holding the image database and image classification labels. The application reads the card on startup and buffers it in DDR RAM. For timing purposes, we prepared a routine to process all 10,000 images in our validation data set with no planned output to the console that would rob performance. The classification accuracy was tracked (number "right" according to the label versus the total). Timing was measured by clock cycle counts returned by the XTime_GetTime( ) function and verified to be accurate by stopwatch.

Baseline performance for original "random" memory-access pattern version neural network (software only):
9832 out of 10000 correct, or 98.32% in 190.191 sec.

Performance for AXI dot product IP hardware-utilizing neural network:
9832 out of 10000 correct, or 98.32% in 183.854 sec.
(Note: the vectors were loaded into the AXI peripheral two elements at a time, not with DMA.)

Performance for streamlined dot-product structured neural network (software only):
9832 out of 10000 correct, or 98.32% in 128.728 sec.

V. Conclusions

While we were able to complete a hardware design that could take advantage of parallel processing, it is clear that overall system performance requires careful consideration of of all operations, including data transfers into and out co-processing units. With conventional explicit (non-DMA) 32-bit data writes, our dot product hardware speed boost was not sufficient to repay the time needed to set up the input data, at least compared to software-only dot-product optimized C code compiled for the ARM core in our Zynq chip, although our hardware implementation was was slightly faster than a more simplistic software implementation with a more random-access memory pattern.

Our overall application performance is quite impacted by the inherent requirement to pass large vectors of data out to our hardware peripheral repeatedly. Because of the way neural networks are configured, the same input pixel or intermediate activation level must be assessed many times to determine the state of a higher layer unit. In the case of our small network, the same input pixel (away from the image edge) would be sent as part of an activation vector one hundred times (!) during first-layer processing to the hardware because it is a member of twenty-five different receptive field configurations for each of the four feature maps. In the second layer of our network, the repetition is even worse with six feature maps, resulting in most activation values being sent 150 times. Obviously it would be better if the data could be sent fewer times.

Another aspect to fast hardware performance only partly addressed in our hardware design is pipelining. We employed a coarse pipeline structure in our calculation subsystem, but did not design the input buffering registers to be fully pipelined. However, pipelining beyond use of input and output FIFOs may not be as useful since the nature of the dot product is already sequential, so a fast multiply-accumulator approach might be able to keep up with the AXI bus speed limitations.

Our design is partially parameterized with VHDL generic parameters (e.g. be able to request a specific implementation of any length vector). Specifically, the input buffer registers, multipliers, and "red" clock FSM all are generic configurable using "for .. generate" and "if … generate" statements, but the adder tree subsystem is not at this time. In principle adder tree could be made generic or use existing adder tree hardware blocks.

Given more time, we could have enjoyed exploring a variety of speed enhancements:

1. DMA memory transfer for weight sets and especially activation data vectors. These

would become increasingly useful with larger vectors, such as the 96 input weights needed for each fully-connected hidden layer unit. We begun development of DMA transfer routines but did not find time to debug and test them.

2. Increased parallelization: CNNs typically have multiple feature maps tied to an earlier layer, each with a distinct set of shared weights. After sending a particular 5x5 receptive field to the hardware, that vector could have been used in parallel dot product calculations for each feature map. The set of results could be read back in a short burst. While our small CNN only had four feature maps in one layer, and six in another, some CNNs have many more, so higher levels of parallelization would be very beneficial. This would require an array of weight vectors to be configured and accessed repeatedly.
3. Explore use of reconfigurable partitions with fixed lookup tables to implement "distributed arithmetic" methods of multiplying varying data by constant factors, which in our case are the network connection weights.
4. Expand the design to work with entire image sets or at least large strips of the input image at a time to reduce the high rates of data transfer repetition described above. The hardware could index within hardware memory buffers to fetch values repeatedly as needed from the sliding receptive field and buffer the calculation results in other hardware memory where it could be reused by later calculations without requiring slow communications back and forth with the main processor. This would remove the need to resend the same pixel/activation value as many as 150 times in our small network and manyfold more in even larger networks.

Acknowledgements.

The authors would like to thank Prof. Llamocca for debugging assistance and design suggestions and the opportunity to explore interesting applications of FPGAs. The authors also acknowledge Michael Nielsen for his very accessible introduction to neural networks, including CNNs.

VI. References

[1] Andrey Kurenkov, "A brief History of Neural Nets and Deep Learning".[http://www.andreykurenkov.com/writing/a-brief-history-of-neural-nets-and-deep-learning/]

[2] S. Sahin, Y. Becerikli and S. Yazici, "Neural Network Implementation in Hardware Using FPGAs," Department of Computer Eng., Kocaeli University, Izmit ,Turkey.

[3] Daniel Llamocca, "Notes - Unit 2", ECE495/595 class handout, Fall 2016.

[4] LeCun, et. al. "Gradient Based Learning Applied to Document Recognition", Nov. 1998, Proceedings of the IEEE.

[5] http://deeplearning.net/tutorial/lenet.html, copyright 2008--2010, LISA lab.

[6] The MNIST database of handwritten digits

[7] Michael Nielsen, Neural Networks and Deep Learning, Jan 2016, published as an online book: http://neuralnetworksanddeeplearning.com/index.html

[8] http://cs231n.github.io/convolutional-networks/